Cambridge Elements

Applied Category Theory

An Introduction to String Diagrams for Computer Scientists

Robin Piedeleu and Fabio Zanasi

ISSN 2633-1861 (online)
ISSN 2633-1853 (print)

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

Cambridge Elements

Elements in Applied Category Theory
edited by
Bob Coecke
Cambridge Quantum Ltd
Joshua Tan
University of Oxford

AN INTRODUCTION TO STRING DIAGRAMS FOR COMPUTER SCIENTISTS

Robin Piedeleu
University College London

Fabio Zanasi
University College London

CAMBRIDGE
UNIVERSITY PRESS

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

CAMBRIDGE
UNIVERSITY PRESS

Shaftesbury Road, Cambridge CB2 8EA, United Kingdom

One Liberty Plaza, 20th Floor, New York, NY 10006, USA

477 Williamstown Road, Port Melbourne, VIC 3207, Australia

314–321, 3rd Floor, Plot 3, Splendor Forum, Jasola District Centre,
New Delhi – 110025, India

103 Penang Road, #05–06/07, Visioncrest Commercial, Singapore 238467

Cambridge University Press is part of Cambridge University Press & Assessment,
a department of the University of Cambridge.

We share the University's mission to contribute to society through the pursuit of
education, learning and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/9781009625708

DOI: 10.1017/9781009625715

© Robin Piedeleu and Fabio Zanasi 2025

This publication is in copyright. Subject to statutory exception and to the provisions
of relevant collective licensing agreements, with the exception of the Creative
Commons version the link for which is provided below, no reproduction of any part
may take place without the written permission of Cambridge University Press &
Assessment.

An online version of this work is published at doi.org/10.1017/9781009625715
under a Creative Commons Open Access license CC-BY-NC-ND 4.0 which permits
re-use, distribution and reproduction in any medium for non-commercial
purposes providing appropriate credit to the original work is given.
You may not distribute derivative works without permission. To view a
copy of this license, visit https://creativecommons.org/licenses/by-nc-nd/4.0

When citing this work, please include a reference to the DOI 10.1017/9781009625715

First published 2025

A catalogue record for this publication is available from the British Library

ISBN 978-1-009-62570-8 Hardback
ISBN 978-1-009-62574-6 Paperback
ISSN 2633-1861 (online)
ISSN 2633-1853 (print)

Cambridge University Press & Assessment has no responsibility for the persistence
or accuracy of URLs for external or third-party internet websites referred to in this
publication and does not guarantee that any content on such websites is, or will
remain, accurate or appropriate.

For EU product safety concerns, contact us at Calle de José Abascal, 56, 1°, 28003
Madrid, Spain, or email eugpsr@cambridge.org.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

An Introduction to String Diagrams for Computer Scientists

Elements in Applied Category Theory

DOI: 10.1017/9781009625715
First published online: April 2025

Robin Piedeleu
University College London

Fabio Zanasi
University College London

Author for correspondence: Fabio Zanasi, f.zanasi@ucl.ac.uk

Abstract: String diagrams are a powerful graphical language used to represent computational phenomena across diverse scientific fields, including computer science, physics, and linguistics, amongst others. The appeal of string diagrams lies in their multi-faceted nature: they offer a simple, visual representation of complex scientific ideas, while also allowing rigorous mathematical treatment. Originating in category theory, string diagrams have since evolved into a versatile formalism, extending well beyond their abstract algebraic roots and offering alternative entry points to their study. This Element provides an accessible introduction to string diagrams from the perspective of computer science. Rather than starting from categorical concepts, the authors draw on intuitions from formal language theory, treating string diagrams as a syntax with its own semantics. They survey the basic theory, outline fundamental principles, and highlight modern applications of string diagrams in different fields. This title is also available as open access on Cambridge Core.

Keywords: string diagram, monoidal category, semantics, programming languages, category theory

© Robin Piedeleu and Fabio Zanasi 2025

ISBNs: 9781009625708 (HB), 9781009625746 (PB), 9781009625715 (OC)
ISSNs: 2633-1861 (online), 2633-1853 (print)

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

Contents

1 The Case for String Diagrams 1

2 String Diagrams as Syntax 6

3 String Diagrams as Graphs 28

4 Categories of String Diagrams 34

5 Semantics 53

6 Other Trends in String Diagram Theory 79

7 String Diagrams in Science: Some Applications 88

Appendix: Category Theory: The Bare Minimum 100

References 103

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

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's *syntax*. This syntax is often specified inductively, as a context-free grammar. For instance, the grammar shown in (1.1) defines the syntax of a very elementary imperative programming language, where variables $x, y, \ldots$ and natural numbers $n \in \mathbb{N}$ may occur:

$$\begin{array}{rcl} b & ::= & \mathit{True} \mid x = y \mid x = n \mid \neg b \mid b \wedge b \mid b \vee b \\ p & ::= & \mathit{skip} \mid x{:=}n \mid x{:=}y \mid x{:=}y{+}1 \mid \mathit{while}\ b\ \mathit{do}\ p \mid p\,;p \end{array} \tag{1.1}$$

With the second row of the grammar, we can write arbitrary programs p featuring assignment of a value to a variable, while loops, and program concatenation. In particular, while loops will depend upon a Boolean expression b, 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 according to the grammar. For computer scientists, who are interested in formal analysis of programs, this information has deeper consequences: it gives us a powerful tool to prove *mathematical properties* of the language by *induction* over the syntax. This principle is a generalisation of how we usually reason about the natural numbers. Indeed, the set $\mathbb{N}$ of natural numbers can also be specified via a grammar:

$$n \quad ::= \quad 0 \mid n + 1 \tag{1.2}$$

When proving properties of $\mathbb{N}$ by induction, what we are really doing is reasoning by case analysis on the clauses of grammar (1.2). For instance, suppose we wish to prove by induction that, for each $n \in \mathbb{N}$, $n + 1 \leq 2^n$. In the base case, we assume that n is 0; we can verify that $0 + 1 \leq 2^0 = 1$. In the inductive step, we consider the case that n is $n' + 1$ for some n'. If we assume $n' + 1 \leq 2^{n'}$, then we can show the statement for $n = n' + 1$, as follows: $(n' + 1) + 1 \leq 2^{n'} + 1 \leq 2^{n'} + 2^{n'} = 2^{n'+1}$.

In the same way, we can reason by induction on programs, whenever their syntax is specified by a grammar such as (1.1). For example, we can prove that a certain property P holds for any program p defined by (1.1), as follows: first, we need to show that P holds for *skip*, $x := n$, $x := y$, and $x := y+1$. Then, assuming P holds for p, we show that it holds for *while b to p*. Finally, assuming P holds for p and p', we show that it holds for $p\,;p'$.

This style of reasoning is extremely useful for several tasks. For instance, we may prove by induction important properties of our program, such as

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

its *correctness*, *safety*, or *liveness*, as studied in the research area of *formal verification*. We may also define the *semantics* by induction, that is, 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 object that abstracts its intended behaviour. An inductively defined semantics is particularly important because it enables *compositional* (or *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 the 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 behaviour. For instance, in any sensible semantics, the program constructs; and *skip* of the grammar (1.1) 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 formalism. Think for instance about data 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 data at a time, or a deadlock could occur. Sophisticated forms of interaction, such as entanglement in quantum processes, or conditional (in)dependence between random variables in 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 would otherwise be difficult to encode into a conventional textual notation. Notable examples of these formalisms include electrical and digital

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

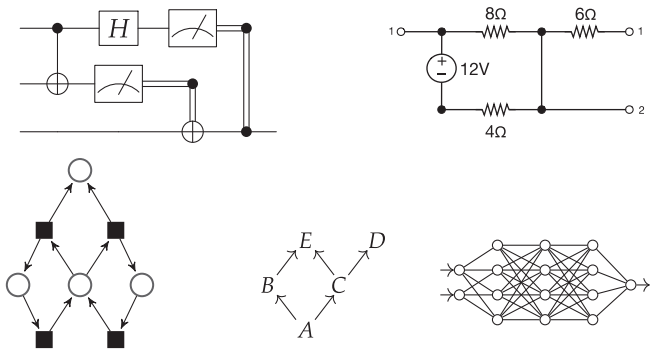


Figure 1.1 Some examples of graphical formalisms: a quantum circuit, an electric circuit, a Petri net, a Bayesian network, and a neural network.

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. (See Figure 1.1.)

On the other hand, graphical models have clear drawbacks compared to syntactically defined formal languages. Our ability to reason mathematically about combinatorial, graph-like structures is limited. We typically miss a formal theory of how to *decompose* these models into simpler components, and also of how to *compose* them together to create more complex models. In short, graphical models are often treated as monolithic rather than modular entities. In turn, this means that we cannot use induction on the model structure to prove their properties, as we would with a standard program syntax. Crucial features of program analysis, such as the definition of a compositional semantics, and the investigation of algebraic structures underpinning model behaviour, face significant obstacles when adapted to graphical formalisms.

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 far 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 also be regarded as a formal *syntax*; we can think of them as made of elementary components (akin to the gates of a circuit, but a lot more general than that), composed via syntactically defined operations.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

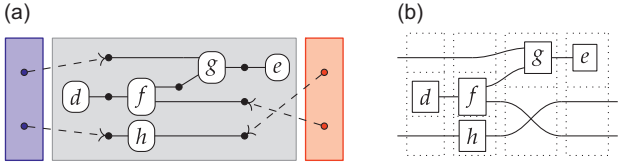


Figure 1.2 An example of a string diagram regarded as a (hyper)graph **(a)**, with the side boxes signalling the interfaces for composing with other string diagrams, and the same string diagram regarded as a piece of syntax **(b)**, with dotted boxes placed to emphasise where elementary components compose, vertically and horizontally.

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 (Figure 1.2). 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, we can assign a semantics to string diagrams compositionally, as a *functor* between categories. This gives us a modular way to specify and reason about the behaviour of the models that they represent.

String Diagrams in Contemporary Research

Thanks to their versatility, string diagrams are increasingly adopted as a reasoning tool by scientists across various research fields. We may identify two major trends in the use of string diagrams: 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.

Within the first trend, string diagrammatic approaches have enabled the adoption of compositional semantics and algebraic reasoning for graphical formalisms that previously lacked these features. Examples include Petri nets [16], linear dynamical systems [6, 20, 55], quantum circuits [109], electrical circuits [11], and Bayesian networks [51, 57, 75], amongst others (see Figure 1.3). Besides providing a unifying mathematical perspective on these models, string diagrams have also demonstrated the ability to produce tangible outcomes, employable at industrial scale. A convincing example is the ZX-calculus, a diagrammatic language that generalises quantum circuits. It now serves as the basis for the development of state-of-the-art quantum circuit optimisation algorithms [48], and is seeing widespread adoption by companies dealing with quantum computing.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

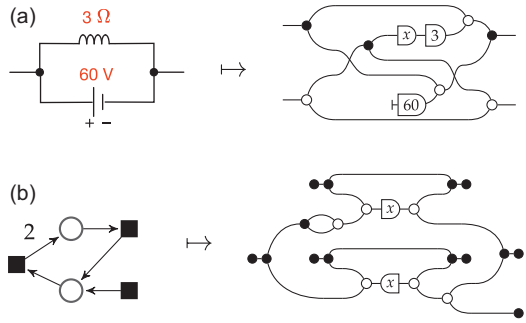


Figure 1.3 String diagrams representing the behaviour of an electrical circuit **(a)** and a Petri net **(b)**. The abstract perspective offered by the diagrammatic approach reveals that seemingly very different phenomena may be captured via the same set of elementary components.

Figure 1.4 A major appeal of string diagrams is *resource sensitivity*: they uncover any implicit assumption on how resources are handled during a computation. For instance, in these string diagrams resources x and y are being fed to processes f and g. Suppose applying g to x is an expensive computation. In the scenario where f receives the value $g(x)$ twice, we are able to distinguish the case where we duplicate x and then feed it to g **(a)**, and the more efficient way, where we duplicate $g(x)$ **(b)**. Note that traditional algebraic syntax would represent both cases as the same term, $f(g(x), g(x), y)$.

As examples of the second trend, string diagrams have been instrumental in the development of compilers [95] for higher-order functional languages, and in a provably sound algorithm for reverse-mode automatic differentiation [3]. In both these examples, string diagrams serve as an intermediate formalism that sits between high-level programming languages and lower-level implementations. As the former, they can be manipulated syntactically. As the latter, they explicitly represent information propagation and other structural properties of systems (see Figure 1.4).

Outline

This introduction provides a basic overview of string diagrams and their applications. 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 also be thought of as certain (hyper)graphs, thus providing an equivalent combinatorial perspective on these

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

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 detail in this introduction. Finally, Section 7 is a non-exhaustive list of applications of string diagrams, both inside and outside of computer science.

The use of category theory is kept to a bare minimum, and we have prioritised intuition over technicalities whenever possible. For the reader's convenience, we have included an appendix containing the most rudimentary definitions of category theory. However, it should not be treated as an introduction to the topic, for which we recommend [86].

2 String Diagrams as Syntax

We have seen a couple of examples, (1.1) and (1.2), of how to specify expressions of a formal language via a grammar. In order to generalise this technique to diagrammatic expressions, it is best viewed through the lens 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. Each operation comes with its *type*, which remained implicit in (1.1) and (1.2). For instance, we may regard; (program composition) in (1.1) as a *binary* operation, which takes as inputs two programs p and p' as arguments and returns as output a program $p; p'$ as value. The type of this operation is thus $\mathit{program} \times \mathit{program} \to \mathit{program}$. Analogously, skip, $x := y$, $x := y+1$, and $x := n$ may be seen as constants (operations with no inputs) of type *program*, and the while loop yields an operation of type $\mathit{boolean} \times \mathit{program} \to \mathit{program}$: given a Boolean expression b and a program p, we obtain a program $\mathit{while}\ b\ \mathit{do}\ p$.

This example suggests that, more generally, a signature Σ should consist of two pieces of information: a set Σ_1 of generating operations, and a set Σ_0 of generating 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 (1.1). In algebra, such expressions are called Σ-*terms*.

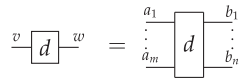
This process works in a fairly similar way for string diagrams, with some key differences. String diagrams will be built from signatures, except that now the type of operations may feature multiple outputs as well as multiple inputs, as displayed pictorially in (2.1). We will also see that *variables*, usually a building block of Σ-terms, are not a native concept, but rather something

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

that may be encoded in the diagrammatic representation. More on this point in Example 2.14, and Remarks 2.15 and 5.2.

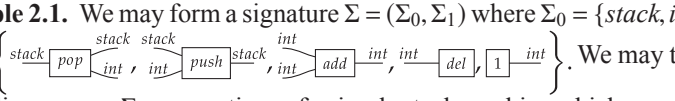
Signatures

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 *generating operations*. We will refer to either simply as generators when it is clear from the context whether we mean a generating object or operation. Each generating operation d has a type $v \to w$, where $v \in \Sigma_0^\star$ (the set of words on alphabet Σ_0) is called the *arity*, and $w \in \Sigma_0^\star$ the *co-arity* of d. Pictorially, an operation d with arity $v = a_1 \dots a_m$ and co-arity $w = b_1 \dots b_n$ will be represented as



(2.1)

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

Example 2.1. We may form a signature $\Sigma = (\Sigma_0, \Sigma_1)$ where $\Sigma_0 = \{stack, int\}$ and $\Sigma_1 =$ 

. We may think of string diagrams on Σ as operations of a simple stack machine which can perform simple arithmetic on integers.

Terms

Terms are generated by combining the generators 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 follows:

- 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 generating 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'). As a preliminary

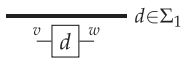
Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

intuition, we may think of the built-in operations as the minimal structure needed to express graphical manipulations of terms, such as 'stretching' a wire or crossing two wires. Fixing a signature $\Sigma = (\Sigma_0, \Sigma_1)$, the Σ-terms are generated by a few simple derivation rules or term formation rules: these are written in the form

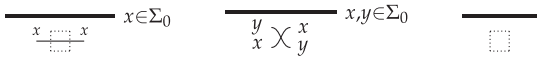
$$\frac{\text{list of terms}}{\text{term}}\ \text{condition}$$

Here, we may regard the list of terms above the line as hypotheses needed to form the term in conclusion of the rule, below the line, provided that the side-condition is satisfied. For symmetric monoidal diagrams, the rules are as follows.

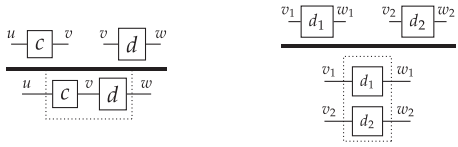
1. First, we have that every generating operation in Σ_1 yields a term:



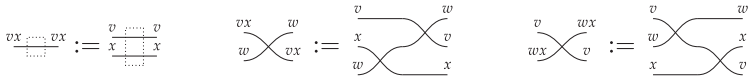
2. Next, each built-in operation (from left to right below: identity, symmetry, and null) also yields a term:



3. For the inductive step, a new term may be built by combining two terms, either sequentially (left) or in parallel (right). Note that, for sequential composition, the output of the leftmost term needs to match the input of the rightmost term. For parallel composition, there is no such requirement, and the resulting term has input (output) the concatenation of the words forming the inputs (outputs) of the starting terms.



Using the composition rules, we can define by induction 'identities' and 'symmetries' for arbitrary words v, w in Σ_0^*.



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.

Another important point: notice that *null*, the identity over the empty word ϵ is not depicted (or is depicted as the *empty diagram*), which is shown in the

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

preceding 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 co-arity. A term of type $d : \epsilon \to w$, sometimes called a *state*, has an empty left boundary,

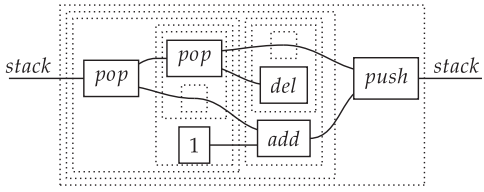
while a term of type $d : v \to \epsilon$, sometimes called an *effect*, has an empty right boundary,

Consequently, a term of type $d : \epsilon \to \epsilon$, 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:

The names 'state', 'effect', and 'scalar' originate from the role played by string diagrams of this type in quantum theory [35].

From Terms to String Diagrams

Terms are not quite the same as string diagrams. As soon as we consider more elaborate terms, we realise that the preceding definition requires us to decorate pictures with extra notation to keep track of the order in which we have applied the different forms of composition. For instance, (2.2) is a term from the signature in Example 2.1.



(2.2)

It denotes a very simple protocol, which pops two values of the stack, deletes the second one, and increments the first by one, before pushing it back onto the stack. We have only kept outer object labels for readability. Its full derivation tree is given in Figure 2.1. Notice that a bracketing by dotted frames fully specifies the corresponding derivation tree.

This example makes apparent that Σ-terms come with lots of extra information on how the graphical representation has been put together: the dotted boxes keep track of the order in which sequential and parallel composition have been

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

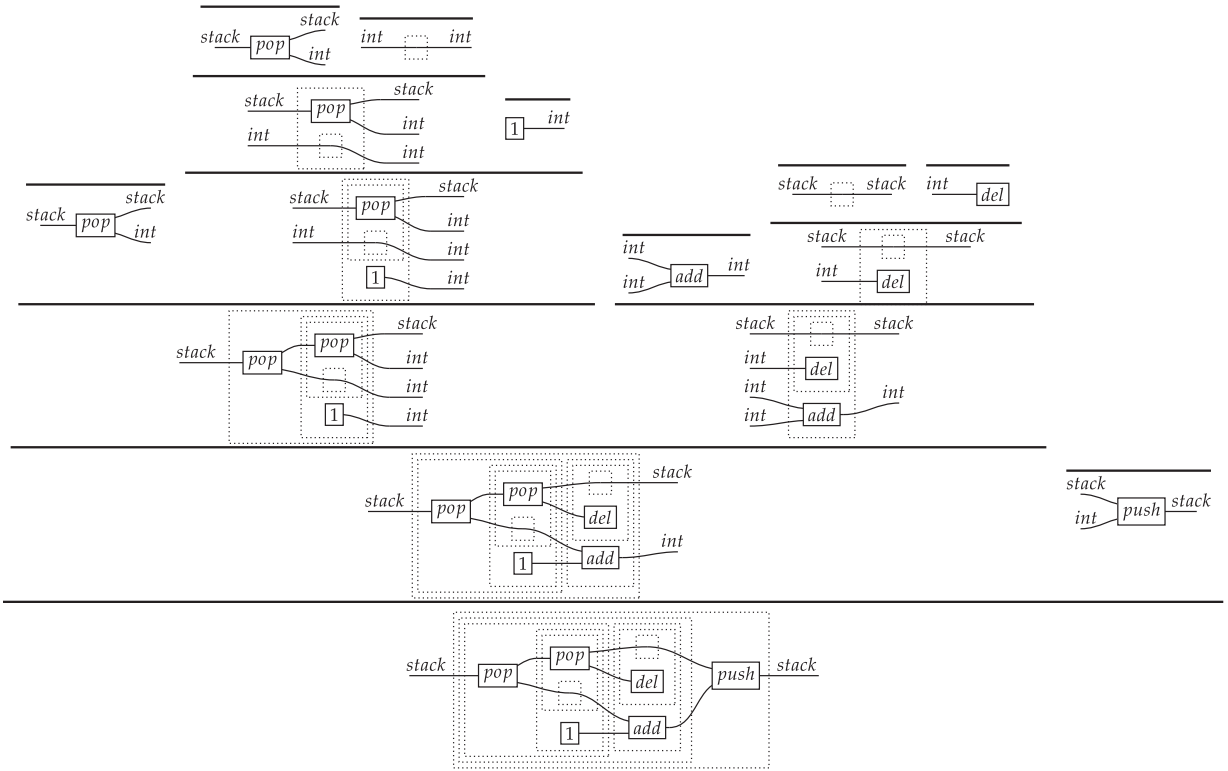
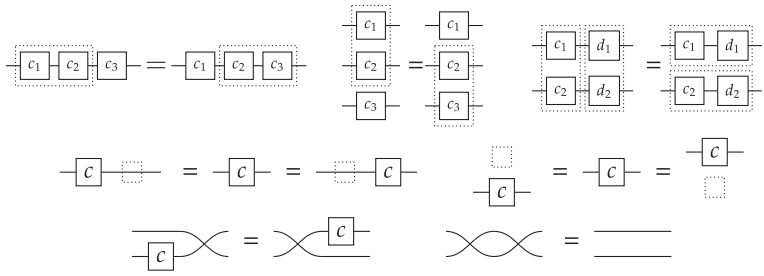


Figure 2.1 An example derivation tree.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

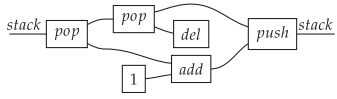
applied. The move to string diagrams allows us to *abstract away* this information and focus solely on how the term components are wired together. More formally, a string diagram on Σ is defined as an equivalence class of Σ-terms, where the quotient is taken with respect to the reflexive, symmetric, 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/co-arity):



(2.3)

If we think of the dotted frames as two-dimensional brackets, these laws tell us 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, justifying the use of the unbracketed expression $a \cdot b \cdot c$. 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 associativity.[1] The rightmost identity of the first line in (2.3) is the interchange law, which concerns the interplay between the two forms of composition: we can take the parallel composition of two sequentially composed terms, or vice versa, and the resulting string diagram will be the same. The remaining axioms of the first two lines encode the associativity and unitality of the two forms of composition. The last line contains two axioms involving wire crossings $\times$: the first, called the *naturality* of $\times$, tells us that boxes can be pulled across wires; the second, that the wire crossing is self-inverse. As a result, wires can be entangled or disentangled as long as we do not modify how the boxes are connected. We call these axioms the laws of symmetric monoidal categories (SMC).

Thanks to the laws of SMC, we can safely remove the brackets from the term in (2.2) to obtain the corresponding string diagram:

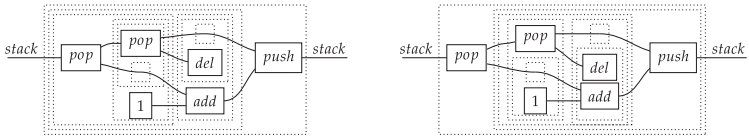


(2.4)

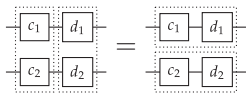
[1] In fact, there is a sense in which this is precisely true; see Section 6.2.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

This representation is now unambiguous because the axioms in (2.3) imply that any way of placing dotted frames around components of (2.4) leads to equivalent Σ-terms. For instance, the following two bracketed terms are equivalent as string diagrams:

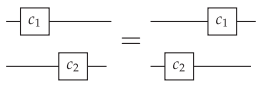


There is an important subtlety: if, formally, a string diagram is an equivalence class of terms quotiented by the laws of (2.3), there is not a unique way to depict a string diagram. In other words, the graphical representation (even without dotted frames) sits in between terms and string diagrams, as it does not distinguish certain equivalent terms. In some cases the depiction absorbs the laws of SMC, for example, for the two sides of the interchange law:



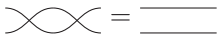
(2.5)

In other cases, *the way we draw* them distinguishes string diagrams that are equivalent under the laws of (2.3). Consider, for example,



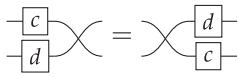
This equality can be seen as an instance of the interchange law (2.5) with identity wires or as a consequence of the unitality of identity wires, which allows us to stretch wires as much as we like.

A related point is that string diagrams do not distinguish different ways of braiding wires, even if our drawings do:

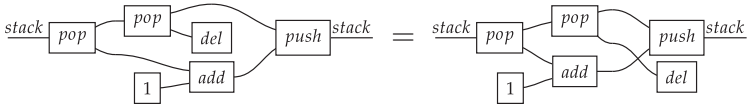


The laws of (2.3) guarantee that any two string diagrams made entirely of wire crossings over the same number of wires are equal when they define the same *permutation* of the wires. If the other rules are two-dimensional versions of associativity, the wire-crossing axioms are two-dimensional generalisations 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 boxes across wires, which is the next best thing:

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

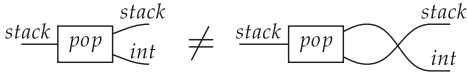


(We invite the reader to show this as their first exercise in diagrammatic reasoning.) Furthermore, we need to keep track of how boxes are wired, *but* only the specific permutation of the wires matters, not how we have constructed it. Coming back to our stack-machine example, the following are equivalent string diagrams:



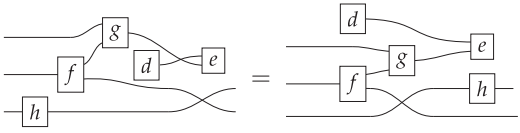
While this situation may appear slightly confusing at first, these examples show that in practice the distinction between string diagrams (as equivalence classes) and how we depict them is harmless. The topological moves that are captured by the equations of (2.3) are designed to be intuitive. They are 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 string diagrams.

