1

Physics, Topology, Logic and Computation: A Rosetta Stone

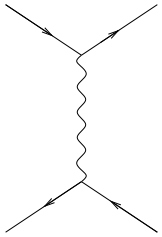
John Baez[1]
Michael Stay[2]

[1]Department of Mathematics, University of California, Riverside CA 92521, USA
[2]Google, 1600 Amphitheatre Pkwy, Mountain View CA 94043, USA

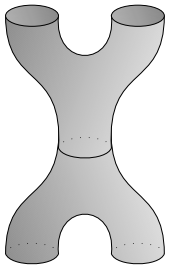
1.1 Introduction

Category theory is a very general formalism, but there is a certain special way that physicists use categories which turns out to have close analogues in topology, logic and computation. A category has *objects* and *morphisms*, which represent *things* and *ways to go between things*. In physics, the objects are often *physical systems*, and the morphisms are *processes* turning a state of one physical system into a state of another system — perhaps the same one. In quantum physics we often formalize this by taking *Hilbert spaces* as objects, and *linear operators* as morphisms.

Sometime around 1949, Feynman [54] realized that in quantum field theory it is useful to draw linear operators as diagrams:



This lets us reason with them pictorially. We can warp a picture without changing the operator it stands for: all that matters is the topology, not the geometry. In the 1970s, Penrose realized that generalizations of Feynman diagrams arise throughout quantum theory, and might even lead to revisions in our understanding of spacetime [75]. In the 1980s, it became clear that underlying these diagrams is a powerful analogy between quantum physics and topology! Namely, a linear operator behaves very much like a 'cobordism' — that is, an n-dimensional manifold going between manifolds of one dimension less:



String theory exploits this analogy by replacing the Feynman diagrams of ordinary quantum field theory with 2-dimensional cobordisms, which represent the worldsheets traced out by strings with the passage of time. The analogy between operators and cobordisms is also important in loop quantum gravity and — most of all — the more purely mathematical discipline of 'topological quantum field theory'.

Meanwhile, quite separately, logicians had begun using categories where the objects represent *propositions* and the morphisms represent *proofs*. The idea is that a proof is a process going from one proposition (the hypothesis) to another (the conclusion). Later, computer scientists started using categories where the objects represent *data types* and the morphisms represent *programs*. They also started using 'flow charts' to describe programs. Abstractly, these are very much like Feynman diagrams!

The logicians and computer scientists were never very far from each other. Indeed, the 'Curry–Howard correspondence' relating proofs to programs has been well-known at least since the early 1970s, with roots stretching back earlier [34, 51]. But, it is only in the 1990s that the logicians and computer scientists bumped into the physicists and topologists. One reason is the rise of interest in quantum cryptography and quantum computation [27]. With this, people begain to think of quantum processes as forms of information processing, and apply ideas from computer science. It was then realized that the loose analogy between between flow charts and Feynman diagrams could be made more precise and powerful with the aid of category theory [3].

By now there is an extensive network of interlocking analogies between physics, topology, logic and computer science. They suggest that research in the area of common overlap is actually trying to build a new science: *a general science of systems and processes*. Building this science will be very difficult. There are good reasons for this, but also bad ones. One bad reason is that different fields use different terminology and notation.

The original Rosetta Stone, created in 196 BC, contains versions of the same text in three languages: demotic Egyptian, hieroglyphic script and classical Greek. Its rediscovery by Napoleon's soldiers let modern Egyptologists decipher the hieroglyphs. Eventually this led to a vast increase in our understanding of Egyptian culture.

At present, the deductive systems in mathematical logic look like hieroglyphs to most physicists. Similarly, quantum field theory is Greek to most computer scientists, and so on. So, there is a need for a new Rosetta Stone to aid researchers attempting to translate between fields. Table 1.1 shows our guess as to what this Rosetta Stone might look like.

Category Theory	Physics	Topology	Logic	Computation
object	system	manifold	proposition	data type
morphism	process	cobordism	proof	program

Table 1.1. The Rosetta Stone (pocket version)

The rest of this paper expands on this table by comparing how categories are used in physics, topology, logic, and computation. Unfortunately, these different fields focus on slightly different kinds of categories. Though most physicists don't know it, quantum physics has long made use of 'compact symmetric monoidal categories'. Knot theory uses 'compact braided monoidal categories', which are slightly more general. However, it became clear in the 1990's that these more general gadgets are useful in physics too. Logic and computer science used to focus on 'cartesian closed categories' — where 'cartesian' can be seen, roughly, as an antonym of 'quantum'. However, thanks to work on linear logic and quantum computation, some logicians and computer scientists have dropped their insistence on cartesianness: now they study more general sorts of 'closed symmetric monoidal categories'.