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



Definition 2.2 (String diagrams over Σ)**.** *String diagrams over* Σ *are* Σ*-terms quotiented by the equations in* (2.3).

Example 2.3. Following the preceding discussion, the reader should convince themselves that the following two (unframed) terms depict the same string diagram:



(Free) Symmetric Monoidal Categories

In algebra, the collection of terms obtained from a signature, without any additional operations or equations, is often called the *free* structure over that

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

signature. The diagrammatic language of string diagrams comes with an associated notion of free structure: the free *symmetric monoidal category* (SMC) over a given signature.[2]

At this point, the more mathematically inclined reader might object that we still have not defined rigorously what an SMC is. Somewhat circularly, we could say that an 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 shown in (2.3). To state them without string diagrams, we need to introduce explicit notation for composition and the monoidal product. We do so in the following definition. Note that we assume basic knowledge of what a category is. The definition, along with related notions, can be found in the Appendix.

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

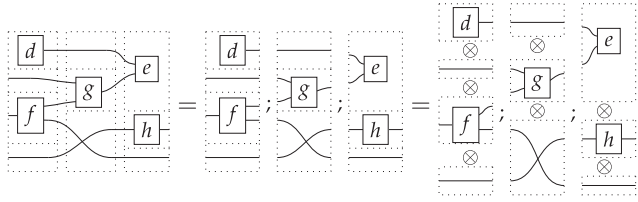
$$c_1 \otimes (c_2 \otimes c_3) = (c_1 \otimes c_2) \otimes c_3 \qquad id_I \otimes c = c = c \otimes id_I$$
$$(c_1 \otimes c_2) \,;\, (d_1 \otimes d_2) = (c_1 \,;\, d_1) \otimes (c_2 \,;\, d_2)$$

and a family of morphisms σ_X^Y *for any two objects* X, Y*, such that*

$$(id_X \otimes c) \,;\, \sigma_X^Z = \sigma_X^Y \,;\, (c \otimes id_X) \quad \textit{for any } c \colon Y \to Z$$
$$\sigma_X^Y \,;\, \sigma_Y^X = id_X \otimes id_Y$$

Observe that these are exactly the laws shown in (2.3) in symbolic form: we can simply replace '$\otimes$' by vertical composition and '$\,;\,$' by horizontal composition.

It is possible to translate any string diagrams into symbolic notation. 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:



[2] The notion of a free category generalises the same construction in algebra. It can be understood in terms of an adjunction, as explained, for instance, in [4, appendix A.2]. For the sake of our exposition, Definition 2.6 suffices.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

As for string diagrams, there are multiple ways to write a given morphism in symbolic notation. In fact, because string diagrams absorb some of the laws of SMCs into the notation, 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 term 'symmetric monoidal category'. 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 and refer the reader to any standard textbook on category theory for a general introduction to SMCs [89, chapter XI]. Our approach is nevertheless theoretically motivated by the following fundamental result: every SMC is equivalent (in a sense that we will not cover here in detail, but do recall in the Appendix, at Definition A.7) to a strict SMC. This fact is what allows us to draw string diagrams. It is known as the* coherence theorem *for SMCs. 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 application.*[3] *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 we can simply concatenate elements of a monoid and omit the symbol for the multiplication and the parentheses to bracket its application, we can use string diagrams to represent morphisms of a SMC. It is then natural that more composition operations require more dimensions to represent. In fact, some of the earliest appearances of string diagrams*[4] *occurred to construct free SMCs with additional structure and prove a coherence theorem for them [79, 80].*

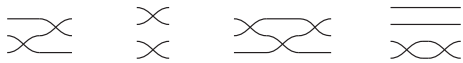
Definition 2.6 (Free SMC on a signature Σ)**.** *The symmetric monoidal category* $\mathsf{Free}_{SMC}(\Sigma)$ *is formed by letting objects be elements of* Σ_0^* *and morphisms be string diagrams over* Σ, that is, Σ*-terms quotiented by* (2.3)*. The monoidal product is defined as word concatenation 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.*

[3] The coherence theorem is due to Mac Lane [89]. A recent exposition based on string diagrams can be found in [111].

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

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

Example 2.7 (Free SMC over a single object). The free SMC over the signature $\Sigma = (\{\bullet\}, \varnothing)$ is easy to describe explicitly. Its string diagrams are generated by 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 string diagram $\bullet^n \to \bullet^n$ is a permutation of n elements, and there are no diagrams $\bullet^n \to \bullet^m$ for $n \neq m$.

Free SMCs on a single generating object (and arbitrary generating operations) are usually called PROPs (**Pro**duct and **P**ermutation categories) [88]. 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 a (strict) 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 plain PROPs.

When we encounter PROPs, we will use natural numbers as objects, since all objects are of the form $\bullet^n$, and write the type of a string diagram $\bullet^n \to \bullet^m$ simply as $n \to m$.

Symmetric Monoidal Functors

Whenever we define a new mathematical structure, it is 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 a semantic interpretation in Section 5.

Definition 2.8. *Let* $(\mathsf{C}, \otimes, I, \sigma)$ *and* $(\mathsf{D}, \boxtimes, J, \theta)$ *be two SMCs. A (strict)* symmetric monoidal functor $F : \mathsf{C} \to \mathsf{D}$ *is a mapping from objects of* C *to those of* D *that satisfies*

$$F(X_1 \otimes X_2) = F(X_1) \boxtimes F(X_2) \qquad \textit{and} \qquad F(I) = J$$

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

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

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

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

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 [89] for a standard treatment and [106] for the connections with string diagrams.

If we have two such functors $F\colon \mathsf{C} \to \mathsf{D}$ and $G\colon \mathsf{D} \to \mathsf{C}$ that are inverses to each other – FG and GF are identity functors – we say that the two SMCs are *isomorphic*. We will also use the notion *equivalence* of SMCs. This is a more relaxed notion than that of isomorphism, where FG and GF are merely isomorphic to identity functors. It is more appropriate in some cases, in particular when the categories involved are not strict monoidal (see Remark 2.5, for example). We will not dwell on equivalences of categories much in this Element, but refer the reader to Definition A.7 and Remark A.8 in the Appendix.

Remark 2.9 (On functors from free SMCs)**.** *When defining a functor F out of a free SMC $\mathsf{Free}_{SMC}(\Sigma)$, there is a clear recipe to follow: we only need to specify to which object we want to map elements of Σ_0, and to which morphism $Fd\colon Fu \to Fv$ we want to map each element of $d\colon u \to v$ of Σ_1. This is because of the universal property of free constructions: if we have a mapping from the set of generating operations of some signature Σ to morphisms of some SMC C, there is a unique way of extending this mapping to a symmetric monoidal functor $\mathsf{Free}_{SMC}(\Sigma) \to \mathsf{C}$. This observation will come in handy when defining the semantics of string diagrammatic theories, in Section 5.*

Example 2.10. In Example 2.7, 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 $n \to n$ are permutations of $n = \{0, \ldots, n-1\}$. We can equip it with a monoidal product, given by addition on objects, and on morphisms $\theta_1 : n_1 \to n_1$ and $\theta_2 : n_2 \to n_2$ 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 $\sigma : 2 \to 2$, given by $\sigma(0) = 1$ and $\sigma(1) = 0$. The isomorphism is straightforward. In one direction, let $F\colon \mathsf{Free}_{SMC}(\{\bullet\}, \varnothing) \to \mathsf{Bij}$ be given by $F(\bullet^n) = n$ on objects and $F(\times) = \sigma$. This is enough to describe F fully because all string diagrams of $\mathsf{Free}_{SMC}(\{\bullet\}, \varnothing)$ are vertical or horizontal composites of $\times$ and F has to preserve these two forms of composition, by Definition 2.8. Furthermore, the required properties are immediately satisfied. To build its inverse, we need to know that we can factor any permutation into a

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

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 string diagram that is the identity everywhere and $\times$ at the i-th and $i+1$-th wires. Call this diagram at_i. Then, let $G\colon \mathsf{Bij} \to \mathsf{Free}_{SMC}(\{\bullet\}, \varnothing)$ be given by $G(n) = \bullet^n$ on objects and on morphisms by $G(\theta) = at_{i_1} at_{i_2} \ldots at_{i_k}$ where $(i_1\, i_1+1)(i_2\, i_2+1)\ldots(i_k\, i_k+1) = \theta$ is a decomposition of θ into adjacent transpositions. One can check that this is well-defined and satisfies all the equations of Definition 2.8. 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 $GF(\times) = G(1\ 2) = \times$. The other direction is a bit more lengthy, but without any major difficulties.

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

In fact, this SMC is also equivalent 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.

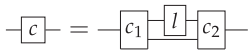
This example is just a taster of an idea that we will develop further in Section 5, dedicated to the *semantics* – that is, the interpretation – of string diagrams.

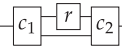
2.1 Adding Equations

The equations shown in (2.3) only capture a very basic notion of equivalence between string diagrams. When describing computational processes for example, it is useful to include more equations, specific to the domain of interest. In string diagram theory, these additional equations are encapsulated in the notion of *symmetric monoidal theory*. More formally, a symmetric monoidal theory – or simply *theory* when no ambiguity can arise – is a pair (Σ, E), where Σ is a signature and E is a set of equalities $l = r$ between string diagrams of the same type over Σ. We write $\overset{E}{=}$ for the smallest congruence relation (w.r.t. sequential and parallel composition) containing E. We will see many examples of symmetric monoidal theories in Section 2.2.

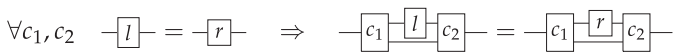
Remark 2.11 (Equations and diagrammatic rewriting.)**.** *It might be helpful to see equations as two-way rewriting rules that can be applied in an arbitrary context. More precisely, assume that we have some equation of the form $l = r$, where l, r have the same type; to apply it in context, we need to identify l in a larger string diagram c, that is, find c_1 and c_2 such that*

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

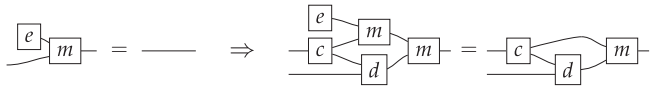


and simply replace l by r, forming the new diagram 

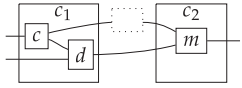
. This is just the diagrammatic version of standard algebraic reasoning. We can summarise this process as follows:



For example,



where the context is



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

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 determines a free structure: given a signature Σ and a theory E, we can form the free SMC $\mathsf{Free}_{SMC}(\Sigma, E)$ obtained by quotienting the free SMC $\mathsf{Free}_{SMC}(\Sigma)$ by the equivalence relation over string diagrams given by $\overset{E}{=}$.

Definition 2.12 (Free SMC over a theory (Σ, E))**.** *The symmetric monoidal category* $\mathsf{Free}_{SMC}(\Sigma, E)$ *is formed by letting objects be elements of* $\Sigma_0^\star$ *and morphisms be equivalence classes of string diagrams over* Σ *quotiented by* $\overset{E}{=}$*. 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

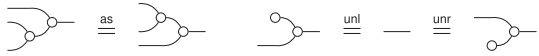
Some theories occur frequently in the literature. Many authors assume familiarity with the axioms hiding behind the words ‘monoids’, ‘comonoids’, ‘bimonoids/bialgebras’, or ‘Frobenius monoid/algebra’, and how all of these theories relate to one another. For this reason it is valuable to know them well,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

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

Example 2.13 (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:

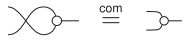
$$m(m(x,y),z) = m(x,m(y,z)) \quad \text{and} \quad m(u,x) = x = m(x,u)$$

Analogously, the symmetric monoidal theory of monoids can be presented by a signature $\Sigma = (\{\bullet\}, \{\supset\!\!\circ\!\!-, \circ\!\!-\})$, based on a single object-type $\bullet$, a *multiplication* $\supset\!\!\circ\!\!- : 2 \to 1$, and a *unit* $\circ\!\!- : 0 \to 1$, and three axioms, for associativity and (two-sided) unitality:

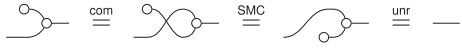


Observe that we have just replaced variables with wires and algebraic operations with diagrammatic generators. As in ordinary algebra, two terms/diagrams are equal if one can be obtained from the other by applying some sequence of these three equations (recall Remark 2.11).

We can also present commutative monoids in the same way. Recall that commutative monoids are those that satisfy $m(x,y) = m(y,x)$; diagrammatically, we can present the corresponding symmetric monoidal theory with the same signature and a single additional equality (note the use of the symmetry $\times$):



Of course, in the presence of commutativity, each of the unitality laws are derivable from the other. In the usual algebraic theory of monoids we would show this as follows: if $m(x,y) = m(y,x)$, then $m(u,x) = m(x,u) = x$ where the last step is right-unitality. The corresponding diagrammatic proof is very similar, with one additional step:

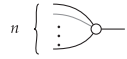


The second equality is a simple instance of the bottom left axiom of (2.3), for a string diagram with no wires on the left (that is, of type $\epsilon \to w$ for some w).

This makes an important point: two theories (Σ, E) and (Σ', E') might present the same structure, in the sense that the corresponding free SMCs $\mathsf{Free}_{SMC}(\Sigma, E)$ and $\mathsf{Free}_{SMC}(\Sigma', E')$ might be isomorphic. It is also a good place to mention that theories do not have to be minimal in any way; they can contain axioms that are derivable from the others. There are various reasons one might prefer a theory that contains redundant axioms: to highlight some of the symmetries, to avoid having to re-derive some equalities as a lemma later on, and others.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

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, for example $m(a,b,c)$, 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 $: 2 \to 1$: we can define a generalised n-ary operation as a dot with n-many wires



as syntactic sugar to denote multiple applications of . 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.

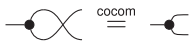
Example 2.14 (Comonoids). Unlike algebraic syntax, string diagrams allow for operations with co-arity different from 1, manifested by multiple (or no) right boundary wires. It is therefore possible to flip the generators and axioms of the theory of monoids to obtain the symmetric monoidal theory of *comonoids*! Unsurprisingly, it is represented by a signature with a single object (which we therefore omit in diagrams), two generators, called comultiplication and counit,

and the following three axioms:

called coassociativity and counitality. As one can see immediately, string 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 syntactic sugar for a generalised comultiplication node with co-arity n for any natural number:

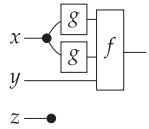
Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

A comonoid is furthemore *cocommutative* if



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 string 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:

$$x, y, z \vdash f(g(x), g(x), y) \qquad \mapsto$$



For this reason, from the diagrammatic perspective, algebraic theories (or Lawvere theories, their categorical cousins, see [74]) always carry a chosen cocommutative 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 5.2.

Remark 2.15 (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. However, the analogy between symmetric monoidal theories and algebraic theories hides an important subtlety: 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 restriction – termed* resource-sensitivity *– is an important feature of diagrammatic syntax. Properties that do not involve multiple uses of variables can be specified completely analogously, as we saw for commutativity. Axioms that use each variable precisely once on each side of the equality sign are called* linear *axioms. Non-linear axioms cannot be translated directly in the diagrammatic context, however. For example, it makes no sense to refer to the symmetric monoidal theory of idempotent monoids: those monoids that satisfy $m(x,x) = x$. Indeed, to even state the idempotency axiom one requires the ability to duplicate wires. As we will see, idempotency can also be expressed diagrammatically, but as a property of a more complex algebraic structure than a monoid; it can be stated as a property of a* bimonoid*, which is*

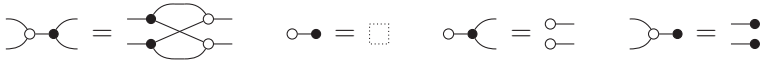
Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

*our next example. This example is an instance of a more general pattern that allows us recover the resource-*in*sensitivity of ordinary algebraic syntax. We will explore this correspondence more systematically in Section 4.2.5.*

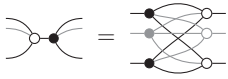
The respective theories of monoids and comonoids can interact in different ways, as the next two examples illustrate. By 'interact' in this context, we mean that there are different equations that one can impose when considering a signature that contains both the generators of monoids and those of comonoids with their respective theories.

Example 2.16 (Co/commutative bimonoids). One possible theory axiomatises a structure called a *bimonoid*. It is presented by the generators of monoids and comonoids

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



Intuitively, these equalities can be seen as instances of the same general principle: whenever one of the monoid generators is composed horizontally 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):



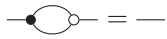
(2.6)

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 2.14, 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. For example, a bimonoid is *idempotent* when it satisfies

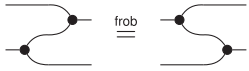
Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

the following additional equality, which clearly translates the usual $m(x,x) = x$ into a diagrammatic axiom:



Example 2.17 (Frobenius monoids). Bimonoids are not the only way that monoids and comonoids can interact – there is another structure that frequently appears in the literature, under the name of *Frobenius monoid*, or *Frobenius algebra*.[5] This structure is presented by the generators of monoids and comonoids. We will write them using nodes of the same colour, as this is how Frobenius monoids tend to appear in the literature, and this will allow us to distinguish them from bimonoids in the rest of the Element:

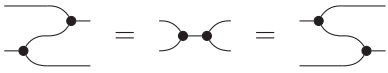
together with their respective axioms, and the following additional axiom, called Frobenius' law:



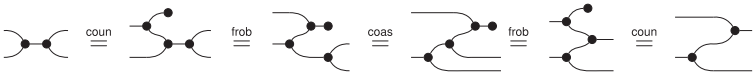
(2.7)

This equality provides 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. We will examine this incompatibility more closely in Section 4.2.10.

The reader might encounter other versions of this axiom in the literature, such as:



In the presence of the other axioms (namely counitality and coassociativity), these two equalities are derivable from (2.7). To get a feel for diagrammatic reasoning, let us prove it:



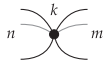
[5] The term 'Frobenius monoid' is due to monoids being traditionally more familiar than comonoids, even though both structures play an equally prominent role in a Frobenius monoid. As for the provenance of 'Frobenius algebra', the term 'algebra' usually refers to a monoid which is also a vector space (and whose multiplication is a linear map). This is the context in which Frobenius monoids were first studied.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

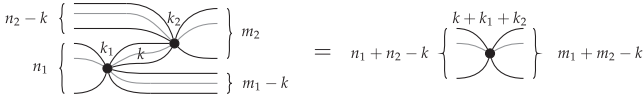
At first, the string diagram novice may find it difficult to internalise all the laws that make up a theory such as that of Frobenius monoids. When proving some equality, it is not always clear which axiom to apply at which point to reach the desired goal and it is easy to get overwhelmed by all the 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 at a glance when any two string diagrams for this theory are equal. To explain this, it is helpful to think of string diagrams for the theory of Frobenius monoids as (undirected) graphs, whose vertices are any of the black dots, and edges are wires. We say that a string diagram is *connected* if there is a path between any two vertices in the corresponding graph. It turns out that, for Frobenius monoids, any connected string diagram composed out of (finitely many) , , , or using vertical or horizontal composition (without wire crossings) is equal to one of the following form [73, section 5.2.1]:

where we use an ellipsis to represent an arbitrarily large composite following the same pattern. In other words, the only relevant structure for a connected string diagram in the theory of Frobenius monoids 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 just depicted). This observation is sometimes called the *spider theorem* and justifies introducing generalised vertices we call spiders as syntactic sugar:



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

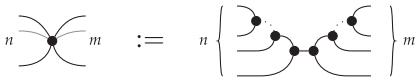


Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

where k is the number of middle wires that connect the two spiders on the left-hand side of the equality. As a result, we need only keep track of the number of open wires and loops for any complicated string diagram; this greatly reduces the mental load of reasoning about this theory.

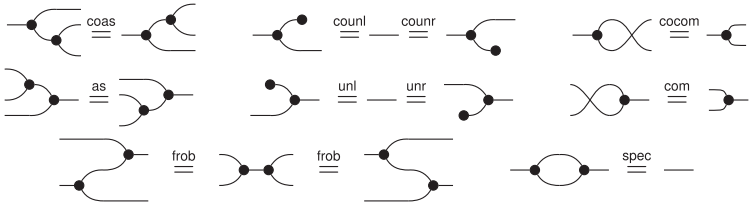
Frobenius monoids that satisfy the following idempotency axiom occur frequently in the literature:

In this case, the Frobenius monoid is called *special* (or sometimes, *separable*). The normal form given by the spider theorem simplifies even further in this case, since we can now forget about the inner loops:



The only relevant structure of any connected string diagram in the theory of special Frobenius monoids 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. 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.

Example 2.18 (Special and commutative Frobenius monoids). The commutative and special Frobenius monoids are very common in the literature, as they are an algebraic structure one finds naturally when reasoning about relations (as we will see when we study the semantics of string diagrams in Section 5). We summarise here the full equational theory for future reference:



(2.8)

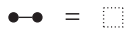
In what follows, we will refer to this theory as scFrob.

When adding commutativity in the picture, the spider theorem still holds and includes string diagrams composed out of (finitely many) *and wire crossings*, using vertical or horizontal composition. Any string diagram in the free SMC over the theory of commutative and special Frobenius monoids is fully determined by a list of spiders, and to where each of their respective legs are connected on the left and on the right boundary. In other words, string

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

diagrams with n left wires and m right wires are in one-to-one correspondence with maps $n + m \to k$ for some k. This result will be the basis of a concrete description of the free SMC on the theory of a special commutative Frobenius monoid in Example 5.11.

Example 2.19. A special Frobenius monoid that moreover satisfies the following axiom is sometimes called *extra-special*:



(2.9)

This means that we can forget about networks of black nodes without any dangling wires – they can always be eliminated. In this case, string diagrams with n left wires and m right wires in the (free SMC over the) theory of a commutative extra-special Frobenius monoid are in one-to-one correspondence with partitions of $\{1, \ldots, n + m\}$. Intriguingly, one may think of (2.9) as a 'garbage collector'; in the relational interpretation of string diagrams, it allows us to capture *equivalence relations*, as it eliminates empty equivalence classes. We will come back to the case of extra-special commutative Frobenius monoids in Example 5.12.

The reader will frequently encounter the theories we covered here as the building blocks of more complex diagrammatic calculi, designed to capture different kinds of phenomena. For example, the ZX-calculus, a theory that generalises quantum circuits (see Section 7), contains not one, but two special commutative Frobenius monoids, often denoted by a red and a green dot respectively. They interact together to form two bimonoids: the green monoid with the red comonoid forms a bimonoid, and so does the red monoid with the green comonoid. At first, this seems like a lot of structure to absorb, but quickly, one learns to use the spider theorem to think of monochromatic string diagrams, so that most of the complexity comes from the interaction of the two colours. And even then, the generalised bimonoid law we saw in Example 2.16 helps a lot. In fact, modern presentations of the ZX-calculus prefer to give the theory using spiders of arbitrary arity and co-arity as operations 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 see several example applications in Section 7.

Remark 2.20 (Distributive Laws)**.** *It is noteworthy that both the equations of bimonoids (Example 2.16) and of Frobenius monoids (Example 2.17) describe*

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

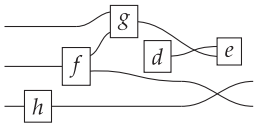
the interaction of a monoid and a comonoid, even though they do it in different ways. A way to put it is in terms of factorisations: the bimonoid laws allow us to factorise any string diagram as one where all the comonoid generators precede the monoid generators, as in (2.6)*; conversely, the Frobenius laws yield a monoid-followed-by-comonoid factorisation, as in the spider theorem. More abstractly, the two equational theories can be described as different specifications of a* distributive law *involving the monoid and the comonoid. Distributive laws are a familiar concept in 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 symmetric monoidal theories, distributive laws are even more powerful, as they can be used to study the interaction of theories with generators with arbitrary co-arity, such as comonoids, Frobenius monoids, and so on. The systematic study of distributive laws of symmetric monoidal theories has been initiated by Lack [84] and expanded in more recent works [22, 23, 116]. Understanding a theory as the result of a distributive law allows us to obtain a factorisation theorem for its string diagrams, such as* (2.6) *and the spider theorem. 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 [116] for a more systematic introduction to distributive laws of symmetric monoidal theories, as well as other ways of combining together 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 string diagrams may be also treated as *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, such as whether there is a path between two components. Another important reason to explore a combinatorial perspective to string diagrams is that their graph representation 'absorbs' the laws of symmetric monoidal categories shown in (2.3). It is thus more adapted than the syntactic representation for certain computation tasks, such as rewriting (see Section 6.1). The goal of this section is to illustrate how string diagrams can be formally interpreted as graphs.

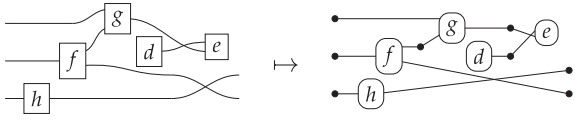
As a starting point, let us take, for example, a string diagram we have previously considered:

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715



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 string diagrams as a mathematical notation; see, for example, [77]. An immediate challenge for this 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: $\overset{x}{\text{—}}$. Historically, a solution to this problem has been to consider as interpretation a more sophisticated notion of graph, endowed with a topology from which one can define when edges are 'loose', 'half-loose', 'pinned', and so on; see [77]. Another, more recent approach understands string diagrams as graphs with two sorts of nodes, where the second sort just plays the bureaucratic role of giving an end to edges that otherwise would be drawn as loose [46].

The approach we present follows [13]. We do 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 and nodes: these interpret the boxes and the loose ends of wires in a string diagram, respectively. And the wires themselves? They are simply a depiction of how hyperedges connect with the associated nodes. Such an interpretation applies as follows to our leading example:

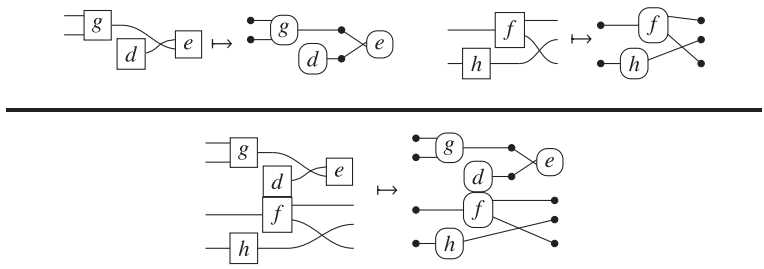


Note that, even though they are seemingly very close in shape, the two entities just displayed are of a very different nature. The one on the left is a syntactic object: the string diagram representing some term modulo the laws of SMCs. The one on the right is a combinatorial object: a hypergraph, with nodes indicated as dots and hyperedges indicated as boxes with round corners, labelled with Σ-operations.[6]

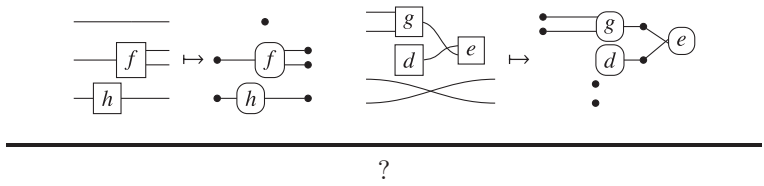
[6] The reader may wonder what happens when there is more than one generating object, so that string diagrams have non-trivial labels on wires. All of this section generalises to that more general setting: in the hypergraph interpretation, nodes may be labelled with the appropriate object, and constructions that merge nodes are disallowed unless the label matches. As this

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

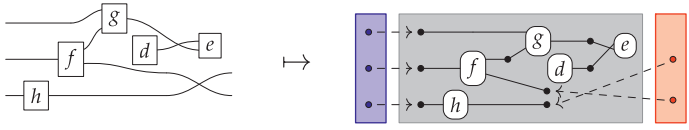
In order to turn this mapping into a formal interpretation, we need an understanding of how to handle composition of string diagrams. Intuitively, parallel composition is simple: if we stack one hypergraph over the other, we still obtain a valid hypergraph. For instance:



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



A proper definition of this composition operation is what leads to the notion of an *open* hypergraph: a hypergraph with a record of what nodes form its *left* interface and what nodes form its *right* interface. Note that one node can be in both, as in (3.1). Pictorially, we will display the interfaces as separate discrete hypergraphs,[7] one on the left and one on the right, with dotted lines indicating which nodes of the actual hypergraph lie on which interface. Our leading example corresponds to the following open hypergraph on the right.



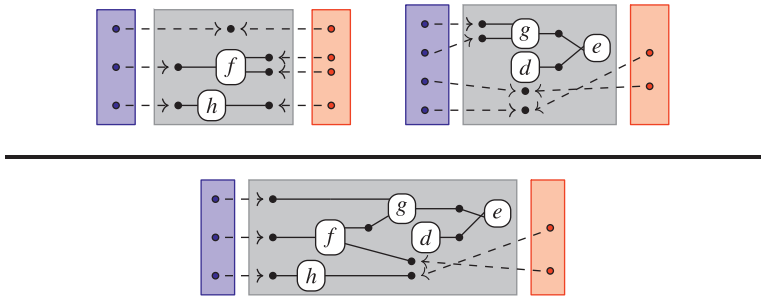
Thanks to this additional information, the open hypergraph comes endowed with a built-in notion of sequential composition, mimicking the sequential composition of string diagrams. We are allowed to compose two open

generalisation poses no significant conceptual difficulty, for the sake of clarity we opted for focussing our exposition on the case of theories with one generating object.

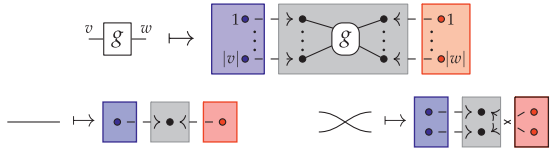
[7] A hypergraph is discrete when it has just nodes and no hyperedge.

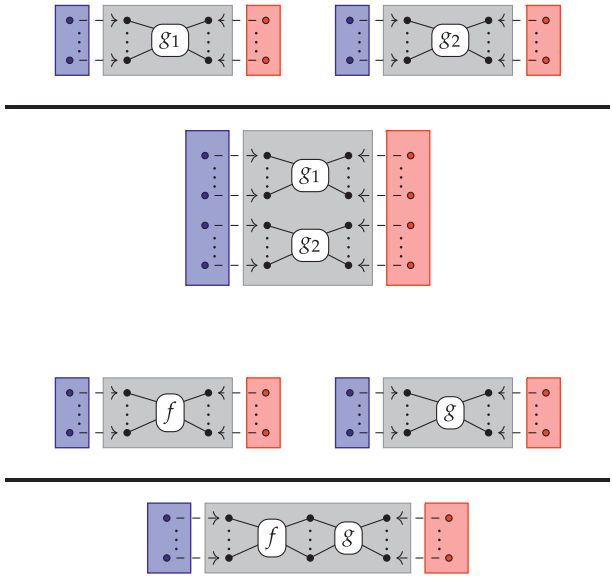
Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

hypergraphs sequentially whenever the right interface of the first coincides with the left interface of the second.



Equipped with this notion, we may define the interpretation of string diagrams as open hypergraphs, inductively on Σ-terms: for each $g : v \to w$ in Σ_1, we have



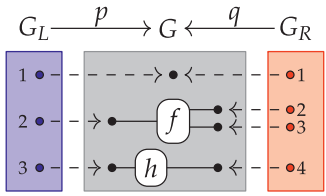


In words, the vertical composition takes the disjoint sum of each of the interfaces and hypergraphs, while the horizontal composition identifies the middle interface labels and includes them as nodes into the composite hypergraph.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

Note that 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 complicate the notion of graph at hand, for instance by adding a different sort of nodes. From a mathematical viewpoint, an open hypergraph G may be simply expressed as a structure consisting of two hypergraph homomorphisms $\langle p : G_L \to G, q : G_R \to G \rangle$, where G_L and G_R are discrete hypergraphs, and the image $p[G_L]$ (resp. $q[G_R]$) identifies the nodes in the left (resp. right) interface of G. The visualisation in (3.1) shows such encoding: G_L is the hypergraph on the left side, G_R is the one on the right side, and G is the one in the middle. The dotted lines, identifying the interfaces of G, now take formal meaning as the definition of functions $p : G_L \to G$ and $q : G_R \to G$.



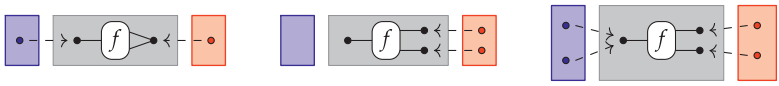
(3.1)

The structure $\langle p : G_L \to G, q : G_R \to G \rangle$ 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 with one such cospan structure. Reasoning with cospans is 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 earlier 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 (3.1) defines a monoidal functor $[\![\cdot]\!]$ from the free SMC over signature Σ to the SMC of cospans of hypergraphs.

An important question stemming from the interpretation (3.1) is: to what extent are the syntactic and the combinatorial perspectives on string diagrams interchangeable? First, one may show that $[\![\cdot]\!]$ 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.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715



These examples have something in common: nodes are allowed to behave more freely than in the image of interpretation of (3.1). 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; 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 $[\![\cdot]\!]$. 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.*

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.*

Moreover, any hypergraph that corresponds to a string diagram is *acyclic*: there are no directed paths containing the same node twice. These two properties are enough to characterise string diagrams.

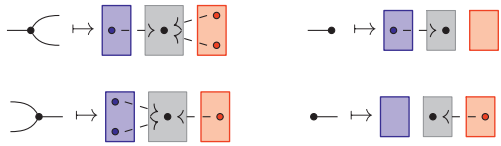
Theorem 3.3. *An open hypergraph is in the image of $[\![\cdot]\!]$ if and only if it is monogamous and acyclic.*

Corollary 3.4. *String diagrams over Σ are in one-to-one correspondence with Σ-labelled monogamous and acyclic open hypergraphs.*

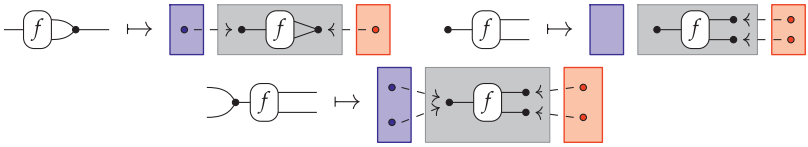
Theorem 3.3 settles the question of what kind of open hypergraphs correspond to 'syntactically generated' string diagrams. We may also ask the converse question: what do we need to add to the algebraic specification of string diagrams in order to capture all the open hypergraphs?

Remarkably, the special and commutative *Frobenius monoid* from Example 2.18 is tailored to the role. Indeed we can give an interpretation to the operations of Example 2.18, as discrete open hypergraphs:

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715



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



These observations generalise to the following result, thus completing the picture of the correspondence between string diagrams and open hypergraphs. In stating it, we write $\Sigma + \mathsf{scFrob}$ for the signature given by the disjoint union of the generators of Σ and of scFrob, the theory of a special commutative Frobenius monoid given in (2.8).

Theorem 3.5. *String diagrams on the signature* $\Sigma + \mathsf{scFrob}$ *modulo the axioms of special commutative Frobenius monoids given in* (2.8) *are in one-to-one correspondence with* Σ*-labelled open hypergraphs.*

Note that it is not just the signature: the axioms of special commutative Frobenius monoids also 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*-labelled nodes and* Σ_1*-labelled hyperedges form a symmetric monoidal category* Hyp_Σ*, whose morphisms are hypergraph homomorphisms respecting the labels. The monogamous and acyclic 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* $\mathsf{Free}_{SMC}(\Sigma)$ *and* MHyp_Σ*, and an isomorphism between* $\mathsf{Free}_{SMC}(\Sigma + \mathsf{scFrob})$ *and* Hyp_Σ*.*

4 Categories of String Diagrams

Manipulating string diagrams can be confusing to the newcomer because there are actually many flavours, each of which authorises or forbids different deformations and manipulations. To make matters worse, many papers

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

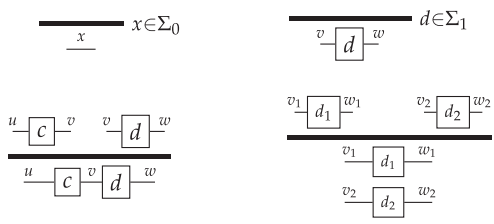
will assume that the reader is comfortable with the rules of the game for the authors' specific flavour, and gloss over the basic transformations. This is not necessarily a bad thing, as the point of string diagrams is to serve as a useful computational tool, a syntax that empowers its users by absorbing irrelevant details into the topology of the notation itself. 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 insight into 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 class of string diagrams under consideration. For the reader willing to delve further into this subject, we recommend Selinger's extensive survey of diagrammatic languages for monoidal categories [106].

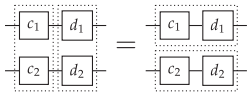
4.1 Fewer Structural Laws

4.1.1 Monoidal Categories What if we take away the ability to cross wires? Terms of the free monoidal category over a chosen signature $\Sigma = (\Sigma_0, \Sigma_1)$ are generated by the following derivation rules:



The difference with *symmetric* monoidal categories is that we no longer have the built-in symmetry components $\times$ at our disposal. We consider two terms structurally equivalent when they can be obtained from one another using the axioms of monoidal category, that is, the strict subset of those symmetric monoidal categories that do not involve the symmetry/wire-crossing, as found in (2.3). For example, we still have the interchange law

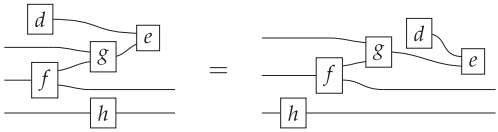
Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715



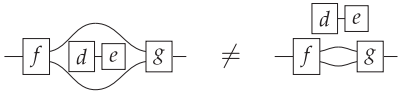
but we do not have

=

, as
is not even a term of our syntax. Intuitively, they are the planar cousins of their symmetric counterpart, that is, 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 simpler: we can only compose string 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. Then, two string diagrams are equivalent if one can be deformed into the other *without any intermediate 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 following diagrams:



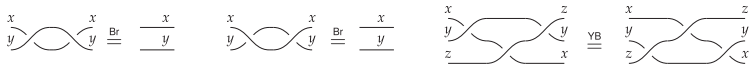
Thus, in the monoidal case, certain string diagrams can be trapped between some wires, without any way to move them on either side – whereas, in the symmetric monoidal case, we could have just pulled the middle diagram out, past the surrounding wires.

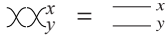
4.1.2 Braided Monoidal Categories Braided monoidal categories [76] are one step up from monoidal categories, but are not symmetric. They allow a form of wire crossing that keeps track of which wire goes over which. To this effect, we introduce a *braiding* for each of the two possibilities depicted suggestively 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:

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

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

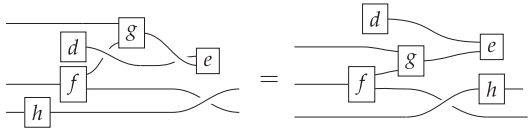


As the drawings suggest, the intuition for the braiding is that string diagrams now inhabit a three-dimensional space in which we are free to cross wires by moving them over or under each other. The first two laws state that the two braidings are inverses of each other, and the third is an instance of the naturality of the wire crossings, called the *Yang–Baxter* equation [29]. Notice that these axioms are similar to those for the symmetry in SMCs except that braidings are not self-inverse:  does not hold if we take instead (see what follows). As a result, we can draw any string 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 string 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 finer than that of symmetric monoidal categories.[8] A simple illustrative example of this phenomenon is the following twist:

If we replaced the two braidings by the symmetry to obtain a term of a symmetric monoidal category, this string diagram would simply be the identity . 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 string 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:



[8] From a formal viewpoint, this statement is not entirely accurate, because the terms of free braided and symmetric monoidal categories are different. However, we can map those of the former to the latter by sending the braid to the symmetry. Our statement then amounts to saying that this mapping is not injective, that is, that two different braided monoidal diagrams may be mapped to the same symmetric monoidal diagram.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

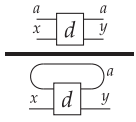
Remark 4.1. *Contrary to the case of SMCs (see Section 3), there is no known representation of string diagrams for braided monoidal categories as graphs.*

4.2 More Structural Laws

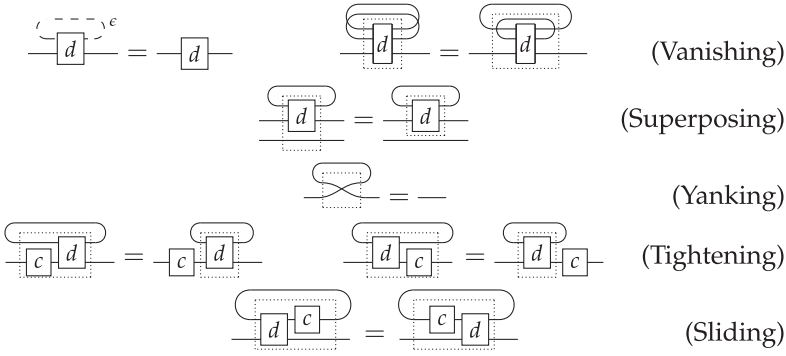
Just like we can weaken the structure of symmetric monoidal categories and draw more restricted diagrams, we can also extend our diagrammatic powers. The following is a non-exhaustive list of the most common variations one might find in the literature.

4.2.1 Traced Monoidal Categories String 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.

Term formation rules for traced monoidal categories are those of symmetric monoidal categories with the addition of an operation that allows us to form loops, called the *partial trace*:[9]



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

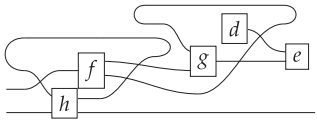


Here, we had to briefly go back to using dotted frames, because the axioms of traced monoidal categories are almost diagrammatic tautologies (which is the point of adopting a diagrammatic notation for them).

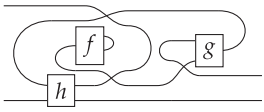
[9] The name comes from the usual linear algebraic notion of trace. We will examine this concrete case in Example 5.13.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

In short, string 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 with the same type*, as in the following example:

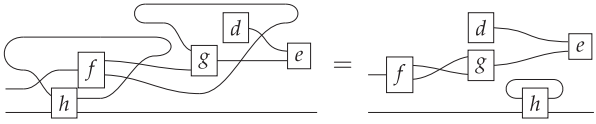


In essence, we have the ability to draw loops, breaking free from the acyclicity requirement of plain symmetric monoidal diagrams. However, we cannot bend wires arbitrarily, or connect right (*resp.* left) ports to right (*resp.* left) ports. For example, the following is *not* allowed:



(although we will soon introduce a syntax for which this kind of diagrams is allowed.)

The reader should convince themselves that the preceding string diagram is equivalent to the following one on the right:



As for symmetric monoidal diagrams, the trick to check this equality lies in verifying that the connectivity of the different boxes is preserved.

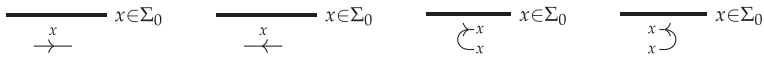
Remark 4.2. *Traced string diagrams are often used when describing computational processes that feature some form of recursion or iteration. In this context, it is also natural to consider trace-like operations that do not satisfy the yanking axiom. This makes sense when the trace-like operation is intended 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 [44, 45, 78].*

4.2.2 Compact Closed Categories Compact closed (or more simply, compact) 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

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

right and vice versa, using extra generators that represent wire-bending directly, as built-in operations.

The term formation rules are those of symmetric monoidal categories *except* the identity introduction rules, with the following additions:



In words, we introduce a new object x^* (called the dual of x) for every object x and write $\overset{x}{\rightarrow}$ for the identity on x and $\overset{x}{\leftarrow}$ for the identity on x^*. The objects x and x^* are related by two wire-bending diagrams $\overset{x}{\underset{x}{\circlearrowright}}$ and $\overset{x}{\underset{x}{\circlearrowleft}}$, called cup and cap respectively.

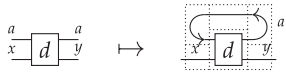
The corresponding notion of structural equivalence is defined by the axioms of symmetric monoidal category and the following two axioms, which capture the duality between inputs and outputs:

(4.1)

These are sometimes called the *snake* or *yanking* equalities.

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

From cups and caps, we also obtain a partial trace operation given simply by



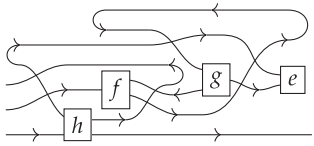
This operation satisfies all the required axioms of traced monoidal categories (it is an instructive exercise to prove them). Importantly, what was a global operation before is now decomposed into smaller components that use the added generators. This is particularly helpful in applications, whenever we aim at reasoning *compositionally* about feedback loops in a system. Whereas the notion of trace is 'native' to traced monoidal categories, it is a derived concept in compact closed categories.

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 flows 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.

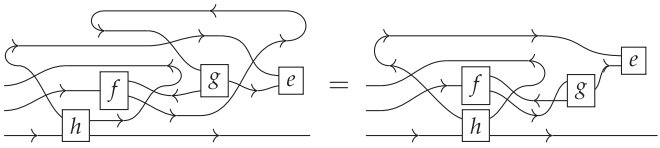
We can understand this as layering a notion of input and output on top of those of left and right ports. We can call inputs those wires that flow into a

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

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,* that is, we can assign a consistent direction to any wiring:



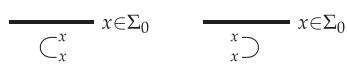
Furthermore, as it is now a leitmotiv, only the connectivity matters: we are allowed to straighten or bend wires at will, as long as we preserve the connections between the different sub-diagrams. For example, the following two diagrams are equivalent:



Finally, compact closed categories have the following very important property: diagrams of type $uv \to w$ are in one-to-one correspondence with diagrams $u \to wv^*$. Diagrammatically, moving v from the domain to the codomain is realised by bending the corresponding wire(s) using the cup $\overset{v}{\underset{v}{\circlearrowright}}$; moving in the other direction simply uses $\overset{v}{\underset{v}{\circlearrowleft}}$ to bend the v-wires back in place.

The fact that these operations are inverses to each other is an easy consequence of the snake equalities (4.1), with which we can straighten the wires back into place. As a result, wv^* can be seen as an internal analogue of the set of string diagrams $v \to w$. This property is found more generally in *closed monoidal* categories, which we cover in Section 4.2.9.

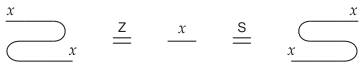
4.2.3 Self-Dual Compact Closed Categories There are significant instances of compact closed category where the distinction between inputs and outputs disappears completely: they are called *self-dual*. The term formation rules are the same as those symmetric monoidal categories, with the following additions:



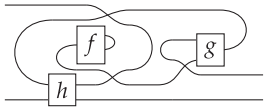
They are also very close to those of compact closed categories, but we identify x and its dual. As a result, there is no need to introduce a direction on wires. The

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

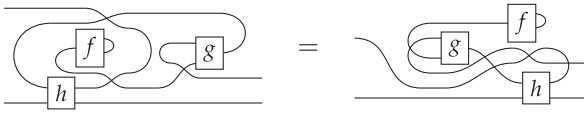
added constants $\subset_x^x$ and $\supset_x^x$ satisfy the same equations as their directed cousins:



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*:



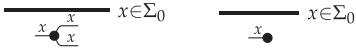
As before, the structural equivalence on diagrams allows us to identify any two diagrams where the same ports are connected:



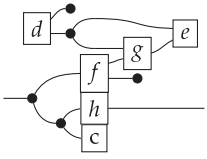
The previous three examples progressively relax which *two* ports we can connect together; in the following examples, we relax the requirement that only two ports can be connected at a time by introducing different ways to split and end wires.

4.2.4 Copy-Delete Monoidal Categories The first of these adds the ability to split and end wires in order to connect some wire in the right boundary of a diagram to a (possibly empty) *set* of wires in the left boundary of another. There are several names for these in the literature (Copy-Delete monoidal categories, gs-monoidal categories, etc.), but we will call them CD categories for short.

The term formation rules for CD categories are the same as those for symmetric monoidal categories, with the addition of wire splitting and ending for each generator:

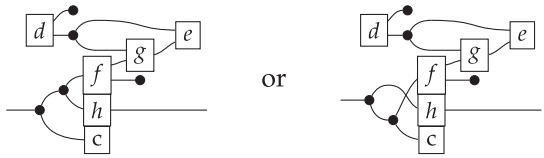


As anticipated, the copying and deleting operations allow us to connect a given right port of a diagram to a (possibly empty) *set* of left ports of another:

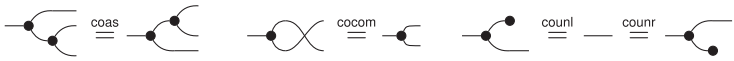


Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

Notice, however, that there are several ways of connecting the same set of wires. For example, we could have connected the left boundary wire to f, h, and c as follows:



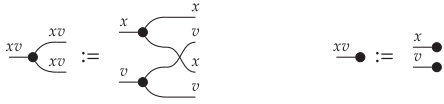
or any other way of connecting this wire to the same three boxes. To define a sensible notion of multi-wire connection, we need to add axioms that allow us to consider all these different ways of composing and equal if they connect the same wire to the same *set* of wires. To achieve this, and obtain a suitable notion of structural equivalence for CD categories, we add the following equalities to those of SMCs:



(4.2)

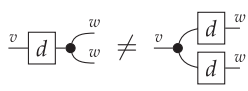
The reader will recognise these laws from Example 2.14 as those of a *commutative comonoid*: they tell us that there is only one way of splitting a single wire into n wires, for any natural n.

Using and for generating objects $x \in \Sigma_0$, we can define and for any word w over Σ_0 or, more plainly, for arbitrarily many wires. We do so by induction:



(4.3)

As mentioned in Example 2.14, it is typical in applications that and are interpreted as gates that *duplicate* and *discard* a resource. With this perspective, CD categories are categories whose structure makes duplicating and discarding of a resource explicit when it is used in some computation. This feature allows for a resource-sensitive analysis of processes. For instance, in CD categories we generally have that



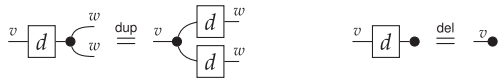
(4.4)

Intuitively, this means that we distinguish the case of a process d using resource of type v once and then copying its output, from a process which duplicates it before letting two copies of d consume it.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

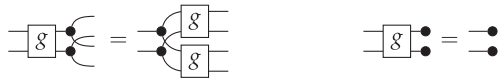
4.2.5 Cartesian Categories Often, one would like to go further than having an operation that allows us to split and end wires – certain SMCs extend the capability of these operations to *copy and delete boxes* too. This is the ability that cartesian categories[10] give us.

The term formation rules for cartesian categories are the same as those for 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 $d : v \to w$ in Σ_1, we have

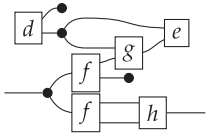


(4.5)

Note that this axiom scheme applies to generating operations with potentially multiple wires. To instantiate it, recall the definition of and for multiple wires given in (4.3). Then, if we apply these to $g : x_1 x_2 \to y_1 y_2$, we get



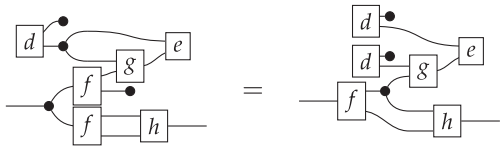
where we omit object labels for clarity. As for CD categories, we can now connect a given right port of a diagram to a (possibly empty) *set* of left ports of another:



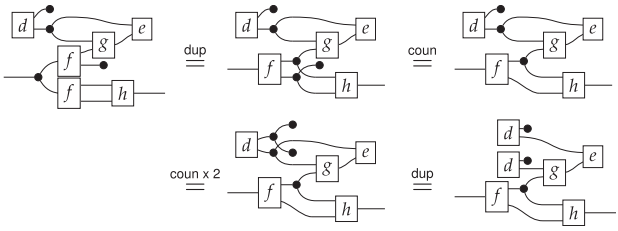
But the structural notion of equivalence for cartesian categories is much coarser. For the first time in this Element, we encounter a diagrammatic language where the structural equivalence is not topological, and where equivalence cannot simply be checked by examining the connectivity of the different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

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

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715



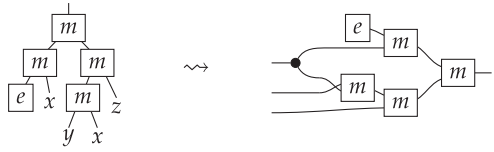
In this diagram, from left to right, we have merged the two occurrences of f and copied d; $(-\bullet \otimes \mathrm{id})$. It is helpful to break down the required equational steps:



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; and 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 (Cartesian categories and algebraic theories)**.** *The diagrammatic syntax of cartesian categories is the diagrammatic counterpart of the standard symbolic notation for algebraic theories. In this correspondence, wires take the place of variables. Since variables can be used arbitrarily many times, we need additional machinery in the diagrammatic setting to handle variable management: this is where* $-\bullet\!\subset$ *and* $-\bullet$ *come in. Moreover, just like we can substitute an arbitrary term for all occurrences of a given variable, we can copy and delete arbitrary diagrams using* $-\bullet\!\subset$ *and* $-\bullet$ *with the axioms* dup-del*. This is what allows us to interpret composition as* substitution*.*

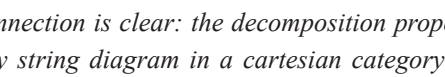
Let us examine the correspondence for a simple example; the general case is worked out (for the single-sorted case) in [23]. Consider the algebraic theory of monoids. It can be presented by two generating operations, $m(-,-)$ *of arity* 2 *and* e *of arity* 0 *(a constant) satisfying the following three 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 variable names, as on the left in the following diagram:*



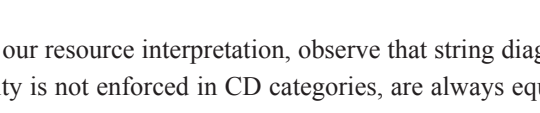
Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

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

We can also go in the other direction: from the diagrammatic syntax of a cartesian category over some signature, we can obtain an algebraic theory. This is done by noticing that every string diagram of such a category can be expressed as the composition of , *with diagrams that have a single outgoing wire. Indeed, the copying and deleting axioms imply that all string diagrams* $d : v \to w$*, with* $w = x_1 \dots x_n$*, can be decomposed uniquely into* n *diagrams* $d_i : v \to x_i$*,* $1 \leq i \leq n$*, where* x_i *are generating objects. Let us see concretely what this decomposition looks like and how to obtain the components for the case* $w = x_1 x_2$*. Let*



We can then check that



The general case is completely analogous.

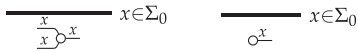
In the single-sorted case, that is*, when the set of objects contains a single generator, the connection is clear: the decomposition property just presented implies that every string diagram in a cartesian category can be seen as a composite of* *and* *with operations with* arity *corresponding to the number of left wires they have (and implicitly, co-arity one).*

Resuming our resource interpretation, observe that string diagrams in (4.4), whose equality is not enforced in CD categories, are always equated in cartesian categories. This means that cartesian categories are *resource-insensitive* by default, because they do not keep track of the interplay of processes and resources the same way CD categories do.

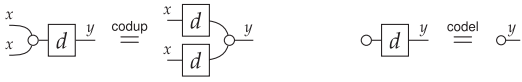
In applications, it is often interesting to enforce only a certain degree of (in)sensitivity, intermediate between CD and cartesian categories. A notable example is the one of *Markov categories*. In a Markov category we can always discard string diagrams, but we cannot copy them at will. More formally, their structure is defined by dropping from the definition of cartesian category the leftmost equation in (4.5). It turns out this setup is convenient for studying probabilistic computation, as it provides a baseline for interpreting string diagrams as stochastic processes. See Section 7 for more pointers to the literature on the topic.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

4.2.6 Cocartesian Categories If we flip all the diagrams of the previous section along the vertical axis, we obtain the duals of cartesian categories, namely *cocartesian categories*. They extend the language of symmetric monoidal categories, not with a commutative comonoid, but with a commutative monoid (see Example 2.13) instead:

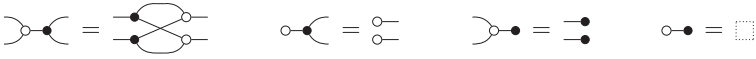


Furthermore, we want these to *merge* (or co-copy) and *spawn* (or co-delete) any diagram as follows:



(4.6)

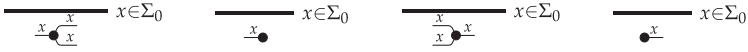
4.2.7 Biproduct Categories Categories that are both cartesian and cocartesian are called biproduct categories. They feature a comonoid and a monoid structure on each object, satisfying the dup-del and codup-codel axioms. Note one important consequence: if we apply dup to $d =$ ⊃○–, dup to $d =$ ○–, codup to $d =$ –●, and del to $d =$ ⊃○–, we get the following:



These are the defining axioms of bimonoids, as introduced in Example 2.16.

4.2.8 Hypergraph Categories Hypergraph categories further extend the capabilities of CD categories. Similar to 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, that is, those for symmetric monoidal categories, with the following additions:



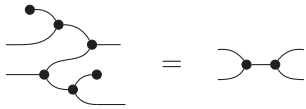
The first two are similar to the extra generators of CD categories. The last two are their mirror image. We write them in black instead of the white generators of cocartesian and biproduct categories since they will play a different role, as we will now see.

The corresponding notion of structural equivalence is given by the laws of symmetric monoidal categories with the addition of the axioms of special commutative Frobenius monoids (Example 2.18) summarised in (2.8). However, observe 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

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

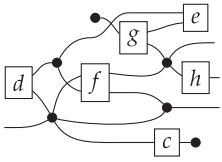
fact, as we will see later, these two requirements turn out to be incompatible in a rather fundamental way.

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 , , , and . In fact, as we have seen in Example 2.17, any two connected[11] 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 monoids to show that the following two diagrams are equivalent:

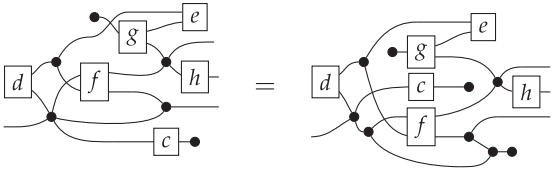


This means that the only relevant structure of a given connected diagram made entirely of , , , and is the number of ports on the left and on the right. As a result, as we saw in Example 2.17, we can introduce the following black nodes as syntactic sugar for any such diagram with m dangling wires on the left and n on the right:

Using this convenient notation, any string diagram in a hypergraph category will look like a hypergraph, as introduced in Section 3: boxes act as hyperedges, which may be wired together via black nodes.



In fact, this observation is what justifies the name 'hypergraph category' for these structures. Once more, two diagrams are equivalent if they connect the same ports via black nodes. For example, the following two are equivalent:

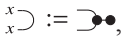


We could justify this equality through a sequence of Frobenius monoid axioms and the laws of symmetric monoidal categories, but it would be very time-consuming! It suffices to check that the connectivity of the different labelled

[11] In the sense that there is a path from any two nodes.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

boxes and black nodes remains the same. This is why hypergraph categories are very appealing.

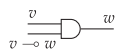
These examples lead us to observe that hypergraph categories are always self-dual compact closed.[12] With the previous intuition, this observation is not too surprising: if we are able to connect any set of ports, we can connect any two pairs of ports. More formally, we can define cups and caps as and  , for any x in the signature. That they satisfy the axioms of compact closed categories is a consequence of the Frobenius monoid axioms (or, more generally, of the spider theorem). We give the diagrammatic proof explicitly here, as it is instructive:

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 4.4 (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 theory that includes some additional generators and equations (*cf. *Section 2.1). For example, the free CD category over the theory* (Σ, E) *is definable as the free SMC over the theory formed by signature* $\left(\Sigma_0, \Sigma_1 \cup \left\{ \overset{x}{\underline{\bullet}}\!\!\overset{x}{\underline{\bullet}}, \overset{x}{\bullet} \mid x \in \Sigma_0 \right\}\right)$ *and equations, those in E plus those in* (4.2)*. Whether we pick one perspective or the other depends on which structure we want to see as built-in and which we want to see as domain-specific in the considered application.*

4.2.9 Closed Monoidal Categories In monoidal categories that are *closed*, the set of string diagrams $v \to w$ can be seen as an object $v \multimap w$ of the category itself. Closed monoidal categories arise naturally in applications where it makes sense to consider *higher-order* functions: processes that can take functions as inputs and can output other functions. Objects of the form $v \multimap w$ are called *exponentials*.

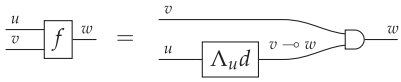
The existence of exponentials $v \multimap w$ for all v, w is not sufficient to form a closed monoidal category. We need extra conditions that encode the behaviour of $v \multimap w$ as some sort of function space. For this, we require the existence of a family of morphisms $\mathsf{eval}_{v,w} : v(v \multimap w) \to w$ depicted as



[12] This is in fact why they were called well-supported compact closed categories, when initially studied in [27].

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

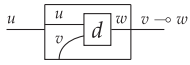
with the following property: for every $d : uv \to w$, there exists a unique morphism $\Lambda_u d : u \to (v \multimap w)$, such that



(4.7)

Intuitively, $\mathsf{eval}_{v,w}$ acts like an evaluation map that applies a function $v \to w$ to a value of type v and returns a w. In usual programming terms, $\Lambda_u d$ is a *curried* version of d which, given an argument of type u, returns a morphism $v \to w$. We can also see Λ as a family of morphisms, sending objects u and morphisms of type $uv \to w$ to morphisms of type $u \to (v \multimap w)$, for any u, v, w. In fact, this defines a *natural* one-to-one correspondences between sets of morphisms of these types[13]. This construction is also known as (λ-)*abstraction* in programming language theory.

It is possible to make the string diagrammatic language of closed monoidal categories even more appealing, by introducing a pictorial notation for the abstraction map Λ, represented as a box surrounding a given string diagram. For instance, given $d : uv \to w$, $\Lambda_u d : u \to (v \multimap w)$ becomes



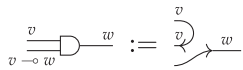
The different orientation of wires u and v in the box signals the different roles they play: intuitively, d awaits an input of type u on its left, in order to form a function with input of type v. As is the case with any string diagrammatic language introduced so far, this notation finds formal justification in terms of categorical structures: it is syntactic sugar for so-called *functorial boxes* [91], which capture the behaviour of Λ. We will not cover functorial boxes in detail here, though we recall briefly what they are in the Appendix.

The reader may find further details about the string diagrammatic language of closed monoidal categories in [65, section 3]. We do not discuss it further, given how different it is from our other examples. We conclude by linking the closed monoidal structure to other categorical structures seen in this section.

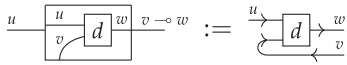
First, as the name suggests, compact closed categories (Section 4.2.2) are closed monoidal. The objects $v \multimap w$ are defined as wv^* and the evaluation maps $\mathsf{eval}_{v,w}$ are defined by:

[13] For a refresher on the categorical concept of naturality, see Definition A.5 in the Appendix. The categorically minded reader might also recognise an adjunction (*cf.* Definition A.9 in the Appendix) between the functor $(-)v$, which takes the monoidal product with an object v, and the exponential functor $v \multimap (-)$. In fact, a closed monoidal category is succinctly defined by the existence of such adjunction. We opted for a slightly different presentation, to emphasise the shape of string diagrams determining the closed structure.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715



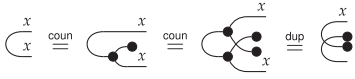
where $v \multimap w := wv^*$ in compact closed categories. Moreover, in compact closed categories, abstraction can be realised by bending a wire as follows:



We see that the resulting diagram has type $u \to wv^*$ as required. The ability to represent evaluation and abstraction as just wire bending is a special property of compact closed categories, which does not hold for generic closed monoidal ones.

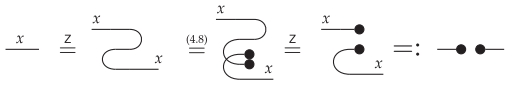
Second, another important class of closed categories are those that are also cartesian (Section 4.2.5). *Cartesian closed* categories are the semantics of choice for most functional programming languages. The cartesian structure witnesses the fact that resources (represented as variables) may be used arbitrarily many times in most programming languages, while the closed structure reflects the ability to manipulate higher-order functions. Once again, we refer the reader to [65] for an extensive discussion.

4.2.10 Mix and Match We have seen several cases in which categorical structures blend together to give rise to interesting combinations. But beware! Certain combinations have undesired consequences. The classic example is that of the incompatibility between cartesian and compact closed categories. This can be made precise as the following claim: a category that is both cartesian and compact closed is degenerate, in the sense that it has at most one morphism between any two objects.[14] To show this, it suffices to derive from the axioms of cartesian and compact closed categories that all morphisms between any two objects are equal. We achieve this by proving that we can disconnect any wire, in two steps: first we show that the cup splits as follows,



(4.8)

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

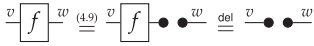


(4.9)

[14] Another way to phrase this is to say that the category collapses to a partially ordered set (poset). Indeed, we may regard a poset as a category whose objects are the poset elements and there is a morphism from x to y only when $x \leq y$. Thus each homset contains at most one morphism.

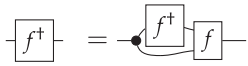
Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

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



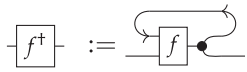
Therefore, modulo equivalence there is at most one string diagram of type $v \to w$. Note that the incompatibility between compact closed and cartesian structure implies that hypergraph and cartesian structure are also incompatible.

Interestingly, if we weaken the cartesian compact closed structure to that of a *cartesian traced* monoidal category, the resulting combination not only avoids degeneracy, but turns out to be closely related to the notion of (parameterised) fixed point [71]. A parameterised fixed-point operator in a cartesian SMC $(\mathcal{C}, \times, 1)$ takes a morphism $f : X \times A \to X$ and produces $f^\dagger : A \to X$. The operator $(-)^\dagger$ is then required to satisfy a certain number of intuitive axioms. For instance, $f^\dagger$ should indeed be a fixed points of f, that is,

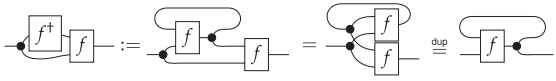


(4.10)

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

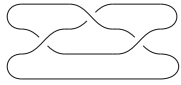


The axioms the fixed-point operator is required to satisfy are consequences of the axioms of traced monoidal categories with those of cartesian categories. For instance, it does satisfy (4.10):



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 notions of parameterised fixed points and cartesian traces are equivalent [71, Theorem 3.1].

We conclude by mentioning another example of a useful interaction between different structures. String 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 boundaries) diagrams are equal if and only if the corresponding knots can be topologically deformed into one another – thus, the topological notion of knot is fully captured by a few algebraic axioms. We see

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

here another advantage of working with string diagrams: they give an algebraic home to topological concepts that are otherwise difficult to express in standard algebraic syntax.

5 Semantics

So far, we have thoroughly explored an arsenal of diagrammatic syntax, each kind corresponding to a specific flavour of monoidal category. We have occasionally discussed what is the intended 'meaning' of these structures, that scientists have in mind when reasoning about a certain phenomenon with string diagrams. In this section, we explain how assigning meaning to string diagrams can be made formal, as a *semantics*.

Our approach to string diagram semantics draws inspiration from the study of *denotational semantics* of programming languages. In programming practice, computations are not mere manipulations of symbols, devoid of content; there is a task, a mathematical object, which we intend to describe with the program. A denotational semantics specifies what that 'something' is intended to be. It allows us to define the behaviour of programs in a given language rigorously, and prove more easily certain properties that they satisfy. The same language can even have different semantic interpretations. A well-chosen semantics may allow us to circumscribe more precisely the expressiveness of the language, or to rule out certain classes of behaviour.

5.1 From Syntax to Semantics, Functorially

Generally speaking, a semantics is a mapping from syntax to a domain of interpretation. Categorically, this idea may be applied to string diagrams using the ingredients introduced in the previous sections. Our starting point is a symmetric monoidal theory (Σ, E). Then string diagrams of the free symmetric monoidal category $\mathsf{Free}_{SMC}(\Sigma, E)$ over (Σ, E) is our syntax.

Now, in order to interpret such syntax, a domain of interpretation should mirror its basic structure. This is why we consider categories Sem that are symmetric monoidal for the task. A semantics for $\mathsf{Free}_{SMC}(\Sigma, E)$ will then be a mapping that preserves such structure, that is, a symmetric monoidal functor $[\![\cdot]\!] : \mathsf{Free}_{SMC}(\Sigma, E) \to \mathsf{Sem}$.

If $\mathsf{Free}_{SMC}(\Sigma, E)$ was a generic category, in order to define $[\![\cdot]\!]$ we would need to come up with a definition of $[\![v]\!]$ and $[\![d]\!]$ for any object v and string diagram d of $\mathsf{Free}_{SMC}(\Sigma, E)$. However, because $\mathsf{Free}_{SMC}(\Sigma, E)$ is freely generated by (Σ, E), our task is simpler. In order to fully define $[\![\cdot]\!]$, it suffices to assign it a value only on the generating objects and operations of Σ, and make sure they satisfy the relevant equations.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

More explicitly, giving a semantic interpretation of $\mathsf{Free}_{SMC}(\Sigma, E)$ in Sem amounts to specifying:

- an object $[\![x]\!]$ of Sem for each generating object $x \in \Sigma_0$;
- a morphism $[\![c]\!] : [\![x_1]\!] \otimes \cdots \otimes [\![x_n]\!] \to [\![y_1]\!] \otimes \cdots \otimes [\![y_m]\!]$ of Sem for each generating operation $c \in \Sigma_1$ of type $x_1 \ldots x_n \to y_1 \ldots y_n$;

Moveover, this should be done in such a way that the equations of E are satisfied, in the sense that $c \overset{E}{=} d$ implies $[\![c]\!] = [\![d]\!]$.

Giving such an interpretation for the generators completely defines a symmetric monoidal functor $[\![\cdot]\!] : \mathsf{Free}_{SMC}(\Sigma, E) \to \mathsf{Sem}$, in a canonical way. The semantics of an arbitrary object of $\mathsf{Free}_{SMC}(\Sigma, E)$, which is a word $w = x_1 \ldots x_n$ of generating objects, is computed as $[\![w]\!] = [\![x_1]\!] \otimes \cdots \otimes [\![x_n]\!]$. The semantics of an arbitrary (composite) string diagram is computed using the composition and monoidal product in the semantics, as long as the latter has the appropriate structure:

(5.1)

Finally, because $[\![\cdot]\!]$ should be a symmetric monoidal functor, symmetries ${}^{y}_{x}\!\times\!{}^{x}_{y}$ are mapped to symmetries $[\![x]\!] \otimes [\![y]\!] \to [\![y]\!] \otimes [\![x]\!]$ of Sem, and similarly for the identities.

In a sense, one may regard such description of $[\![\cdot]\!]$ as a definition of semantics by *structural induction* on string diagrammatic syntax. Remarkably, it is an inductive definition where we just need to specify the base cases, and the inductive step is always given by (5.1). It is worth emphasising once more that this style of definition is only possible because $\mathsf{Free}_{SMC}(\Sigma, E)$ is a *free* symmetric monoidal category. Our recipe for $[\![\cdot]\!]$ implicitly exploits the universal property of free constructions; see Remark 2.9. Also, note that (5.1) is what we commonly refer to as the property of *compositionality*: the semantics of a compound diagram is entirely determined by the semantics of its elementary components. Compositionality is a crucial property in software analysis, as it makes formal reasoning feasible at a large scale. Being able to reason semantically about graphical models using decompositions based on (5.1) is a major appeal of string diagrammatic approaches.

One last word about syntax. In the examples of semantics that we consider in the following sections, we will often remark that the domain of interpretation Sem has more structure than just symmetric monoidal. In particular, we will see categories that are also cartesian, hypergraph, and so on, in the sense

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

of Section 4. In such cases, it is often interesting to consider string diagrammatic syntax that also exhibits such structure, and study structure-preserving interpretations. To do so, we can introduce $\mathsf{Free}_X(\Sigma, E)$ for the free X-category over (Σ, E), where X stands for one of the structures considered in Section 4, for example, cartesian, hypergraph, and so on. These can be defined analogously to the free symmetric monoidal category $\mathsf{Free}_{SMC}(\Sigma, E)$, except that the built-in generators are not just the symmetries $\overset{y}{x}\!\times\!\overset{x}{y}$ and identities $\overset{x}{\vdash}\!\!\!\!\vdash\overset{x}{}$, but also the generators and equations specific to that structure. For instance, the free cartesian category $\mathsf{Free}_{Cart}(\Sigma, E)$ will have additional generators $\overset{x}{-}\!\bullet\!\overset{x}{\underline{x}}$ and $\overset{x}{-}\!\bullet$ for each generating object x, satisfying all the appropriate axioms.

5.2 Soundness and Completeness

Given a semantics $[\![\cdot]\!] : \mathsf{Free}_{SMC}(\Sigma, E) \to \mathsf{Sem}$, it is often insightful to understand which string diagrams c and d are identified by $[\![\cdot]\!]$, that is, when $[\![c]\!] = [\![d]\!]$. These equalities may inform us on the behaviour of the processes represented by string diagrams, and the differences of picking one semantics over another. First, the very existence of such a functor $[\![\cdot]\!]$ requires that $c \overset{E}{=} d$ implies $[\![c]\!] = [\![d]\!]$; otherwise, $[\![\cdot]\!]$ would not be well defined. Borrowing terminology from logic, in this case we say that E is *sound* for $[\![\cdot]\!]$. Furthermore E is said to be *complete* if the reverse implication holds. Compared to soundness, completeness is typically much harder to prove and often relies on identifying a 'canonical shape' for the morphisms of Sem. When we have that $c \overset{E}{=} d$ if and only if $[\![c]\!] = [\![d]\!]$, we say that the theory E is sound and complete, and call it an *axiomatisation* of the target SMC Sem. In categorical terms, E is sound and complete if $[\![\cdot]\!]$ is a faithful symmetric monoidal functor. The reader will often read that a theory is 'complete for Sem', rather than $[\![\cdot]\!]$, when the functor $[\![\cdot]\!]$ is clear from the context. Another question that is often relevant is how expressive a diagrammatic language is. That means we may investigate which behaviours lie in the image of $[\![\cdot]\!]$, and whether this image may be characterised by some property. In particular, if this image is (equivalent to) the whole of Sem, we say that $[\![\cdot]\!]$ is *full*.

5.3 Examples

We now cover useful examples of SMCs in which we can interpret different flavours of string diagrams. As we will see, some allow us to interpret symmetric monoidal theories with varying degrees of complexity. Some semantics have a special link with certain commonly occurring theories, in that the latter is *complete* for the former. In these cases, the string diagrams for a given theory capture exactly the semantics of interest.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

Each time, we will use the same notation $[\![\cdot]\!]$ for the semantic functor, and only specify its domain and codomain when necessary.

Example 5.1 (Functions, $\times$). The category Set of sets and functions can be equipped with a monoidal structure in different ways. The cartesian product of sets is one example of a monoidal product. On objects, it is simply the set of pairs, given by $X_1 \times X_2 = \{(x_1, x_2) \mid x_1 \in X_1 \wedge x_2 \in X_2\}$; on morphisms, it is given by $(f_1 \times f_2)(x_1, x_2) = (f(x_1), f_2(x_2))$. 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 from (2.3). As an exercise, let us prove the interchange law:

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

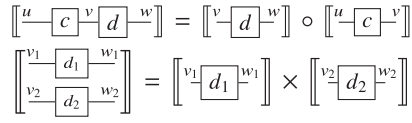
(Side note: strictly speaking, using pairs for '$\times$' does not define an associative monoidal product, because $(X_1 \times X_2) \times X_3$ is not *equal* to $X_1 \times (X_2 \times X_3)$, but merely isomorphic to it. See Remarks 2.5 and 5.4.) Furthermore, $(Set, \times)$ is a *symmetric* monoidal category, with symmetry given by the function σ_X^Y: $X \times Y \to Y \times X$ defined by $\sigma_X^Y(x, y) = (y, x)$.

Since Set is a symmetric monoidal category, we can use it to interpret string diagrams from a symmetric monoidal theory (Σ, E). Formally, this amounts to defining a symmetric monoidal functor

$$[\![\cdot]\!] : \mathsf{Free}_{SMC}(\Sigma, E) \to \mathsf{Set}$$

As explained in Section 5.1, this places significant constraints on $[\![\cdot]\!]$:

1. *Monoidal functoriality* means that $[\![v_1 v_2]\!] = [\![v_1]\!] \times [\![v_2]\!]$ for all $v_1, v_2 \in \Sigma_0^*$, that the identity wire $\overset{v}{—}$ over any object $v \in \Sigma_0^*$ is sent to the identity map over $[\![v]\!]$, and that the two compositions are preserved by $[\![\cdot]\!]$:



(5.2)

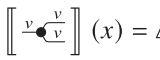
   In addition, $[\![\cdot]\!]$ is also a *symmetric* monoidal functor, so we necessarily have $[\![\chi_w^v]\!] = \sigma_Y^X$, for $[\![v]\!] = X$, $[\![w]\!] = Y$.
2. Since $\mathsf{Free}_{SMC}(\Sigma, E)$ is *free*, to fully specify such a monoidal functor $[\![\cdot]\!] : \mathsf{Free}_{SMC}(\Sigma) \to \mathsf{Set}$, it suffices to assign a set to each element of Σ_0

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

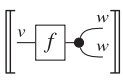
and a function $[\![c]\!] : [\![v]\!] \to [\![w]\!]$ for each operation $c : v \to w$ in Σ_1, such that they verify the axioms in E.

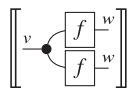
We see from points 1 and 2 that the semantics of an arbitrary string diagram d can be computed from the semantics of the generating operations of Σ and how they are composed together to form d, using (5.2).

This symmetric monoidal category also has the structure to interpret string diagrams for *cartesian* categories (Section 4.2.5). In fact, there is only one such structure. For an object v of a given signature, the comonoid structure ,
is given by the following copy $\Delta : [\![v]\!] \to [\![v]\!] \times [\![v]\!]$ and discarding $! : [\![v]\!] \to 1$ maps, defined respectively by:

 $(x) = \Delta(x) = (x, x)$ $\qquad$ $(x) = !(x) = \bullet$

One can easily check that these satisfy the axioms of commutative comonoids (Example 2.13) and that any function satisfies the equations dup and del from (4.5). To build a bit more intuition, let us verify dup, for example. For any $x \in [\![v]\!]$, we have

 $(x) = ($ $\circ [\![f]\!])(x) =$ $\big([\![f]\!](x)\big)$

$= \big([\![f]\!](x), [\![f]\!](x)\big) = \big([\![f]\!] \times [\![f]\!]\big)(x) =$  (x)

We encourage the reader to verify the other axioms as an exercise.

Note that there is only one map $X \to 1$ for any set X, namely the discarding map $!_X$, given by $!_X(x) = \bullet$. In conjunction with the counitality axiom of the comonoid structure, this condition forces the interpretation of the cartesian structure to be the one we have given – there are no other possible choices.

Remark 5.2 (Models of algebraic theories and cartesian categories)**.** *We have seen in Remark 4.3 that there is a close syntactic correspondence between cartesian categories and algebraic theories. It is natural to wonder whether the correspondence carries over to the semantic side. This is indeed the case: symmetric monoidal functors out of the free cartesian category over a given theory into the SMC* $(\mathsf{Set}, \times)$ *are models (in the usual algebraic sense) of the corresponding algebraic theory. For example, to specify such a functor for the cartesian theory of monoids involves choosing a carrier set X and functions $[\![m]\!] : X \times X \to X$, $[\![e]\!] : 1 \to X$ of the appropriate arity that satisfy the relevant axioms, which is precisely a model of the algebraic theory of monoids.*

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

Example 5.3 (Relations, $\times$). The category Rel has as objects, sets, and as morphisms $R : X \to Y$, binary relations, that is, subsets $R \subseteq X \times Y$. The composition of two relations $R : X \to Y$ and $S : Y \to Z$ 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 $X \times Y \to Y \times X$ given by the relation $\{((x,y),(y,x)) \mid x \in X, y \in Y\}$. It is easy to verify that these satisfy all the laws of symmetric monoidal categories. Even though this monoidal product is the same as in Set on objects, the properties of the two SMCs are very different.

Once again, given the free symmetric monoidal category $\mathsf{Free}_{SMC}(\Sigma, E)$ over some theory (Σ, E), specifying a symmetric monoidal functor $[\![\cdot]\!] : \mathsf{Free}_{SMC}(\Sigma, E) \to \mathsf{Rel}$ means assigning a set to each element of Σ_0 and a relation $[\![c]\!] \subseteq [\![v]\!] \times [\![w]\!]$ for each $c : v \to w$ in Σ_1 such that the axioms of E are satisfied in Rel. Here, monoidal functoriality, aka compositionality, means that, in particular:

$$\left[\!\!\left[\overset{u}{\text{—}}\boxed{c}\overset{v}{\text{—}}\boxed{d}\overset{w}{\text{—}} \right]\!\!\right] = \{(x,z) \mid \exists y\, ((x,y) \in [\![c]\!] \wedge (y,z) \in [\![d]\!])\}$$

$$\left[\!\!\left[\begin{array}{l} \overset{v_1}{\text{—}}\boxed{d_1}\overset{w_1}{\text{—}} \\ \overset{v_2}{\text{—}}\boxed{d_2}\overset{w_2}{\text{—}} \end{array} \right]\!\!\right] = \{((x_1,x_2),(y_1,y_2)) \mid (x_1,x_2) \in [\![d_1]\!] \wedge (x_2,y_2) \in [\![d_2]\!]\}$$

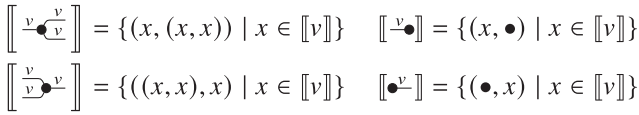
One can moreover interpret string diagrams for *self-dual compact closed* categories (Section 4.2.3) into Rel, by choosing a relation for the cups $\subset^{v}_{v}$ and caps $^{v}_{v}\supset$ on every generating object v of our signature, such that axiom (4.1) is satisfied. Once more, there are many possible choices, but the following interpretation is a common one:

$$[\![\subset^{v}_{v}]\!] = \{(\bullet, (x,x)) \mid x \in [\![v]\!]\} \qquad [\![{}^{v}_{v}\supset]\!] = \{((x,x), \bullet) \mid x \in [\![v]\!]\}$$

It is clear that these two relations satisfy the defining axiom (4.1) of (self-dual) compact closed categories.

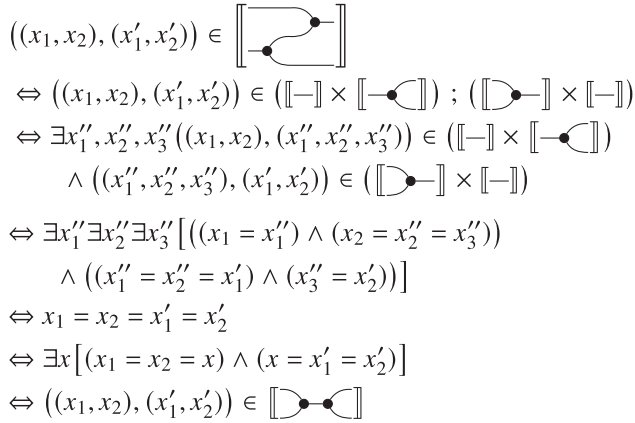
In fact, we can go even further: Rel can interpret string diagrams for *hypergraph* categories (Section 4.2.8). For this we need to choose a special, commutative Frobenius monoid to which we map $\overset{v}{\text{—}}\bullet\overset{v}{\underset{v}{\text{—}}}$, $\overset{v}{\text{—}}\bullet, \bullet\overset{v}{\text{—}}$, $\overset{v}{\underset{v}{\text{—}}}\bullet\overset{v}{\text{—}}$, for every generating object v of our chosen signature. There are many possibilities. One that occurs often in the literature is an extension of the comonoid structure chosen for functions in Example 5.1. We take the diagonal relation as comultiplication and the projection as counit, with the multiplication and unit given by the converse relations. Formally:

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715



$$\begin{aligned} &[\![\text{—}\bullet\!\!\subset\!\!\text{—}\,{}^{v}_{v}]\!] = \{(x,(x,x)) \mid x \in [\![v]\!]\} \qquad [\![\text{—}\bullet\,{}^{v}_{v}]\!] = \{(x,\bullet) \mid x \in [\![v]\!]\} \\ &[\![\text{—}\bullet\!\!\supset\!\!\text{—}\,{}^{v}_{v}]\!] = \{((x,x),x) \mid x \in [\![v]\!]\} \qquad [\![\bullet\!\!\text{—}\,{}^{v}_{v}]\!] = \{(\bullet,x) \mid x \in [\![v]\!]\} \end{aligned} \tag{5.3}$$

Let us check (one side of) the Frobenius law, to see how this works in more detail. In the following, all xs belong to $[\![v]\!]$ for some v, which we omit:



$$\begin{aligned} &\big((x_1,x_2),(x_1',x_2')\big) \in [\![\text{—}\bullet\!\!\supset\!\!\text{—}\,{}^{\bullet}_{\bullet}]\!] \\ &\Leftrightarrow \big((x_1,x_2),(x_1',x_2')\big) \in \big([\![\text{—}]\!] \times [\![\text{—}\bullet\!\!\subset\!\!\text{—}\,{}^{\bullet}_{\bullet}]\!]\big)\ ;\ \big([\![\text{—}\bullet\!\!\supset\!\!\text{—}\,{}^{\bullet}_{\bullet}]\!] \times [\![\text{—}]\!]\big) \\ &\Leftrightarrow \exists x_1'', x_2'', x_3''\big((x_1,x_2),(x_1'',x_2'',x_3'')\big) \in \big([\![\text{—}]\!] \times [\![\text{—}\bullet\!\!\subset\!\!\text{—}\,{}^{\bullet}_{\bullet}]\!]\big) \\ &\qquad \wedge \big((x_1'',x_2'',x_3''),(x_1',x_2')\big) \in \big([\![\text{—}\bullet\!\!\supset\!\!\text{—}\,{}^{\bullet}_{\bullet}]\!] \times [\![\text{—}]\!]\big) \\ &\Leftrightarrow \exists x_1''\exists x_2''\exists x_3''\big[\big((x_1 = x_1'') \wedge (x_2 = x_2'' = x_3'')\big) \\ &\qquad \wedge \big((x_1'' = x_2'' = x_1') \wedge (x_3'' = x_2')\big)\big] \\ &\Leftrightarrow x_1 = x_2 = x_1' = x_2' \\ &\Leftrightarrow \exists x\big[(x_1 = x_2 = x) \wedge (x = x_1' = x_2')\big] \\ &\Leftrightarrow \big((x_1,x_2),(x_1',x_2')\big) \in [\![\text{—}\bullet\!\!\supset\!\!\text{—}\,{}^{\bullet}_{\bullet}]\!] \end{aligned}$$

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 connected networks of black nodes force all variables labelling its left and right legs to be equal. With this in mind, one may observe that any connected network of black nodes forces all the corresponding variables to be the same. This can be seen as a semantic rendition of the spider theorem (covered in Example 2.17)! The special case we have shown earlier falls out as a corollary. Functions can also be seen as relations via their *graph*:

$$\mathsf{Graph}(f) = \{(x,y) \mid y = f(x)\}$$

Moreover, the composite (as relations) of two functional relations is the graph of the composite of the two corresponding functions, that is, $\mathsf{Graph}(g \circ f) = \mathsf{Graph}(f)\,;\,\mathsf{Graph}(g)$. In other words, Graph defines a functor $\mathsf{Graph} : \mathsf{Set} \to \mathsf{Rel}$. We call relations that are the graph of some function, *functional*.

Using $\text{—}\bullet\!\!\subset$, $\text{—}\bullet$ as above, we can interpret string diagrams for cartesian categories in Rel, since, as we have just seen, it contains Set. However, not all interpretations of a signature that includes $\text{—}\bullet\!\!\subset$, $\text{—}\bullet$ will satisfy the axioms of cartesian categories, unlike in Set. In fact, in Rel, it is possible to characterise functional relations purely by how they interact with $\text{—}\bullet\!\!\subset$, $\text{—}\bullet$: they are precisely those that satisfy the dup and del axioms in (4.5), as in the example of Set above. Indeed, a relation f satisfies dup if and only if it is single-valued, and it satisfies

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

del if and only if it is total. This is a useful characterisation that often comes up in the literature.

Finally, as we have mentioned, there are other choices of special, commutative Frobenius monoids in this SMC. The interested reader will find a full classification of all such choices in [96].

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,* that is, *the set* $(X \times Y) \times Z$ *is not* equal *to the set* $X \times (Y \times Z)$*. However, because of the coherence theorem for SMCs (Remark 2.5) 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.*

Similarly, we required the semantic functor $[\![\cdot]\!]$ *to be strict. To be fully formal, for many examples, we should allow* $[\![w_1 w_2]\!]$ *to merely be isomorphic to* $[\![w_1]\!] \otimes [\![w_2]\!]$*. However, for all intents and purposes, we can act as if* $[\![\cdot]\!]$ *was strict, with codomain the strictification of the semantics we have in mind, as we do here.*

Example 5.5 (Functions, +). The cartesian product is only one among several possible choices of monoidal structures that one can impose on the category of sets and functions. In fact, we can turn it into an 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, defined as $X_1 + X_2 := ((X_1 \times \{1\}) \cup (X_2 \times \{2\}))$ on objects, and given by $f_1 + f_2 = (f_1 + f_2)(x, i) = f_i(x)$ on maps. The unit for this monoidal product is the empty set. Moreover, it is also symmetric monoidal, with symmetry $\varsigma^X_Y : X + Y \to Y + X$ given by $\varsigma(z, 1) = (z, 2)$ and $\varsigma(z, 2) = (z, 1)$.

If we can no longer interpret diagrams for cartesian categories in this SMC, we can, however, interpret those for *cocartesian* categories (Section 4.2.6). For each generating object v of our chosen signature, we can interpret the commutative monoid operations $\overset{v}{\underline{\;\;}}\!\!\circ\!\!\overset{v}{\underline{\;\;}}$ and $\circ\!\!\overset{v}{\underline{\;\;}}$ as the following maps:

$$\left[\!\!\left[\overset{v}{\underline{\;\;}}\!\!\circ\!\!\overset{v}{\underline{\;\;}} \right]\!\!\right] (x, i) = x \text{ for } x \in [\![v]\!] \text{ and } i = 1, 2 \qquad [\![\circ\!\!\overset{v}{\underline{\;\;}}]\!] = \varnothing$$

It is a straightforward exercise to check that these are associative, unital and commutative. Moreover, every map (of the appropriate type) satisfies the codup and codel axioms with respect to $\overset{v}{\underline{\;\;}}\!\!\circ\!\!\overset{v}{\underline{\;\;}}$ and $\circ\!\!\overset{v}{\underline{\;\;}}$.

Amazingly, *every* map between finite sets can be represented using this syntax. We only need to give ourselves a single generating object $\bullet$ in our signature,

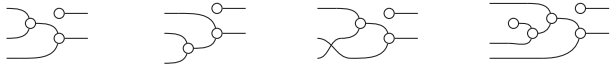
Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

which we interpret as the singleton set $[\![\bullet]\!] = 1$. Then we simply write and as and since object labels are redundant in this context. Given a map $f\colon X \to Y$, for X and Y two finite sets, we first fix some ordering of X and Y. In this way, we can identify them with finite sets of the form $\{0, \ldots, n\}$. This allows us to encode finite sets as sequences of wires in the diagrammatic setting (we will assume a similar encoding for several of the other examples that follow). With this encoding fixed, we can represent any map $f : X \to Y$: use as many as necessary to connect all elements x in the domain to the single $y = f(x)$ in the codomain to which f sends them; those elements of Y that are not in the image of f are each connected to a . Here are a few examples of the translation:

$$f : \{0,1,2\} \to \{0,1\} \qquad g : \{0,1,2,3\} \to \{0,1\} \qquad ! : \varnothing \to \{0,1\}$$

where 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 and . The following diagrams all represent $f : \{0,1,2\} \to \{0,1\}$ above:



In a cocartesian category, all these diagrams are equal, as and form a commutative monoid. This is the first instance of a monoidal theory we encounter that fully characterises the chosen semantics: *the free cocartesian category over a single object* (and no morphisms) is equivalent to the SMC $(\mathsf{fSet}, +)$ of finite sets and functions, with the disjoint sum as monoidal product. In other words, the symmetric monoidal theory of a commutative monoid is *complete* for this semantics (in the sense explained in Section 5.2). This also means that $\mathsf{Free}_{SMC}(\Sigma, E)$, the free symmetric monoidal category over $\Sigma = (\bullet, \{$ $,$ $\})$ and where E is the theory of commutative monoids (Example 2.13), is equivalent to $(\mathsf{fSet}, +)$. Any two diagrams made of and that denote the same map can be shown equal using only the axioms of commutative monoids.

The proof of this fact is typical for this kind of completeness result: it works by showing how, given an arbitrary Σ-diagram d, we can rewrite it to some normal form, using only the equations of commutative monoids. The chosen normal form is one from which the corresponding relation can be recovered uniquely: we can choose, for example, to eliminate all connected to a using unitality (un) and to associate all connected to the top using

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

associativity (as). Then, any two diagrams have the same normal form if and only if they are interpreted as the same map. The fact that any diagram can be rewritten to a normal form using only the preceding axioms, is typically proven by induction, considering each individual case, much like normalisation proofs in programming language theory, or cut elimination proofs in logic. Also, much like these, they tend to be quite tedious and combinatorial, so we do not reproduce them here.

Example 5.6 (Bijections, +). If we restrict the previous example to bijections (one-to-one and onto functions), we obtain the simplest example of a SMC – call it Bij. String diagrams for Bij are simply permutations of the wires! If we restrict further to finite ordinals, the resulting SMC is equivalent to $\mathsf{Free}_{SMC}(\Sigma)$, *the free SMC* over the signature $\Sigma = (\{\bullet\}, \varnothing)$, the SMC of permutations we have already encountered in Example 2.7.

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 $((x,i),(y,i)) \in R_1 + R_2$ if and only if $(x,y) \in R_i$ for some $i \in \{1,2\}$. Once again, the unit is the empty set and the symmetry is the graph of the corresponding function ς seen earlier. In this case, we cannot interpret string diagrams for compact closed nor hypergraph categories.

However, with the disjoint sum as monoidal product, we can interpret string diagrams for cartesian as well as cocartesian categories in (Rel, +). For each generating object v of a given signature, the monoid of the cocartesian structure is given by the graph of the relations that give the cocartesian structure to (Set, +):

$$\left[\!\!\left[\overset{v}{\underset{v}{\supset\!\!\circ\!\!-}} \right]\!\!\right] = \{((x,i),x) \mid x \in [\![v]\!], i = 1,2\} \qquad [\![\circ\!\!-^{v}]\!] = \varnothing$$

For the cartesian structure on with copying and deleting relations given by the converse of the preceding relations:

$$\left[\!\!\left[-\!\!\bullet\!\!\overset{v}{\underset{v}{\subset}} \right]\!\!\right] = \{(x,(x,i)) \mid x \in [\![v]\!], i = 1,2\} \qquad [\![-\!\!\bullet^{v}]\!] = \varnothing$$

One can check that $-\!\!\bullet\!\!\overset{v}{\underset{v}{\subset}}$, $-\!\!\bullet^{v}$ form a commutative comonoid for any interpretation of v, and that they can copy and delete any relation, that is, that they satisfy the dup and del axioms from (4.5). This means that we can interpret string diagrams for *biproduct* categories (Section 4.2.7) in (Rel, +).

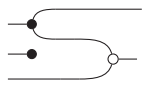
Intuitively, we can think of the diagrams $\supset\!\!\circ\!\!-$, $\circ\!\!-$, $-\!\!\bullet$, $-\!\!\bullet\!\!\subset$ in this category as directing the flow of a single token that travels around the wires. The intuition here is that the $\supset\!\!\circ\!\!-$ transfers to the right wire the token that comes through any

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

one of its left wires, ○— is able to generate a token, —●⊂ is a non-deterministic fork and —● a dead end. As we have already said, the relations for ⊃○— and ○— 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 non-deterministic —●⊂ and —●.

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 the variables 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*. A more systematic discussion of this perspective can be found in [1].

As in the previous example, we can represent *any* relation between finite sets, using only the signature $\Sigma = (\{\bullet\}, \{$—●⊂, —●, ○—, ⊃○—$\})$, where we set $[\![\bullet]\!] = 1$ and interpret —●⊂, —●, ○—, ⊃○— as the preceding relations. In our interpretation, a relation $R : X \to Y$ 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 $R : \{0, 1, 2\} \to \{0, 1\}$ given by $\{(0, 0), (0, 1), (2, 1)\}$ can be represented by the following diagram:

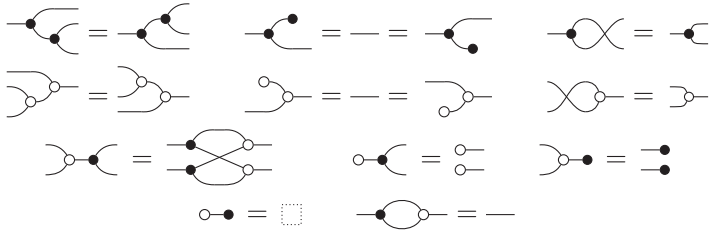


(5.4)

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, between determinism and non-determinism, reflected in the diagrams.

Note that a relation can also be seen as a matrix with Boolean coefficients. The relationship between the preceding string diagrams and matrices (over arbitrary semirings) will be explained in Example 5.14.

Not only do these string diagrams allow us to represent any relation between finite sets, we can also produce an axiomatisation of this SMC, that is, a sound and complete equational theory for the chosen semantics. To do this, we simply quotient Σ-diagrams by the axioms of an idempotent, commutative bimonoid:

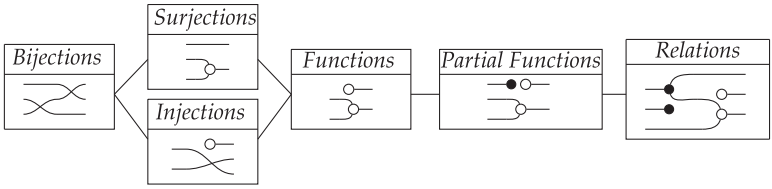


(5.5)

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

This equational theory turns out to be complete for our choice of $[\![\cdot]\!]$. In other words, any two diagrams c, d made from , , , are equal modulo the axioms in (5.5) if and only if $[\![c]\!] = [\![d]\!]$, that is, if and only if they denote the same relation.

Remark 5.8. *We just saw that going straight from functions to relations (with the disjoint sum as product) amounts to adding* *and* *to* *and* . *These two examples fit into a hierarchy of expressiveness, from bijections to relations:*



Of course, any subset of the generators , , , *gives a well-defined sub-SMC of* $(\mathsf{Rel}, +)$. *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 ways in which these theories are combined define distributive laws, a topic we mentioned in Remark 2.20.

Example 5.9 (Spans, $\times$). The category **Span**(Set) has sets as objects and, as morphisms $X \to Y$, pairs of maps $f : A \to X, g : A \to Y$ 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. In other words, they 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'.

The composition of two spans is obtained by computing what is called *the pullback* of g and p and composing the resulting outer two functions on each side:

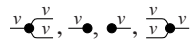
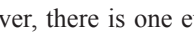
$$\begin{tikzcd}
 & & A \times_Y B \arrow[dl, "\pi_1"'] \arrow[dr, "\pi_2"] & & \\
 & A \arrow[dl, "f"'] \arrow[dr, "g"] & & B \arrow[dl, "p"'] \arrow[dr, "q"] & \\
X & & Y & & Z
\end{tikzcd}$$

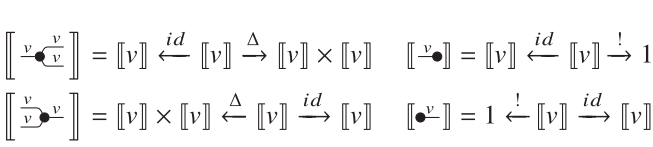
where $A \times_Y B := \{(a, b) \mid (g(a) = p(b)\}$ and π_1, π_2 are the two projections onto A and B. Thus, the composition of $X \xleftarrow{f} A \xrightarrow{g} Y$ followed by $Y \xleftarrow{p} B \xrightarrow{q} Z$ is $X \xleftarrow{f \circ \pi_1} A \times_Y B \xrightarrow{q \circ \pi_2} Y$. For a set X, the identity span is $X \xleftarrow{id_X} X \xrightarrow{id_X} X$. As

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

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 $h : A \to B$ 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, we can interpret string diagrams for hypergraph categories in the SMC of spans. There are many possible choices of where to map the Frobenius monoid  ,  for a given generating object v of a given signature. However, there is one evident choice, dictated by the presence of finite products:



$$\begin{aligned} \left[\!\left[\text{(diagram)} \right]\!\right] &= [\![v]\!] \xleftarrow{id} [\![v]\!] \xrightarrow{\Delta} [\![v]\!] \times [\![v]\!] \qquad [\![\text{(diagram)}]\!] = [\![v]\!] \xleftarrow{id} [\![v]\!] \xrightarrow{!} 1 \\ \left[\!\left[\text{(diagram)} \right]\!\right] &= [\![v]\!] \times [\![v]\!] \xleftarrow{\Delta} [\![v]\!] \xrightarrow{id} [\![v]\!] \qquad [\![\text{(diagram)}]\!] = 1 \xleftarrow{!} [\![v]\!] \xrightarrow{id} [\![v]\!] \end{aligned} \tag{5.6}$$

Here Δ is the usual diagonal map, defined by $\Delta(x) = (x,x)$, and ! is the unique map $[\![v]\!] \to 1$. The proof that these satisfy the axioms of commutative Frobenius monoids can be computed 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, in (5.3). This stems from a more general fact about spans and relations. If spans (of sets or any category with categorical products) of type $X \to Y$ can be seen as maps $A \to X \times Y$, relations are precisely *injective* (or monomorphic, in the general categorical setting) spans. One can always obtain a relation from a span $A \to X \times Y$ by first factorising the map into a surjective map (epimorphism) followed by an injective (monomorphism) one, and keeping only the latter. The interested reader will find the construction of Rel from **Span**(Set) explained in more detail in [61, 117].

Example 5.10 (Spans,+). As for functions and relations, spans can also be made into a symmetric monoidal category with the *disjoint sum* as monoidal product. On objects, it is defined in the same way, and on spans, it is given by taking the disjoint sum of each pair of legs of the two spans.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

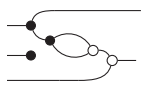
With this monoidal product, we can use the same signature as in Example 5.7 to represent any span of finite sets. Let $\Sigma = (\{\bullet\}, \{$ $, \bullet, \supset\!\!\circ\!\!-, \circ\!\!-\})$ with the following slightly modified interpretation: $[\![\bullet]\!] = 1$ and (omitting the only object label)

$$[\![\text{ }]\!] = 1 \xleftarrow{\nabla} 1+1 \xrightarrow{id} 1+1 \qquad [\![\text{ }]\!] = 1 \xleftarrow{0} \varnothing \xrightarrow{id} \varnothing$$

$$[\![\supset\!\!\circ\!\!-]\!] = 1+1 \xleftarrow{id} 1+1 \xrightarrow{\nabla} 1 \qquad [\![\circ\!\!-]\!] = \varnothing \xleftarrow{id} \varnothing \xrightarrow{0} 1$$

where $\nabla : 1+1 \to 1$ is defined by $\nabla(x,i) = x$ and $0 : \varnothing \to 1$ is the only map from the empty set. These also satisfy the axioms for string diagrams of biproduct categories; see Section 4.2.7. In fact, it is *the free biproduct category over a single object* and no additional morphism [84, section 5.3].

As for relations, we can use this signature to represent any span of finite sets with + as monoidal product: for the span $\{0,\ldots,n\} \xleftarrow{f} A \xrightarrow{g} \{0,\ldots,n\}$, 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 $\{0,1,2\} \xleftarrow{f} \{0,1,2,3\} \xrightarrow{g} \{0,1\}$, 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 the following diagram:



Notice that this diagram denotes the same *relation* as in (5.4), but the two represent different spans.

Another way to understand the correspondence is to observe 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 $\{0,\ldots,n\} \xleftarrow{f} A \xrightarrow{g} \{0,\ldots,n\}$, 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 will be explained in more detail in Example 5.14. 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 introduce next.

Example 5.11 (Cospans). Cospans, as their name indicates, are formed by inverting the arrows in the definition of spans. Let **Cospan**(Set) be the category with sets as objects and morphisms $X \to Y$ given by pairs of maps $f : X \to A$ and $g : Y \to A$ with the same set A as codomain, which we write as $X \xrightarrow{f} A \xleftarrow{g} Y$.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

The composition of two cospans is obtained by computing what is called *the pushout* of g and p and composing the resulting outer two functions on each side:

$$\begin{tikzcd}
 & & A +_Y B & & \\
 & A \arrow[ur, "\iota_1"'] & & B \arrow[ul, "\iota_2"] & \\
X \arrow[ur, "f"'] & & Y \arrow[ul, "g"] \arrow[ur, "p"'] & & Z \arrow[ul, "q"]
\end{tikzcd}$$

where $A +_Y B = \big(\{(a,1) \mid a \in A\} \cup \{(b,2) \mid b \in B\}\big)/\sim$ where $\sim$ is the equivalence relation defined by $(a,1) \sim (b,2)$ if and only if $a = g(y)$ and $b = p(y)$ for some $y \in Y$, and ι_1, ι_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 $X \xrightarrow{\iota_1 \circ f} A \times_Y B \xleftarrow{\iota_2 \circ q} Y$. 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 $\mathbf{Cospan}(\mathsf{Set})$ 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 $h : A \to B$ 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 $(f_1 + f_2)(x,i) = f_i(x)$; the empty set is the unit of this monoidal product and the symmetry is given by $X + Y \xrightarrow{id} X + Y \xleftarrow{\iota_2+\iota_1} Y + X$, where $\iota_1 : X \hookrightarrow X$, $\iota_2 : Y \hookrightarrow X$ are the injections into the first and second components given respectively by $\iota_1(x) = (x,1)$ and $\iota_2(y) = (y,2)$.

Once again, we can interpret Frobenius monoids into the SMC of cospans and thus draw string diagrams for hypergraph categories. For any generating object v of a chosen signature, let

$$\begin{aligned}
&\left[\!\!\left[\overset{v}{-}\!\!\bullet\!\!\overset{v}{\underset{v}{-}} \right]\!\!\right] = [\![v]\!] \xrightarrow{id} [\![v]\!] \xleftarrow{\nabla} [\![v]\!] + [\![v]\!] \qquad [\![\overset{v}{-}\!\bullet]\!] = [\![v]\!] \xrightarrow{id} [\![v]\!] \xleftarrow{0} \varnothing \\
&\left[\!\!\left[\overset{v}{\underset{v}{-}}\!\!\bullet\!\!\overset{v}{-} \right]\!\!\right] = [\![v]\!] + [\![v]\!] \xrightarrow{\nabla} [\![v]\!] \xleftarrow{id} [\![v]\!] \qquad [\![\bullet\!\overset{v}{-}]\!] = \varnothing \xrightarrow{0} [\![v]\!] \xleftarrow{id} 0
\end{aligned} \tag{5.7}$$

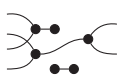
where $\nabla : [\![v]\!] + [\![v]\!] \to [\![v]\!]$ is defined as before by $\nabla(x,i) = x$ and $0 : \varnothing \to [\![v]\!]$ is the unique map from the empty set to $[\![v]\!]$. Notice the similarity (and differences) with (5.6).

In fact, there is more than a coincidental relationship between cospans and hypergraph categories: $(\mathbf{Cospan}(\mathsf{fSet}), +)$, the SMC of cospans restricted to finite sets, is equivalent to the *free hypergraph category on a single object* and no morphisms [84, section 5.4], that is, on the signature $\Sigma = (\{\bullet\}, \varnothing)$.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

Another way to say the same thing is that $(\mathbf{Cospan}(\mathsf{fSet}), +)$ is equivalent to $\mathsf{Free}_{SMC}(\mathsf{scFrob})$, the free SMC over the theory of a special commutative Frobenius monoid (see Example 2.18).

This means in particular that any cospan between finite sets (seen again as ordinals $\{0, \ldots, n\}$) can be represented as a diagram using only , (where we omit the single object label $\bullet$ once again) and $[\![\bullet]\!] = 1$. To build some intuition for this correspondence, take for example the pair $f : \{0, 1, 2, 3\} \to \{0, 1, 2\}$ and $g : \{0, 1\} \to \{0, 1, 2\}$, given by $f(0) = f(2) = 0$, $f(1) = f(3) = 1$, and $g(0) = g(1) = 1$; this cospan can be depicted as:



or, using spider notation (5.8)

The rule of thumb is easy to formulate: each element of the apex of the cospan – here, $\{0, 1, 2\}$ – 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 monoids, by the spider theorem, a result we saw in Example 2.17.

Completeness means that string diagrams in $\mathsf{Free}_{SMC}(\mathsf{scFrob})$ are equal if and only if they denote the same cospan. The proof of this fact is essentially the spider theorem from Example 2.18. 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. This is the same as defining a cospan!

Finally, many hypergraph categories can be seen as categories of cospans equipped with additional structure [52]. Interestingly, not *all* hypergraph categories can be described in this way. For that, we need the notion of *corelation*, which we cover in the next example.

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 [38]. It turns out that we can organise equivalence relations into a SMC. In fact, they can be organised into a hypergraph category.

A corelation $C : X \to Y$ is an equivalence relation (*i.e.* a reflexive, symmetric and transitive relation) over $X + Y$. Given two corelations $C : X \to Y$ and $D : Y \to Z$, their composition $C ; D : X \to Z$ is defined by glueing together equivalence classes from C and D along shared elements. To define it formally,

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

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 corelations to give a monoidal product. It is moreover symmetric with symmetry given by $[\![\times^{v}_{w}]\!] = \{\{(x,1),(x,2)\}x \in [\![v]\!]\} \cup \{\{(y,1),(y,2)\}y \in [\![w]\!]\}$. Once more, we can interpret the Frobenius monoids that define hypergraph categories in this SMC – for a generator v of our signature; let

$$\begin{aligned} [\![\overset{v}{-}\!\bullet\!\overset{v}{\underline{\subset}}\!\!\overset{v}{\underline{\;}}]\!] &= \{\{(x,1),((x,1),2),((x,2),2)\} \mid x \in [\![v]\!]\} \qquad [\![\overset{v}{-}\!\bullet]\!] = [\![v]\!] \\ [\![\overset{v}{\underline{\;}}\!\!\overset{v}{\underline{\;}}\!\!\bullet\!\overset{v}{-}]\!] &= \{\{((x,1),1),((x,2),1),(x,2)\} \mid x \in [\![v]\!]\} \qquad [\![\bullet\!\overset{v}{-}]\!] = [\![v]\!] \end{aligned} \tag{5.9}$$

In plain English, $\overset{v}{-}\!\bullet\!\overset{v}{\underline{\subset}}\!\!\overset{v}{\underline{\;}}$ (resp. $\overset{v}{\underline{\;}}\!\!\overset{v}{\underline{\;}}\!\!\bullet\!\overset{v}{-}$) is mapped to the equivalence relation over $[\![v]\!] + ([\![v]\!] + [\![v]\!])$ (resp. $([\![v]\!] + [\![v]\!]) + [\![v]\!]$) that identifies all occurrences of $x \in [\![v]\!]$ in the different components of the disjoint sum.

As we did for cospans, it is easy to represent any corelation between finite sets as a diagram using only $-\!\bullet\!\subset, -\!\bullet, \bullet\!-, \supset\!\bullet\!-$. For example, the corelation $\{0,1,2,3\} \to \{0,1\}$ given by the two equivalence classes $\{\{(0,1),(2,1)\}, \{(1,1),(3,1),(0,2),(1,2)\}\}$ over the disjoint sum $\{0,1,2,3\} + \{0,1\}$, can be depicted by any of the following string diagrams, using spider notation:

Observe that the first is the same diagram as in (5.8). The individual black dot, which represents an element of the apex of the cospan that was not in the image of any of the two leg maps, is missing in the second diagram. These two string diagrams represent the same corelation, since an isolated black dot represents an empty equivalence class: $[\![\bullet]\!] := [\![\bullet\!-\!\bullet]\!] = \varnothing$. 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 $\bullet\!-\!\bullet = \square$. However, the two string diagrams represent different cospans. This is an instance of a more general observation: at the semantic level, the only difference between corelations and cospans is that the former do not allow empty equivalence classes.

By the spider theorem, a network of black generators is fully characterised by its number of legs. Thus, there is only one such network, up to the laws of special commutative Frobenius monoids: the single dot $\bullet := \bullet\!-\!\bullet$. Putting all of the above together with the completeness result for cospans, we can get a

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

similar completeness result for corelations: for this, we need only add a single axiom to remove isolated dots to the theory of special commutative Frobenius monoids:

The resulting theory is known as the theory of *extraspecial* commutative Frobenius monoids [38, theorem 1.1].

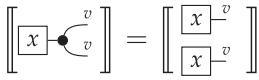
Recall that relations can be seen as jointly injective spans, that is, injective maps $R \hookrightarrow X \times Y$. Dually, corelations $C : X \to Y$ are jointly *surjective* cospans, that is, surjective maps $X + Y \twoheadrightarrow S$. We have already seen that, given a span $A \to X \times Y$, 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 $X + Y \to S$. As we have just seen, diagrammatically, this corresponds to removing isolated black dots. In category theory, the factorisation of Set maps into a 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, their apex 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* [53, 54].

Example 5.13 (Linear maps, $\otimes$). The category 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; suffice it 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 $(f_1 \otimes f_2)(u_1 \otimes u_2) = f(x_1) \otimes f_2(x_2)$. 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 $\sigma(x_1 \otimes x_2) = x_2 \otimes x_1$.

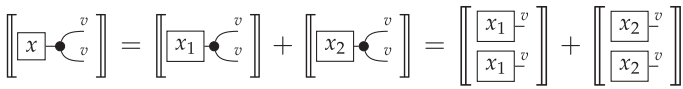
Crucially, this SMC is *not* cartesian: like in Rel, not all interpretations of a theory containing a commutative comonoid structure , for each generating object of the signature, satisfy the axioms of cartesian categories (dup and del). The intuition here is that, when we choose an interpretation of the comultiplication operation over $[\![v]\!]$, we also choose some set of vectors $x \in [\![v]\!]$ that this operation copies, that is, that verify:

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715



(5.10)

But then, given two such copyable vectors x_1, x_2, consider their sum $x = x_1 + x_2$ – we should have



$$= x_1 \otimes x_1 + x_2 \otimes x_2$$

On the other hand:

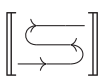
$$= x \otimes x = (x_1 + x_2) \otimes (x_1 + x_2)$$
$$= x_1 \otimes x_1 + x_1 \otimes x_2 + x_2 \otimes x_1 + x_2 \otimes x_2$$

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, we can interpret string diagrams for *compact closed* categories in (fVect, $\otimes$) (recall that the requirement of cartesian-ness and compact closed-ness are incompatible in the sense explained in Section 4). For a given generating object v, its dual v^* is interpreted as the algebraic dual of $[\![v]\!]$ in the usual sense, that is, as $[\![v]\!]^*$, the space of linear maps $[\![v]\!] \to \mathbb{K}$. Then, the cap on a vector space $[\![v]\!]$ is the unique linear map $[\![v]\!]^* \otimes [\![v]\!] \to \mathbb{K}$ that satisfies $[\![{}^{v}_{v}\!\circlearrowleft]\!](f \otimes x) = f(x)$ (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 $[\![v]\!]$ and a dual basis $\{f_i\}_i$ of $[\![v]\!]^*$ in the sense that $f_i(e_j) = 1$ if $i = j$ and 0 otherwise; $[\![\circlearrowright^{v}_{v}]\!]$ is then the map $\mathbb{K} \to [\![v]\!] \times [\![v]\!]^*$ 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:

$$[\![\circlearrowright^{v}_{v}]\!](k) = \sum_i k(e_i \otimes f_i) \qquad [\![{}^{v}_{v}\!\circlearrowleft]\!]\left(\sum_{i,j} \lambda_{i,j}(e_i \otimes f_j)\right) = \sum_i \lambda_i f_i(e_i)$$

However, observe that the resulting maps are independent of the specific choice of bases. With these expressions, we can verify the yanking equation for $\circlearrowright^{v}_{v}$ and ${}^{v}_{v}\!\circlearrowleft$. Let u of $[\![v]\!]$ such that $x := \sum_i \lambda_i e_i$; we have:

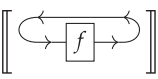


$$(x) = ([\![\circlearrowright]\!] \otimes [\![\rightarrowtail]\!]) \; ; \; ([\![\rightarrowtail]\!] \otimes [\![\circlearrowleft]\!])(x)$$
$$= ([\![\rightarrowtail]\!] \otimes [\![\circlearrowleft]\!])\left(\left(\sum_i (e_i \otimes f_i)\right) \otimes x\right) = \sum_i f_i(x) e_i$$
$$= \sum_i f_i\left(\sum_i \lambda_j e_j\right) e_i = \sum_i \sum_j \lambda_j f_j(e_j) e_i = \sum_i \lambda_i e_i =: x.$$

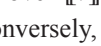
Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

In this SMC, the elements of the vector space $[\![v]\!]$ are precisely the morphisms $\mathbb{K} \to [\![v]\!]$.

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

$$[\![\text{  }]\!] = \mathsf{Tr} f = \mathsf{Tr} A = \sum_i a_{ii}$$

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 $\mathsf{Tr}(AB) = \mathsf{Tr}(BA)$ or, more generally, that it is invariant under circular shifts.

Another important feature is that commutative and special Frobenius monoid in $(\mathsf{fVect}, \otimes)$ correspond to a choice of a basis for the supporting vector space [36, section 6]. Even if there is no linear copying map for all the elements of a vector space, we have seen earlier that, when we choose a comultiplication operation over $[\![v]\!]$ we also choose some set of elements $x \in [\![v]\!]$ that  copies. Conversely, given any basis, we can define a comonoid operation that copies its elements, that is, 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 $(\mathsf{fVect}, \otimes)$ is more commonly known as an *algebra*. Any basis defines not only a comonoid but an algebra given by extending the comparison map $e_i \otimes e_j \mapsto \delta^i_j e_i$ by linearity. Not only that, the corresponding monoid–comonoid pair satisfies the Frobenius axioms and defines a special and commutative Frobenius monoid. Conversely, the copyable states of any commutative and special Frobenius monoid form a basis of $[\![v]\!]$. The last direction is more difficult to prove, and we will not do so here. Instead, we refer the interested reader to the lecture notes of Vicary and Heunen, who deal with a related case in detail [73, chapter 5] and use string diagrams throughout.

A historical note: one of the earliest instances of string diagrams is *Penrose graphical notation* [101] for working with tensors, which are precisely string diagrams for $(\mathsf{fVect}, \otimes)$, later systematised and generalised in [77].

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 $\sigma^Y_X(x, y) = (y, x)$, the resulting structure is a SMC. It is well known that isomorphic finite-dimensional vector spaces

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

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 fVect whose objects are $\mathbb{K}^n$ for some $n \in \mathbb{N}$. We can go even further: given a linear map $\mathbb{K}^m \to \mathbb{K}^n$, we can identify it with its representation in the canonical bases of $\mathbb{K}^m$ and $\mathbb{K}^n$. We call $\mathsf{Mat}_{\mathbb{K}}$ the category whose objects are natural numbers (representing the dimension of a vector space) and morphisms $m \to n$ are $n \times m$ matrices (notice the reversal). Nothing is lost, since $\mathsf{Mat}_{\mathbb{K}}$ and fVect are equivalent.

The SMC $(\mathsf{Mat}_{\mathbb{K}}, \oplus)$ can interpret diagrams for cartesian categories: given any object v of some signature, the canonical comonoid structure over the vector space $[\![v]\!] = \mathbb{K}^n$ is given by

$$[\![\ \text{—}\bullet\!\!\subset\!\!\overset{v}{\underset{v}{\text{—}}}\ ]\!]\,(x) = (x, x) \qquad\qquad [\![\ \overset{v}{\text{—}}\bullet\ ]\!]\,(x) = \bullet$$

Since linear maps are maps with extra structure, this comonoid is inherited from Set (see Example 5.1) and the proof that it satisfies the axioms of comonoid as well as the copying and deleting axioms dup and del, is similar. We can also interpret diagrams for cocartesian categories in $\mathsf{Mat}_{\mathbb{K}}$: for any object v, the monoid structure is given by addition and zero:

$$[\![\ \overset{v}{\underset{v}{\text{—}}}\!\!\supset\!\!\circ\!\!\overset{v}{\text{—}}\ ]\!]\,(x_1, x_2) = x_1 + x_2 \qquad\qquad [\![\ \circ\!\!\overset{v}{\text{—}}\ ]\!] = 0$$

Note that the comonoid and monoid do not interact to form a Frobenius monoid but a *bimonoid* (see Section 2.16). In fact, we have even more structure: these string diagrams satisfy the axioms of *biproduct categories* (Section 4.2.7). This means that maps satisfy not only 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 (see 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 preceding bimonoid, we are very close to being able to express all matrices diagrammatically. As before, we will use the signature $\Sigma = (\{\bullet\}, \{\text{—}\bullet\!\!\subset, \text{—}\bullet, \supset\!\!\circ\!\text{—}, \circ\!\text{—}\} \cup \{\text{—}\boxed{a}\text{—} \mid a \in \mathbb{K}\}$. We interpret the single generating object as $[\![\bullet]\!] = \mathbb{K}$. Contrary to the case of relations, we cannot express arbitrary matrices with just $\text{—}\bullet\!\!\subset, \text{—}\bullet, \supset\!\!\circ\!\text{—}, \circ\!\text{—}$; this is why we have added a new generating operation $\text{—}\boxed{a}\text{—}$ for each $a \in \mathbb{K}$, intended to represent scalar multiplication and interpreted correspondingly:

$$[\![\ \text{—}\boxed{a}\text{—}\ ]\!]\,(x) = ax$$

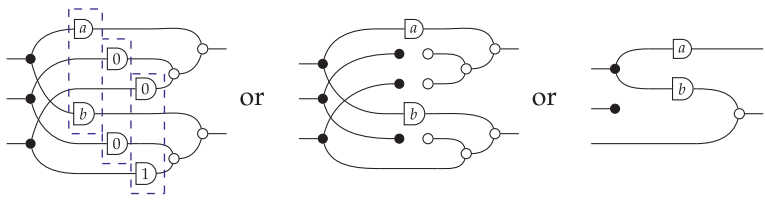
Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

Before explaining the encoding of matrices, there are a few special cases of $-\boxed{a}-$ that we should mention: multiplying by one is the same as the identity, so $[\![-\boxed{1}-]\!] = [\![-]\!]$, and the result of multiplying by zero is always zero, so $[\![-\boxed{0}-]\!] = [\![-\bullet\ \circ-]\!]$.

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 $[\![-\boxed{a}-]\!] = [\![-]\!]$, we can simply draw the connection as a plain wire when $a_{ij} = 1$ and since $[\![-\boxed{0}-]\!] = [\![-\bullet\ \circ-]\!]$ 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

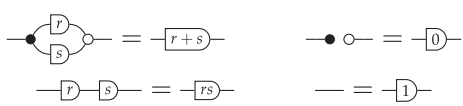
$$A = \begin{pmatrix} a & 0 & 0 \\ b & 0 & 1 \end{pmatrix}$$

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.

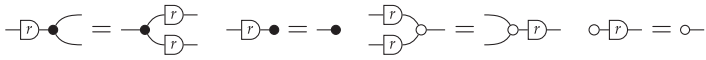
Amazingly, we can then quotient the diagrammatic syntax by an equational theory that makes these three equal. More generally, we can give an axiomatisation of $(\mathsf{Mat}_{\mathbb{K}}, \oplus)$. The equational theory is very similar to that of relations with the disjoint union. It has all axioms of (5.5) *except* the last one, namely $-\bullet\!\!-\!\!\circ\!\!- = -$ (which encodes $x+x = x$, a specific feature of the Boolean semiring). Furthermore, we need axioms that encode the additive and multiplicative structure of $\mathbb{K}$, namely:



(5.11)

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

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715



(5.12)

Taken with the axioms of (5.5) *minus* the last one, the axioms listed in (5.11)–(5.12) give a complete theory for matrices over $\mathbb{K}$: diagrams modulo these equations are equal if and only if they denote the same matrix.

Note that 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 generating operations $-\boxed{a}-$ for all $a \in \mathsf{R}$. 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 $(\mathbf{Span}(\mathsf{fSet}), +)$ (Example 5.10).

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 $X \to Y$ is a linear subspace of $X \oplus Y$, that is, 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 $\mathsf{LinRel}_{\mathbb{K}}$ the category whose objects are natural numbers and morphisms $m \to n$ are linear relation $\mathbb{K}^m \to \mathbb{K}^n$. With the direct sum, $\mathsf{LinRel}_{\mathbb{K}}$ becomes a SMC, with unit and symmetry the same as those of $\mathsf{Mat}_{\mathbb{K}}$ (see the 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 $\mathsf{Graph}(f)$, by taking its graph: $\mathsf{Graph}(f) := \{(x,y) \mid y = f(x)\}$. Thus, we can also interpret the diagrams that allowed us to depict linear maps/matrices diagrammatically in $\mathsf{LinRel}_{\mathbb{K}}$: taking , where each wire represent a single generating object $\bullet$, interpreted as $[\![\bullet]\!] = \mathbb{K}$. Their interpretation as relations is given by the graph of the corresponding maps:

$$[\![\text{}]\!] = \{(x,(x,x)) \mid x \in \mathbb{K}\} \qquad [\![\text{}]\!] = \{(x,\bullet) \mid x \in \mathbb{K}\}$$

$$[\![\text{}]\!] = \{((x_1,x_2),x_1+x_2) \mid x_1,x_2 \in \mathbb{K}\} \qquad [\![\text{}]\!] = \{(0,\bullet)\}$$

Interestingly, the converses of these relations are also linear; to depict them, we add to our signature the mirror image of the corresponding diagrams: , with semantics given by

$$[\![\text{}]\!] = \{((x,x),x) \mid x \in \mathbb{K}\} \qquad [\![\text{}]\!] = \{(\bullet,x) \mid x \in \mathbb{K}\}$$

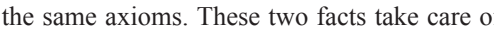
$$[\![\text{}]\!] = \{(x_1+x_2,(x_1,x_2)) \mid x_1,x_2 \in \mathbb{K}\} \qquad [\![\text{}]\!] = \{(\bullet,0)\}$$

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

Notice that these are the same relations as the first two, with the pairs flipped. We can do this for any map f: let $\mathsf{coGraph}(f) := \{(y,x) \mid f(x) = y\}$. 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 now cover.

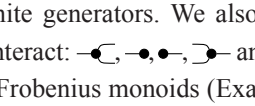
There is a complete axiomatisation of $\mathsf{LinRel}_{\mathbb{K}}$ with the direct sum, which is sometimes called the theory of *Interacting Hopf algebras*, or IH for short. The reader will find the complete theory and further details in [22]. We discuss its most salient features in what follows, less formally.

As for Example 5.7, the image by $[\![\cdot]\!]$ of diagrams made from , , , are precisely those relations that are the graph of some linear map (aka a matrix). For these, 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-swap , , , satisfy exactly the same axioms. These two facts take care of all interactions between the black and white generators. We also need to specify how diagrams of the same colour interact: , , , and , , , both form extraspecial commutative Frobenius monoids (Example 2.17)! The remaining axioms specify the behaviour of scalars , which may now encounter their mirrored version:



for $a \neq 0$

where



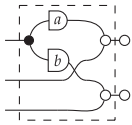
These axioms force the mirrored version of to be division by a, which we have over any field, as long as a is non-zero. The two cups and caps are also related in an obvious way:

From these axioms, most of linear algebra can be reformulated, with subspaces and linear maps on an equal (diagrammatic) footing.

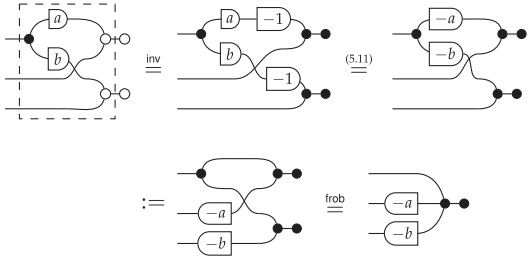
As an elementary illustration of the basic principles of diagrammatic reasoning in linear algebra, let us look at systems of linear equations. The idea is simple: a system of linear equations in the form $Ax = 0$ can be expressed by simply plugging into the right side of a diagram that encodes the matrix A (Example 5.14), that is, by the diagram . For example,

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

$$\begin{pmatrix} a & 1 & 0 \\ b & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad \text{becomes}$$



Computing a basis of the set of solutions then involves rewriting the diagram into a form from which any solution can be generated easily:



Here we find that the kernel of A has dimension 1, with basis vector, for example, $\begin{pmatrix} 1 & -a & -b \end{pmatrix}^T$.

Remark 5.16. *If we identify the complete equational theory of a particular structure in a semantic model, we can seek it in different models, and thereby identify seemingly unrelated algebraic objects as instances of the same abstract structure. For example, we have seen many different interpretations of Frobenius monoids 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 [34, 47].*

Example 5.17 (Monotone relations, $\times$). So far, all the examples of compact closed categories we have covered (spans, relations, cospans, corelations) also have the structure of hypergraph categories. Of course, there are compact closed categories where the compact structure does not come from some chosen Frobenius monoid. The category of *monotone relations* is one such example. It has pre-ordered sets as objects (that is, sets equipped with a reflexive and transitive binary relation); its morphisms $(X, \preceq) \to (Y, \leq)$ are relations $R \subseteq X \times Y$ that preserve the order in the following sense: if $(x, y) \in R$ and $x' \preceq x, y \leq y'$, 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

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

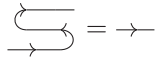
two monotone relations is monotone, they form a category, with identity on each object $(X, \preceq)$ given by the pre-order relation $\preceq \subseteq X \times X$ itself.

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

As anticipated, this category can interpret the string diagrams of compact closed categories. For this, given an object v of a given signature, we need to define its dual v^*: if $[\![v]\!] = (X, \preceq)$, then $[\![v^*]\!] = (X, \succeq)$, the same underlying set with the opposite pre-order relation $\succeq := \preceq^{op}$. The cups and caps on each object are then given by

$$\left[\!\!\left[\begin{smallmatrix} \succ v \\ \subset v \end{smallmatrix} \right]\!\!\right] = \{(\bullet, (x', x)) \mid x \preceq x'\} \qquad \left[\!\!\left[\begin{smallmatrix} v \prec \\ v \supset \end{smallmatrix} \right]\!\!\right] = \{((x, x'), \bullet) \mid x' \preceq x\}$$

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



The left-hand side of this equation has the following semantics:

$$\begin{aligned}
\left[\!\!\left[\begin{smallmatrix} \succ \\ \subset \\ \succ \end{smallmatrix} \right]\!\!\right] &= \left([\![\succ \subset]\!] \times [\![\succ\!\!-]\!] \right) ; \left([\![\succ\!\!-]\!] \times [\![\supset \prec]\!] \right) \\
&= \left(\{(\bullet, (x, x')) \mid x \preceq x'\} \times \preceq \right) ; \left(\preceq \times \{((x, x'), \bullet) \mid x \preceq x'\} \right) \\
&= \left(\{(x, (x_1, x_2, x_3)) \mid x_2 \preceq x_1 \wedge x \preceq x_3\} \right) ; \\
&\qquad \left(\{(x_1, (x_2, x_3), x') \mid x_1 \preceq x' \wedge x_3 \preceq x_2\} \right) \\
&= \left\{ (x, x') \mid \exists x_1 \exists x_2 \exists x_3 \left[x \preceq x_3 \wedge x_3 \preceq x_2 \wedge x_2 \preceq x_1 \wedge x_1 \preceq x' \right] \right\} \\
&= \{(x, x') \mid x \preceq x'\}
\end{aligned}$$

where the last step holds 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 an interesting monoid and comonoid structure: given the signature $\left(\{\bullet\}, \succ\!\!\bullet\!\!\begin{smallmatrix}\succ \\ \subset \\ \succ\end{smallmatrix}, \succ\!\!\bullet, \bullet\!\!\succ, \begin{smallmatrix}\succ \\ \subset \\ \succ\end{smallmatrix}\!\!\bullet\!\!\succ \right)$, we can interpret its generators as follows: let $[\![\bullet]\!] = (X, \preceq)$ and

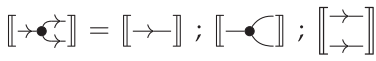
$$\begin{aligned}
[\![\succ\!\!\bullet\!\!\begin{smallmatrix}\succ \\ \subset \\ \succ\end{smallmatrix}]\!] &= \left\{ (x, (x_1', x_2') \mid x \preceq x_1' \text{ and } x \preceq x_2') \right\} \\
[\![\succ\!\!\bullet]\!] &= \{(x, \bullet) \mid x \in X\}
\end{aligned} \tag{5.13}$$

$$\begin{aligned}
[\![\begin{smallmatrix}\succ \\ \subset \\ \succ\end{smallmatrix}\!\!\bullet\!\!\succ]\!] &= \{((x_1, x_2), x') \mid x_1 \preceq x' \text{ and } x_2 \preceq x')\} \\
[\![\bullet\!\!\succ]\!] &= \{(\bullet, x) \mid x \in X\}
\end{aligned} \tag{5.14}$$

That these satisfy the monoid and comonoid axioms respectively is a simple exercise. 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

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

partial order relation on each wire: for example, if we momentarily reinterpret $[\![\cdot]\!]$ as a mapping into Rel (since monotone relations are, after all, relations), we have that



since $[\![\rightarrow\!\!-]\!] = \{(x,x') \mid x \leq x'\}$.

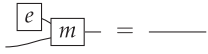
It should be noted that interpreted in this way do not necessarily form a Frobenius monoid, nor do they give rise to a (co)cartesian structure. Their interaction is still interesting, and can be axiomatised, but doing so requires turning to theories with *inequalities* rather than just equalities. We will look at these briefly in Section 6.3 and at monotone relations again in Example 6.5.

6 Other Trends in String Diagram Theory

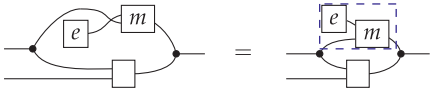
6.1 Rewriting

When reasoning about programs, *reductions* of a program p into another one q are important objects of study: such reductions may witness, for instance, the evaluation of p on a certain input (akin to β-reduction in the λ-calculus [8]), 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 with 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



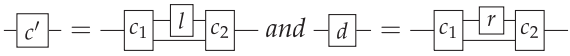
We claim the rule has a matching in the following string diagram on the left. However, strictly speaking, the matching isolates a subterm only when we 'massage' the string diagram as on the right:



Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

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 be a signature, $l \rightsquigarrow r$ be a rewrite rule of Σ-terms, and c be a Σ-term. We say that c rewrites into d modulo E if* $=_E$ *and*



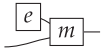
(6.1)

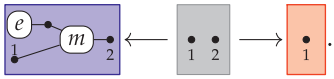
The standard case is when E are the laws of SMCs, but the notion can be adapted to fit other categorical structures where string diagrams occur, as those illustrated in Section 4. This definition seems reasonable enough from a mathematical viewpoint. However, it is completely unpractical when it comes 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 not even terminate given that, in 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 with 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 to orient such equations to turn them into rewrite rules.

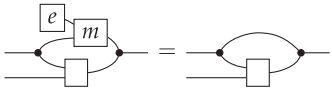
The way out of this impasse comes from the combinatorial interpretation of string diagrams, introduced in Section 3. Recall that under this interpretation, equivalent Σ-terms are mapped to a *single* open hypergraph. In other words, if c is mapped to $[\![c]\!]$, and c and d are equivalent modulo the laws of SMCs, then $[\![c]\!] = [\![d]\!]$. 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 just presented, 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* [50]. The fundamental result linking double-pushout rewriting and syntactic rewriting is the following:

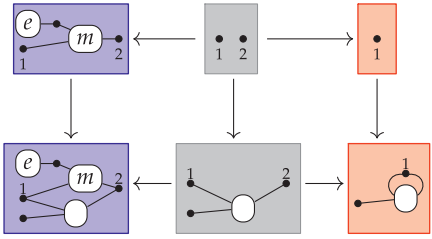
Theorem 6.2. *c rewrites into d modulo* scFrob *if and only if the open hypergraph $[\![c]\!]$ rewrites into $[\![d]\!]$ modulo double-pushout rewriting.*

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

Example 6.3. The rule  is interpreted as the span



Its application in the rewrite step  is interpreted as the double-pushout rewriting step



Intuitively, double-pushout rewriting matches the left-hand side of the rewrite rule to a subgraph and replaces it with the right-hand side.

Theorem 6.2 is a consequence of the correspondence established by Theorem 3.5 between string diagrams modulo Frobenius monoid 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 6.4. *c rewrites into d modulo the laws of SMCs if and only if $[\![c]\!]$ rewrites into $[\![d]\!]$ modulo convex double-pushout rewriting.*

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

The preceding two theorems settle the question of rewriting for string diagrams in symmetric monoidal categories (Theorem 6.4) 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 cartesian monoidal categories. It has been answered recently in [93]

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

for copy-delete monoidal categories,[15] and in [67] for traced copy-delete categories. Finally, we point out the recent work [2], which studied rewriting for monoidal closed categories.

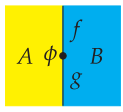
6.2 Higher-Dimensional Diagrams

The string diagrams we have considered so far are a particular case of a more general notation for higher categories. We highlight the connection in this section. First, we need to explain what an n-category is. Given the limited scope of our work, we will confine ourselves to a sketch, which should, however, be sufficient to grasp the link between the graphical language(s) of n-categories 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 notion.

Very roughly, an n-category is a category that may have 2-morphisms between morphisms, 3-morphisms between 2-morphisms, and so on. For example, in a 2-category there can be 2-morphisms between morphisms, indicated as follows:

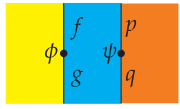
$$\begin{tikzcd} A \arrow[r, bend left=60, "f", "g"'] \arrow[r, phantom, "\Downarrow\phi"] & B \end{tikzcd}$$

Like monoidal categories, 2-categories also admit a visual representation, called *surface diagrams*: objects are represented as (labelled/shaded) regions of space, morphisms as (labelled) strings or wires, and 2-morphisms as (labelled) dots.



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

$$\begin{tikzcd} A \arrow[r, bend left=60, "f", "g"'] \arrow[r, phantom, "\Downarrow\phi"] & B \arrow[r, bend left=60, "p", "q"'] \arrow[r, phantom, "\Downarrow\psi"] & C \end{tikzcd}$$

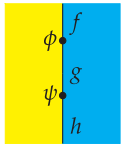


[15] The work [59], which appeared at the same time as [93] also establishes the correspondence between string diagrams in copy-delete categories and suitably defined open hypergraphs, but without considering the corresponding notion of rewriting.

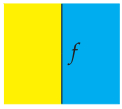
Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

and vertically

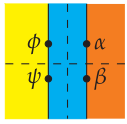
$$\begin{tikzcd} A \arrow[r, bend left=60, "f"{name=U}] \arrow[r, "g"{name=M}] \arrow[r, bend right=60, "h"'{name=D}] & B \arrow[from=U, to=M, phantom, "\Downarrow\phi"] \arrow[from=M, to=D, phantom, "\Downarrow\psi"] \end{tikzcd}$$



The identity (1-)morphism $id_A : A \to A$ can be depicted as a single-shaded 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 $f : A \to B$:



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 here) 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

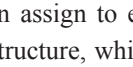
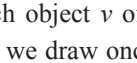
Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

3-categories, which have 3-morphisms between 2-morphisms, 4-categories, which have 4-morphisms between 3-morphisms, and so on. 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. These correspondences are known as the *periodic table* of n-categories [5].

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 recommend [90].

6.3 Inequalities

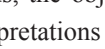
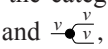
In the same way that we can reason equationally about string diagrams (see Section 2.1), it is also possible to reason with *inequalities*. From the syntactic point of view, the changes are minimal: we can define theory with inequalities in the same way that we defined equational theories (equalities 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 earlier, where morphisms can be ordered by inclusion: for two relations $R, S : X \to Y$, 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.

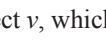
Example 6.5 (Cartesian bicategories). Cartesian bicategories [27] are SMCs in which we can assign to each object v of our chosen signature a monoid and comonoid structure, which we draw once again as  , and  , , respectively. These have to satisfy the following additional axioms:[16]

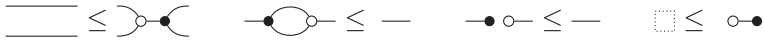
[16] The categorically minded reader will notice that these define an adjunction in the 2-categorical sense, between the comonoid and monoid structures, between and on the one hand, and and on the other.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

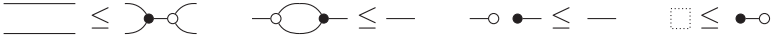
(6.2)

The SMC of monotone relations (Example 5.17) is an example of a cartesian bicategory, with a monoid-comonoid pair given by those of (5.14)–(5.13) for each pre-ordered set $[\![v]\!] = (X, \leq)$. Note however that these, as we have seen, do not form a Frobenius monoid 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 $[\![v]\!]$ in the category of monotone relations for which the interpretations of  , and  , form a Frobenius monoid are just plain sets and monotone relations between them are just ordinary 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 exist two more generating operations on the same object v, which we write as  , , and such that the duals of the inequalities (6.2) hold:



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



One might also wonder how , and , 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 monoid!

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 [69] and that the natural categorical semantics for linear logic takes place in monoidal categories [92], 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 $\parr$. These have non-trivial interplay, axiomatised in the notion of linearly (or weakly) distributive category [33]; if we also want a classical

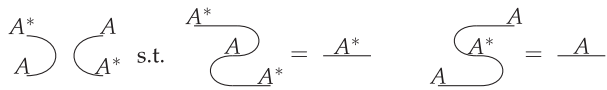
Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

negation, they are related via the usual DeMorgan duality, and the relevant semantics given by $*$-autonomous categories [9].

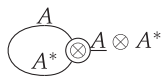
The problem is that our diagrams already use two dimensions: one for the composition operation and one for the monoidal product. How should we deal with two monoidal products? There are different answers. The first (though not the first one historically) is to move one dimension up, from string diagrams in two-dimensional space, to surface diagrams in three-dimensional space. This is what the authors in [49] 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 products[17]

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:



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* [41, 69]. 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 [49]. This projection comes at a cost: not all two-dimensional diagrams are shadows of a three-dimensional surface. Correctness criteria can therefore be

[17] Note 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.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

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), $\otimes = \parr$, 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 [108] deals with string diagrams for SMCs, the central concept of this introduction. It allows the user to specify arbitrary theories in this setting, and apply them as rewrites. String diagram rewriting is implemented as double-pushout 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 programming).
- *DisCoPy* [43] 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.
- `homotopy.io` (the successor of a tool known as Globular [110]) 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` (using a correspondence that we have briefly covered in Section 6.2). 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 in the internal representation.
- `rewalt` [70] is a Python library for higher-dimensional diagrammatic rewriting in which it is possible to build presentations of higher and

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

monoidal categories, among other applications to algebraic topology and algebra. For SMCs, this requires an encoding similar to what was needed with `homotopy.io`. As a bonus, `rewalt` can generate TikZ output to embed diagrams directly into a LaTeX document.

- *Quantomatic* [83] 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 these 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 [113, 115].

Finally, if one is exclusively interested in the typesetting of string diagrams into LaTeXdocuments, we mention the TikZ library, which is especially convenient when paired with TikZit (https://tikzit.github.io/), a GUI editor designed to handle PGF/TikZ pictures.

7 String Diagrams in Science: Some Applications

In the last few years, string diagrams have found application in several 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 of

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

the following examples), 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 to start this section by applications of string diagrams in physics, since a notable ancestor of string diagrams is Penrose's pictorial notation, invented to carry out the complex tensor calculations in the differential geometry of general relativity [101] (see [100] 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 represented as a monoidal product. These generalise and formalise the circuit 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[18] called spiders, each labelled by an angle:

The two colours denote one of two classical observables or measurement bases: the Z (or computational) basis and X (or Hadamard) basis respectively. The angle denotes a phase relative to this basis, that is, a rotation of the Bloch sphere along the axis determined by the chosen observable.

Equationally, the spiders form a special commutative Frobenius monoid and the two colours interact with each other to form a 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 (see Figure 7.1).

[18] Traditionally, green and red, but white and gray (as shown in the diagram here) have been used more recently, for accessibility.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

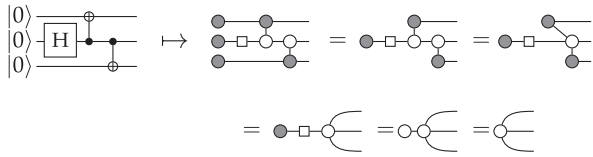


Figure 7.1 The quantum circuit on the left prepares the GHZ state $|000\rangle + |111\rangle$; the ZX-calculus derivation is a diagrammatic proof of this (unlabelled spiders have 0 phase and the square box is a Hadamard gate). See [109, section 5] for similar examples of ZX-calculus proofs.

Reasoning equationally at the level of circuits is 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, 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 [109] 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 [81].

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). Its generators are further from the conventional gates of the classical quantum circuit model, but closely related to linear optical quantum circuits [42, 103]. This last reference builds a calculus that unifies both ZX- and ZW-calculus and is shown complete for arbitrary dimension. 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 [35]. Alternatively, [73] 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 [109]. 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 [31, 42],

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

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 [82]. 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, and so on. 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, that is, 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 SMC $(\mathsf{Rel}, \times)$! One important such subcategory is that of vector spaces over a field and *linear relations* (see Example 5.15) 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 [6], 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 (see Figure 7.2). 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 [55]. 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: a more accurate, but closely related semantics in terms of bi-infinite streams is given in [55], along with a complete equational theory.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

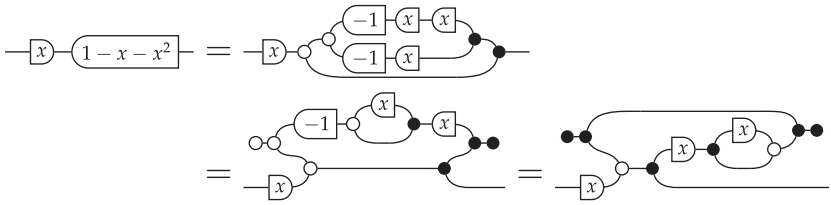


Figure 7.2 Steps of a derivation transforming the Fibonacci generating function $\frac{x}{1-x-x^2}$ into a signal flow graph [21]. The x can be interpreted as derivation in the field $\mathbb{R}(x)$ of rational functions or as a time delay. Note that both the specification (on the left), the signal flow graph which realises it (on the right), and the intermediate steps are all string diagrams of the same calculus, and all the steps apply laws of IH over $\mathbb{R}(x)$.

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 in 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 (see Figure 7.3). 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 previous paragraph, we do have a complete diagrammatic calculus. The case of circuits

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

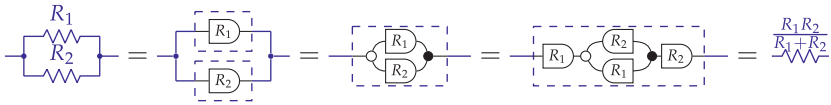


Figure 7.3 Deriving textbook properties of electrical circuits by compiling them to graphical linear algebra. The black diagrams represent the voltage–current pairs enforced by elements of the circuit [11].

with passive components is treated in [4] while the extension to current and voltage sources (in AC regime only) is carried out in [17]. Building on this, [11] develops a convenient calculus that blends both circuit elements and their compilation in IH; see Figure 7.3. The resulting language allows for entirely diagrammatic proofs of standard textbook results of electrical engineering such as the superposition theorem or Thévenin/Norton's theorem. In this work, the ability to reason inductively on circuits as a genuine syntax proves very useful to prove these general theorems.

For *digital circuits*, there are many possible variants of interest to consider, each at its own level of abstraction (and each with its 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 Remark 5.2 for example) – details can be found in the pioneering [85].

More complex cases, involving delays or cycles, are more delicate. 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 [68]. 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 already been done in this direction [66].

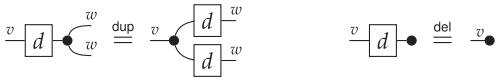
Probability and Statistics

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

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

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:



At the semantic level, this equation simply means that conditional probabilities are measures that normalise to 1. Crucially, not all morphisms satisfy this dup equation, so that categories of Markov kernels are not cartesian; those morphisms that can be copied and do satisfy dup are precisely deterministic kernels, that is, those that, given a 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 [57] 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 [51] which gave a functorial account of Bayesian networks. Around the same time, the authors of [37] proposed a diagrammatic calculus for Bayesian inference. Since then, a surprising amount of probability theory has been recast in this synthetic mould: 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 [60], de Finetti theorem [58], and the ergodic decomposition theorem [94].

The same diagrams (in Markov or CD categories) also allow for a treatment of standard concepts in causal reasoning. In [75], 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; see Figure 7.4. The recent [87] 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

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

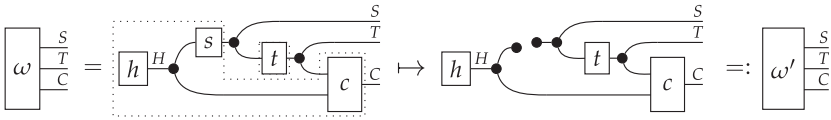


Figure 7.4 In this scenario, from [75], we seek to identify the causal effect of smoking on cancer. String diagrams represent generalised Bayesian networks, encoding causal dependencies between a set of variables. The prior is ω, the joint probability distribution of smoking (variable S), presence of tar in the lungs (variable T), and developing cancer (variable C). A tobacco company contends that, even though there is a statistical correlation between S and C, there might be some confounding factor H (perhaps genetic) which causes both smoking and cancer (decomposition of ω, on the left). How can we rule out this causal scenario, when direct intervention is impossible? We see that performing a 'cut' at S and replacing it with the uniform distribution, as on the right, would remove any confounding influence of H over S. In this case, we can infer from the structure of the diagram that the distribution corresponding to the resulting diagram ω' can be computed from observational data only. If, under ω', a smoker is still more likely to develop cancer, then we have demonstrated that there is a causal relationship between S and C.

graphical model of machine learning. Categorical approaches to these structures are fairly recent and so far have mainly focussed on providing an abstract account of the gradient-based learning process [39, 56, 63]. See, for example, [107] for an overview. Use of string diagrams to describe the network structure only cursorily appears in [56]. String diagrams are heavily used in [39] 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 [112] and polynomial circuits [114], 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 [32] and Lafont's algebraic presentation of Boolean circuits [85]. 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 [56] for feedforward neural networks (see also the diagrammatic presentation of piecewise-linear relations found in [10], which may be used to model networks with ReLu activation units). Another important research thread concerns automatic differentiation (AD): the work [3] 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

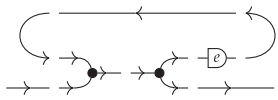
Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

formalisms, which in traditional AD are introduced via compilation. Finally, we recommend the webpage [62] to the interested reader, 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 as 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 [102]. 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, which come together to form a feedback loop:



It should be noted that a similar insight was already present in the work of Ştefănescu [40] who studied the algebraic properties of traced monoidal categories with several additional axioms in order to capture the properties of automata, flowchart schemes, Petri nets, data-flow networks, and more.

Databases and Logic

As we have discussed earlier 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 properties, such as NP-completeness (and thus, decidability) of query inclusion. Moreover, it admits a flexible diagrammatic language, which is exactly that of cartesian bicategories

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

(Example 6.5). 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 into string diagrams). The precise connection with conventional conjunctive query languages is worked out in [18]. More recently, the authors of [72] have generalised these diagrams to all of (classical) predicate logic. This requires adding boxes that represent negation to the diagrams of [18, 27]. Remarkably, these give a categorical account of a graphical notation for first-order logic invented by the American philosopher C. S. Peirce in a series of manuscripts [99] dating as far back as the nineteenth century!

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 run a given program on an input state of the machine. These simple requirements, with the ability to copy and delete data, are what [97] 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 is [98].

Concurrency Theory

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 states (places) or shared actions (transitions). The former was developed in [7], 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 nets). This simple change also changes the equational theory dramatically.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

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 models. The semantics obtained in this way is called *distributional*. On the syntax side, 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, first proposed in [30], is premised on a formal correspondence between grammatical structure and distributional semantics: both fit into monoidal categories! We have seen 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, certain symmetric monoidal theories can model grammatical features of language whose distributional semantics is less clear. Relative pronouns, for example, have been successfully interpreted as certain Frobenius algebras [104, 105], allowing equational reasoning about the meaning of sentences that contain them. String diagrams also reveal that two a priori distinct areas of scientific enquiry can share some formal structure.

Game Theory

In classical game theory, games and their various solution concepts are usually studied monolithically. Remarkably, a compositional approach was shown to be possible in [64]. 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 an SMC with an interesting diagrammatic

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

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.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

Appendix

Category Theory: The Bare Minimum

In this appendix, we recall the most basic notions of category theory: category, functor, natural transformation, adjunctions. This very brief recap is intended as reference material for some of the explanations provided in the main text. For a more pedagogical treatment, the reader should turn to an introductory textbook on the topic, such as [86].

Definition A.1. *A* category C *consists of*

- *a collection of objects;*
- *a collection of morphisms such that every morphism f of* C *has a unique object x called its domain, and a unique object y called its codomain, which we then write $f : x \to y$;*
- *for every pair of morphisms $f : x \to y$ and $g : y \to z$, a morphism $f ; g : x \to z$ which we call the* composition *of f and g;*
- *for every object x, a morphism* $\mathsf{id} : x \to x$ *which we call the* identity *on x;*
- *such that*
  1. *composition is associative, that is,*

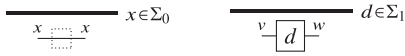
$$f ; (g ; h) = (f ; g) ; h$$

  2. *identities are the (two-sided) unit for composition, that is,*

$$f ; \mathsf{id}_y = f = \mathsf{id}_x ; f$$

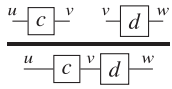
Remark A.2. *The order of composition in the preceding definition is chosen to adhere to the diagrammatic order of composition, from left to right. It is common to see the reverse-order operation, $g \circ f = f ; g$, in particular when dealing with maps between sets.*

Remark A.3. *In the main text, the first categorical structure in which we consider string diagrams are (symmetric) monoidal categories (Definition 2.6). However, plain categories already accommodate a representation of their morphisms as string diagrams, albeit of a simpler kind. Just as in the monoidal case, one may use wires to depict (the identity on) each object, and boxes for each morphism:*

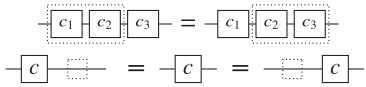


Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

Sequential *composition of boxes with matching wires in the middle is the only allowed operation:*



The diagrammatic notation has the benefit of absorbing the associativity and unitality laws of Definition A.1:



The appropriate notion of structure-preserving mapping between categories is called a functor.

Definition A.4. *Given two categories* C *and* D*, a* functor $F : \mathsf{C} \to \mathsf{D}$ *is a map from the objects of* C *to those of* D*, and a map from the morphisms of* C *to those of* D *that preserves composition and identities, that is,*

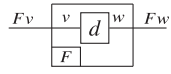
$$F(f\,;\,g) = Ff\,;\,Fg \qquad F(\mathsf{id}_x) = \mathsf{id}_{Fx}$$

Clearly, functors can also be composed like ordinary maps. With functors as morphisms, categories themselves form a category. A functor is called *faithful* if it is injective on morphisms of the same type, that is, if $Ff = Fg$ implies that $f = g$; it is *full* if it is surjective on morphisms of the same type, that is, for any g in D, there exists f in C such that $Ff = g$.

There is also a notion of mapping between functors, which we now recall.

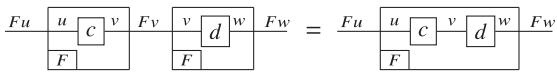
Definition A.5. *Given two functors* $F : \mathsf{C} \to \mathsf{D}$ *and* $G : \mathsf{C} \to \mathsf{D}$*, a* natural transformation $\eta : F \Rightarrow G$ *is a family of morphisms* $\eta_x : Fx \to Gx$ *indexed by the objects of* C*, such that* $\eta_x\,;\,Gf = Ff\,;\,\eta_y$ *for every morphism* $f : x \to y$*.*

Remark A.6. *Functors can also be represented pictorially, via* functorial boxes *[91]. Plainly, for a functor* $F : \mathsf{C} \to \mathsf{D}$*, they are F-labelled boxes that frame diagrams*



Diagrams inside the box live in category C*, and those outside in* D*. To represent functors, functorial boxes have to be functorial! This means that they satisfy the following equality:*

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715



This is easily seen to be the translation of the first equality in Definition A.4. Note that the preservation of identities required by Definition A.4 is already a diagrammatic tautology, as it is absorbed by the diagrammatic notation.

We do not cover these here in detail, though we refer to their use in representing diagrams for closed monoidal categories, in Section 4.2.9.

Recall that an isomorphism is a morphism $f : x \to y$ which has an inverse: a morphism $g : y \to x$ such that $f ; g = \mathsf{id}_x$ and $g ; f = \mathsf{id}_y$. We then say that x is isomorphic to y. The notion of isomorphism makes sense for functors and categories too. The inverse of a functor $F : \mathsf{C} \to \mathsf{D}$ is a functor $G : \mathsf{D} \to \mathsf{C}$ such that $F ; G = \mathsf{id}_{\mathsf{C}}$ and $G ; F = \mathsf{id}_{\mathsf{D}}$. However, it is sometimes too restrictive to ask for isomorphisms between categories. Indeed, there are categories that we would like to identify, which are not isomorphic. The more general notion of equivalence of categories is often more appropriate.

Definition A.7. *A functor $F : \mathsf{C} \to \mathsf{D}$ is an* equivalence *of categories if there exists a functor $G : \mathsf{D} \to \mathsf{C}$ and two natural transformations $\epsilon : FG \Rightarrow \mathsf{id}_{\mathsf{C}}$ and $\eta : \mathsf{id}_{\mathsf{D}} \Rightarrow GF$ whose components are isomorphisms.*

Remark A.8. *In this Element, we make use of* monoidal equivalences *of monoidal categories. The definition is the obvious modification of Definition A.7: it is a monoidal functor $F : \mathsf{C} \to \mathsf{D}$ (*see *Definition 2.8) for which there exists a monoidal functor G satisfying the conditions of the preceding definition. The same goes for* symmetric *monoidal categories.*

We can even weaken further the notion of equivalence to that of *adjunction*. Instead of requiring equalities $F ; G = \mathsf{id}_{\mathsf{C}}$ and $G ; F = \mathsf{id}_{\mathsf{D}}$, we can consider functors for which we have natural transformations $G ; F \Rightarrow \mathsf{id}_{\mathsf{C}}$ and $\mathsf{id}_{\mathsf{D}} \Rightarrow F ; G$ that satisfy some conditions that we now recall.

Definition A.9. *An* adjunction *between two categories consists of a pair of functors $F : \mathsf{C} \to \mathsf{D}$ and $G : \mathsf{D} \to \mathsf{C}$ and two natural transformations $\epsilon : FG \Rightarrow \mathsf{id}_{\mathsf{C}}$ (the* unit*) and $\eta : \mathsf{id}_{\mathsf{D}} \Rightarrow GF$ (the* counit*) such that*

$$F\eta_x ; \epsilon_{Fx} = \mathsf{id}_{Fx} \quad \textit{and} \quad \eta_{Ga} ; G\epsilon_a = \mathsf{id}_{Ga}$$

We call F the left adjoint *and G the* right adjoint.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

References

[1] Samson Abramsky. Retracing some paths in process algebra. In Ugo Montanari and Vladimiro Sassone (eds.), *CONCUR'96 Proceedings*, pages 1–17. Springer, 2005.

[2] Mario Alvarez-Picallo, Dan R. Ghica, David Sprunger, and Fabio Zanasi. Rewriting for monoidal closed categories. In Amy P. Felty (ed.), *FSCD'22 Proceedings*, volume 228 of *LIPIcs*, pages 29:1–29:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

[3] Mario Alvarez-Picallo, Dan R. Ghica, David Sprunger, and Fabio Zanasi. Functorial string diagrams for reverse-mode automatic differentiation. In Bartek Klin and Elaine Pimentel (eds.), *CSL'23 Proceedings*, volume 252 of *LIPIcs*, pages 6:1–6:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

[4] John C. Baez, Brandon Coya, and Franciscus Rebro. Props in network theory. *Theory and Applications of Categories*, 33(25):727–783, 2018.

[5] John C. Baez and James Dolan. Higher-dimensional algebra and topological quantum field theory. *Journal of Mathematical Physics*, 36(11):6073–6105, 1995.

[6] John C. Baez and Jason Erbele. Categories in control. *Theory and Applications of Categories*, 30:836–881, 2015.

[7] John C. Baez and Jade Master. Open petri nets. *Mathematical Structures in Computer Science*, 30(3):314–341, 2020.

[8] Hendrik Pieter Barendregt (ed.). *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of Studies in Logic and the Foundations of Mathematics. North-Holland, 1985.

[9] Michael Barr. $*$-*Autonomous Categories*, volume 752, Springer Lecture Notes in Mathematics. Springer, 2006.

[10] Guillaume Boisseau and Robin Piedeleu. Graphical piecewise-linear algebra. In Patricia Bouyer and Lutz Schröder (eds.), *FOSSACS'22 (ETAPS) Proceedings*, pages 101–119. Springer International Publishing, 2022.

[11] Guillaume Boisseau and Paweł Sobociński. String diagrammatic electrical circuit theory. In K. Kishida (Ed.), *Fourth International Conference on Applied Category Theory (ACT 2021)*. EPTCS 372, pp. 178–191. DOI: https://doi.org/10.4204/EPTCS.372.13, 2022.

[12] Joe Bolt, Jules Hedges, and Philipp Zahn. Bayesian open games. *Compositionality*, October 4, Volume 5. DOI: https://doi.org/10.32408/compositionality-5-9, 2023.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

[13] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Paweł Sobociński, and Fabio Zanasi. Rewriting modulo symmetric monoidal structure. In *LICS'16 Proceedings*, https://dl.acm.org/doi/proceedings/10.1145/2933575, pages 710–719. Institute of Electrical and Electronics Engineers, 2016.

[14] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. String diagram rewrite theory I: Rewriting with Frobenius structure. *Journal of the ACM*, 69(2):14:1–14:58, 2022.

[15] Filippo Bonchi, Joshua Holland, Dusko Pavlovic, and Paweł Sobociński. Refinement for signal flow graphs. In Roland Meyer and Uwe Nestmann (eds.), *CONCUR'17 Proceedings*, pages 24:1–24:16, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 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, Association for Computing Machinery, 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 Sobociński. Graphical conjunctive queries. In Dan R. Ghica and Achim Jung (eds.), *CSL'18 Proceedings*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

[19] Filippo Bonchi, Paweł Sobociński, and Fabio Zanasi. Interacting bialgebras are Frobenius. In Anca Muscholl (ed.), *FOSSACS'14 Proceedings*, volume 8412 of *LNCS*, pages 351–365. Springer, 2014.

[20] Filippo Bonchi, Paweł Sobociński, and Fabio Zanasi. Full abstraction for signal flow graphs. *ACM SIGPLAN Notices*, 50(1):515–526, 2015.

[21] Filippo Bonchi, Paweł Sobociński, and Fabio Zanasi. The calculus of signal flow diagrams I: Linear relations on streams. *Information and Computation*, 252:2–29, 2017.

[22] Filippo Bonchi, Pawel Sobocinski, and Fabio Zanasi. Interacting Hopf algebras. *Journal of Pure and Applied Algebra*, 221(1):144–184, 2017.

[23] Filippo Bonchi, Paweł Sobociński, and Fabio Zanasi. Deconstructing Lawvere with distributive laws. *Journal of Logical and Algebraic Methods in Programming*, 95:128–146, 2018.

[24] Roberto Bruni and Fabio Gadducci. Some algebraic laws for spans (and their connections with multirelations). *Electronic Notes in Theoretical Computer Science*, 44(3): 175–193, 2001.

[25] Roberto Bruni, Hernán Melgratti, and Ugo Montanari. Connector algebras, Petri nets, and BIP. In Edmund Clarke, Irina Virbitskaite, and Andrei Voronkov (eds.), *Perspectives of Systems Informatics*. PSI 2011.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

Lecture Notes in Computer Science, vol. 7162, 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. *Logical Methods in Computer Science*, 9(16): n. pag., 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. Institute of Electrical and Electronics Engineers, 2021.

[29] Eugenia Cheng. Iterated distributive laws. *Mathematical Proceedings of the Cambridge Philosophical Society*, 150(3):459–487, 2011.

[30] Stephen Clark, Bob Coecke, and Mehrnoosh Sadrzadeh. A compositional distributional model of meaning. In Peter D. Bruza, William F. Lawless, Keith van Rijsbergen, Donald A. Sofge, and Bob Coecke (eds.), *Proceedings of the Second Quantum Interaction Symposium (QI-2008)*, pages 133–140. College Publications, 2008.

[31] Alexandre Clément, Nicolas Heurtel, Shane Mansfield, Simon Perdrix, and Benoît Valiron. LOv-calculus: A graphical language for linear optical quantum circuits. In Stefan Szeider, Robert Ganian, and Alexandra Silva (eds.), *MFCS'22 Proceedings*, volume 241 of *LIPIcs*, pages 35:1–35:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.

[32] J. Robin B. Cockett, Geoff S. H. Cruttwell, Jonathan Gallagher, Jean-Simon Pacaud Lemay, Benjamin MacAdam, Gordon D. Plotkin, and Dorette Pronk. Reverse derivative categories. In Maribel Fernández and Anca Muscholl (eds.), *CSL'20 Proceedings*, volume 152 of *LIPIcs*, pages 18:1–18:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.

[33] J. Robin B. Cockett and Robert A. G. Seely. Weakly distributive categories. *Journal of Pure and Applied Algebra*, 114(2):133–173, 1997.

[34] Bob Coecke and Ross Duncan. Interacting quantum observables. In Luca Aceto, Ivan B. Damgärd, and Leslie A. Goldberg (eds.), *ICALP'08 Proceedings, Part II*, pages 298–310. Springer, 2008.

[35] Bob Coecke and Aleks Kissinger. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press, 2017.

[36] Bob Coecke, Dusko Pavlovic, and Jamie Vicary. A new description of orthogonal bases. *Mathematical Structures in Computer Science*, 23(3):557–567, 2012.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

[37] Bob Coecke and Robert W. Spekkens. Picturing classical and quantum Bayesian inference. *Synthese*, 186:651–696, 2012.

[38] Brandon Coya and Brendan Fong. Corelations are the prop for extraspecial commutative Frobenius monoids. *Theory and Applications of Categories*, 32(11):380–395, 2017.

[39] Geoffrey S. H. Cruttwell, Bruno Gavranović, Neil Ghani, Paul W. Wilson, and Fabio Zanasi. Categorical foundations of gradient-based learning. In Ilya Sergey (ed.), *ESOP'22*, volume 13240 of *LNCS*, pages 1–28. Springer, 2022.

[40] Gheorghe Ştefănescu. *Network Algebra*. Discrete Mathematics and Theoretical Computer Science. Springer, 2000.

[41] Vincent Danos and Laurent Regnier. The structure of multiplicatives. *Archive for Mathematical Logic*, 28(3):181–203, 1989.

[42] Giovanni De Felice and Bob Coecke. Quantum linear optics via string diagrams. *arXiv:2204.12985*, 2022.

[43] Giovanni de Felice, Alexis Toumi, and Bob Coecke. DisCoPy: Monoidal categories in Python. *Electronic Proceedings in Theoretical Computer Science*, 333:183–197, 2021.

[44] Elena Di Lavore, Giovanni de Felice, and Mario Román. Monoidal streams for dataflow programming. In Christel Baier (ed.), *LICS'22 Proceedings*, pages 1–14, 2022.

[45] Elena Di Lavore, Alessandro Gianola, Mario Román, Nicoletta Sabadini, and Paweł Sobociński. A canonical algebra of open transition systems. In Gwen Salaün and Anton Wijs (eds.), *FACS'21 Proceedings 17*, pages 63–81. Springer, 2021.

[46] Luca Dixon and Aleks Kissinger. Open-graphs and monoidal theories. *Mathematical Structures in Computer Science*, 23(2):308–359, 2013.

[47] Ross Duncan and Kevin Dunne. Interacting Frobenius algebras are Hopf. In *LICS'16*, pages 535–544, Institute of Electrical and Electronics Engineers, 2016.

[48] 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.

[49] Lawrence Dunn and Jamie Vicary. Coherence for Frobenius pseudomonoids and the geometry of linear proofs. *Logical Methods in Computer Science*, 15, 2019.

[50] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph-grammars: An algebraic approach. In *SWAT'73 Proceedings*, pages 167–180. Institute of Electrical and Electronics Engineers, 1973.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

[51] Brendan Fong. Causal theories: A categorical perspective on Bayesian networks. Master's thesis, University of Oxford, 2012. arXiv: 1301.6201.

[52] Brendan Fong. Decorated cospans. *Theory and Applications of Categories*, 30(33):1096–1120, 2015.

[53] Brendan Fong. *The Algebra of Open and Interconnected Systems*. PhD thesis, University of Oxford, 2016. arXiv:1609.05382.

[54] Brendan Fong. Decorated corelations. *Theory & Applications of Categories*, 33, 2018.

[55] Brendan Fong, Paweł Sobociński, and Paolo Rapisarda. A categorical approach to open and interconnected dynamical systems. In *LICS'16 Proceedings*, pages 495–504, Association for Computing Machinery, 2016.

[56] 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.

[57] Tobias Fritz. A synthetic approach to Markov kernels, conditional independence and theorems on sufficient statistics. *Advances in Mathematics*, 370:107239, 2020.

[58] Tobias Fritz, Tomáš Gonda, and Paolo Perrone. De Finetti's theorem in categorical probability. *Journal of Stochastic Analysis*, 2(4):6, 2021.

[59] Tobias Fritz and Wendong Liang. Free gs-monoidal categories and free Markov categories. *Applied Categorical Structures*, 31(2):21, 2023.

[60] Tobias Fritz and Eigil Fjeldgren Rischel. The zero-one laws of Kolmogorov and Hewitt–Savage in categorical probability. *Compositionality*, 2:3, 2020.

[61] Fabio Gadducci and Reiko Heckel. An inductive view of graph transformation. In Francesco Parisi Presicce (ed.), *WADT'97 Proceedings*, pages 223–237, Springer, 1997.

[62] Bruno Gavranović. Category theory ∩ machine learning. https://github.com/bgavran/Category_Theory_Machine_Learning. Accessed: 9 June 2023.

[63] Bruno Gavranović. Compositional deep learning. *arXiv:1907.08292*, 2019.

[64] Neil Ghani, Jules Hedges, Viktor Winschel, and Philipp Zahn. Compositional game theory. In *LICS'18 Proceedings*, pages 472–481, Association for Computing Machinery, 2018.

[65] Dan Ghica and Fabio Zanasi. String diagrams for λ-calculi and functional computation. *arXiv:2305.18945*, 2023.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

[66] Dan R. Ghica. Diagrammatic reasoning for delay-insensitive asynchronous circuits. In Bob Coecke, Luke Ong, and Prakash Panangaden (eds.), *Computation, Logic, Games, and Quantum Foundations: The Many Facets of Samson Abramsky*, pages 52–68. Springer, 2013.

[67] Dan R. Ghica and George Kaye. Rewriting modulo traced comonoid structure. *arXiv:2302.09631*, 2023.

[68] Dan R. Ghica, George Kaye, and David Sprunger. Full abstraction for digital circuits. *arXiv:2201.10456*, 2022.

[69] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[70] Amar Hadzihasanovic and Diana Kessler. Data structures for topologically sound higher-dimensional diagram rewriting. In *ACT'22 Proceedings. arXiv:2209.09509*, 2022.

[71] Masahito Hasegawa. Recursion from cyclic sharing: Traced monoidal categories and models of cyclic lambda calculi. In Philippe de Groote and J. Roger Hindley (eds.), *TLCA'97 Proceedings*, pages 196–213. Springer, 1997.

[72] Nathan Haydon and Paweł Sobociński. Compositional diagrammatic first-order logic. In Ahti-Veikko Pietarinen, Peter Chapman, Leonie Bosveld-de Smet, Valeria Giardino, James Corter, and Sven Linker (eds.), *Diagrams'20 Proceedings*, pages 402–418. Springer, 2020.

[73] Chris Heunen and Jamie Vicary. *Categories for Quantum Theory: An Introduction*. Oxford University Press, 2019.

[74] Martin Hyland and John Power. The category theoretic understanding of universal algebra: Lawvere theories and monads. In Luca Cardelli, Marco Fiore, and Glynn Winskel (eds.), *Computation, Meaning, and Logic: Articles Dedicated to Gordon Plotkin*, volume 172 of *Electronic Notes in Theoretical Computer Science*, pages 437–458. Elsevier, 2007.

[75] 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.

[76] André Joyal and Ross Street. Braided monoidal categories. *Mathematics Reports*, 86008, 1986.

[77] André Joyal and Ross Street. The geometry of tensor calculus, I. *Advances in Mathematics*, 88(1):55–112, 1991.

[78] Piergiulio Katis, Nicoletta Sabadini, and Robert F. C. Walters. Feedback, trace and fixed-point semantics. *RAIRO-Theoretical Informatics and Applications*, 36(2):181–194, 2002.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

[79] Gregory Maxwell Kelly. Many-variable functorial calculus. I. In Gregory M. Kelly, Miguel L. Laplaza, Geoffrey Lewis, and Saunders Mac Lane (eds.), *Coherence in Categories*, pages 66–105. Springer, 1972.

[80] Gregory Maxwell Kelly and Miguel L. Laplaza. Coherence for compact closed categories. *Journal of Pure and Applied Algebra*, 19:193–213, 1980.

[81] Andrey Boris Khesin, Jonathan Z. Lu, and Peter W. Shor. Graphical quantum Clifford-encoder compilers from the ZX calculus. *arXiv: 2301.02356*, 2023.

[82] Aleks Kissinger and John van de Wetering. PyZX: Large scale automated diagrammatic reasoning. *arXiv:1904.04735*, 2019.

[83] Aleks Kissinger and Vladimir Zamdzhiev. Quantomatic: A proof assistant for diagrammatic reasoning. In Amy P. Felty and Aart Middeldorp (eds.), *CADE-25 Proceedings*, pages 326–336. Springer, 2015.

[84] Stephen Lack. Composing PROPs. *Theory and Application of Categories*, 13(9):147–163, 2004.

[85] Yves Lafont. Towards an algebraic theory of Boolean circuits. *Journal of Pure and Applied Algebra*, 184(2–3):257–310, 2003.

[86] Tom Leinster. *Basic Category Theory*, volume 143. Cambridge University Press, 2014.

[87] Robin Lorenz and Sean Tull. Causal models in string diagrams. *arXiv:2304.07638*, 2023.

[88] Saunders Mac Lane. Categorical algebra. *Bulletin of the American Mathematical Society*, 71:40–106, 1965.

[89] Saunders Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics, Vol. 5. Springer, 1971.

[90] Daniel Marsden. Category theory using string diagrams. *arXiv:1401. 7220*, 2014.

[91] Paul-André Melliès. Functorial boxes in string diagrams. In Zoltán Ésik (ed.), *International Workshop on Computer Science Logic*, pages 1–30. Springer, 2006.

[92] Paul-André Mellies. Categorical semantics of linear logic. *Panoramas et Syntheses*, 27:15–215, 2009.

[93] Aleksandar Milosavljevic and Fabio Zanasi. String diagram rewriting modulo commutative monoid structure. To appear in *CALCO'23 Proceedings, arXiv:2204.04274*, 2023.

[94] Sean Moss and Paolo Perrone. A category-theoretic proof of the ergodic decomposition theorem. *arXiv:2207.07353*, 2022.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

[95] Koko Muroya and Dan R. Ghica. The dynamic geometry of interaction machine: A call-by-need graph rewriter. *CSL'17 Proceedings*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. DOI: https://doi.org/10.4230/LIPIcs.CSL.2017.32.

[96] Dusko Pavlovic. Quantum and classical structures in nondeterministic computation. In Peter Bruza, Donald Sofge, William Lawless, Keith van Rijsbergen, and Matthias Klusch (eds.), *International Symposium on Quantum Interaction*, pages 143–157. Springer, 2009.

[97] Dusko Pavlovic. Monoidal computer I: Basic computability by string diagrams. *Information and Computation*, 226:94–116, 2013.

[98] Dusko Pavlovic. Categorical computability in monoidal computer: Programs as diagrams. *arXiv:2208.03817*, 2022.

[99] Charles Sanders Peirce. *Collected Papers of Charles Sanders Peirce*, volume 4. Harvard University Press, 1974.

[100] Roger Penrose. Applications of negative dimension tensors. In Dominic Welsh, editor, *Combinatorial Mathematics and Its Applications*, pages 221–244. Academic Press, 1971.

[101] Roger Penrose. Applications of negative dimensional tensors. *Combinatorial Mathematics and Its Applications*, 1:221–244, 1971.

[102] Robin Piedeleu and Fabio Zanasi. A finite axiomatisation of finite-state automata using string diagrams. *Logical Methods in Computer Science*, 19, 2023.

[103] Boldizsár Poór, Quanlong Wang, Razin A. Shaikh, Lia Yeh, Richie Yeung, and Bob Coecke. Completeness for arbitrary finite dimensions of ZXW-calculus, a unifying calculus. *arXiv:2302.12135*, 2023.

[104] 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.

[105] 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.

[106] Peter Selinger. A survey of graphical languages for monoidal categories. *Springer Lecture Notes in Physics*, 13(813):289–355, 2011.

[107] Dan Shiebler, Bruno Gavranović, and Paul Wilson. Category theory in machine learning. *arXiv:2106.07032*, 2021.

[108] Paweł Sobociński, Paul W. Wilson, and Fabio Zanasi. CARTOGRAPHER: A tool for string diagrammatic reasoning. In *CALCO'19 Proceedings*, page 25. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

[109] John van de Wetering. ZX-calculus for the working quantum computer scientist. *arXiv:2012.13966*, 2020.

[110] Jamie Vicary, Aleks Kissinger, and Krzysztof Bar. Globular: An online proof assistant for higher-dimensional rewriting. *Logical Methods in Computer Science*, 14. *arXiv:1612.01093v1*, 2018.

[111] Paul W. Wilson, Dan R. Ghica, and Fabio Zanasi. String diagrams for non-strict monoidal categories. In Bartek Klin and Elaine Pimentel (eds.), *CSL*, volume 252 of *LIPIcs*, pages 37:1–37:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

[112] Paul W. Wilson and Fabio Zanasi. Reverse derivative ascent: A categorical approach to learning Boolean circuits. In David I. Spivak and Jamie Vicary (eds.), *ACT'20 Proceedings*, volume 333 of *EPTCS*, pages 247–260, 2020.

[113] 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. *arXiv:2105.09257*, 2021.

[114] Paul W. Wilson and Fabio Zanasi. Categories of differentiable polynomial circuits for machine learning. In Nicolas Behr and Daniel Strüber (eds.), *ICGT*, volume 13349 of *Lecture Notes in Computer Science*, pages 77–93. Springer, 2022.

[115] Paul W. Wilson and Fabio Zanasi. Data-parallel algorithms for string diagrams. *arXiv:2305.01041*, 2023.

[116] Fabio Zanasi. *Interacting Hopf Algebras: The Theory of Linear Systems*. PhD thesis, Ecole Normale Supérieure de Lyon, 2015.

[117] Fabio Zanasi. The algebra of partial equivalence relations. *Electronic Notes in Theoretical Computer Science*, 325:313–333, 2016.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

Acknowledgements

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC) [grant number. EP/V002376/1]. This funding also made it possible for this Element to be published open access, making the digital version freely available for anyone to read and reuse under a Creative Commons licence.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

Cambridge Elements

Applied Category Theory

Bob Coecke
Cambridge Quantum Ltd

Bob Coecke is Chief Scientist at Cambridge Quantum Ltd. and Emeritus Professor at Wolfson College, University of Oxford. His pioneering research includes categorical quantum mechanics, ZX-calculus, quantum causality, resource theories, and quantum natural language processing. Other interests include cognitive architectures and diagrams in education. Most of his work uses the language of tensor categories and their diagrammatic representations. He co-authored the book *Picturing Quantum Processes. A First Course in Quantum Theory and Diagrammatic Reasoning*. He is considered as one of the fathers of the field of applied category theory, and a co-founder of the journal *Compositionality*.

Joshua Tan
University of Oxford

Joshua Tan is a doctoral student at the University of Oxford and a Practitioner Fellow at Stanford University. He is the executive director of the Metagovernance Project and an executive editor of the journal *Compositionality*. He works on applications of category theory and sheaf theory to theoretical machine learning and to the design of complex, multi-agent systems.

About the Series

Elements in Applied Category Theory features Elements intended both for mathematicians familiar with category theory and seeking elegant, graduate-level introductions to other fields in the language of categories, and for subject-matter experts outside of pure mathematics interested in applications of category theory to their field.

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715

Cambridge Elements

Applied Category Theory

Elements in the Series

Theoretical Computer Science for the Working Category Theorist
Noson S. Yanofsky

An Introduction to String Diagrams for Computer Scientists
Robin Piedeleu and Fabio Zanasi

A full series listing is available at: www.cambridge.org/EACT

Downloaded from https://www.cambridge.org/core. IP address: 169.224.197.5, on 28 Oct 2026 at 04:55:06, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/9781009625715