In Section 1.2 we explain these concepts, how they illuminate the analogy between physics and topology, and how to work with them using string diagrams. We assume no prior knowledge of category theory, only a willingness to learn some. In Section 1.3 we explain how closed symmetric monoidal categories correspond to a small fragment of ordinary propositional logic, which also happens to be a fragment of Girard's 'linear logic' [44]. In Section 1.4 we explain how closed symmetric monoidal categories correspond to a simple model of computation. Each of these sections starts with some background material. In Section 1.5, we conclude by presenting a larger version of the Rosetta Stone.

Our treatment of all four subjects — physics, topology, logic and computation — is bound to seem sketchy, narrowly focused and idiosyncratic to practitioners of these subjects. Our excuse is that we wish to emphasize certain analogies while saying no more than absolutely necessary. To make up for this, we include many references for those who wish to dig deeper.

1.2 The Analogy Between Physics and Topology

1.2.1 Background

Currently our best theories of physics are general relativity and the Standard Model of particle physics. The first describes gravity without taking quantum theory into account; the second describes all the other forces taking quantum theory into account, but ignores gravity. So, our world-view is deeply schizophrenic. The field where physicists struggle to solve this problem is called *quantum gravity*, since it is widely believed that the solution requires treating gravity in a way that takes quantum theory into account.

Nobody is sure how to do this, but there is a striking similarity between two of the main approaches: string theory and loop quantum gravity. Both rely on the analogy between

Physics	Topology
Hilbert space (system)	$(n-1)$-dimensional manifold (space)
operator between Hilbert spaces (process)	cobordism between $(n-1)$-dimensional manifolds (spacetime)
composition of operators	composition of cobordisms
identity operator	identity cobordism

Table 1.2. Analogy between physics and topology

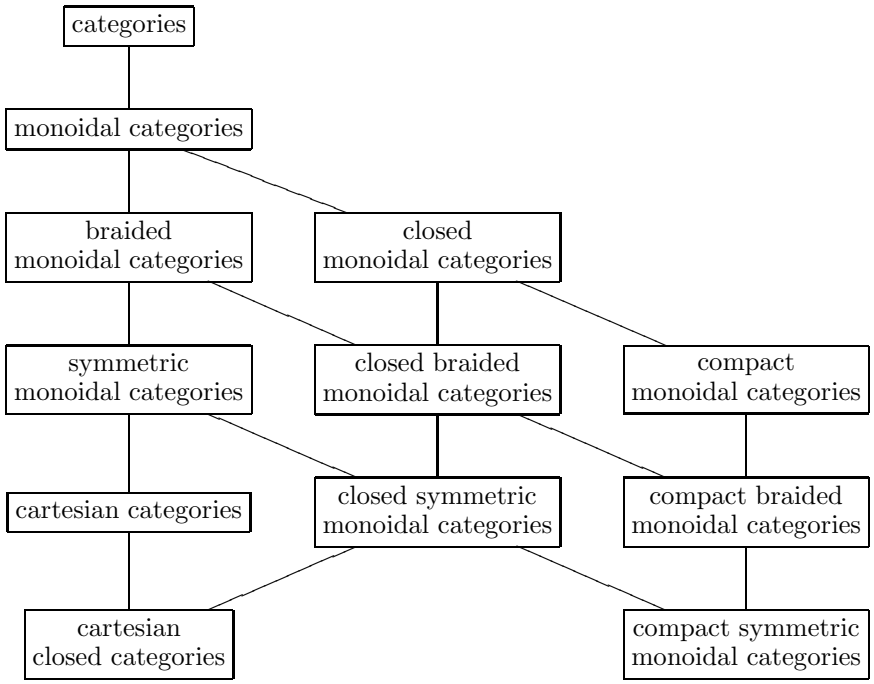
physics and topology shown in Table 1.2. On the left we have a basic ingredient of quantum theory: the category Hilb whose objects are Hilbert spaces, used to describe physical *systems*, and whose morphisms are linear operators, used to describe physical *processes*. On the right we have a basic structure in differential topology: the category nCob. Here the objects are $(n-1)$-dimensional manifolds, used to describe *space*, and whose morphisms are n-dimensional cobordisms, used to describe *spacetime*.

As we shall see, Hilb and nCob share many structural features. Moreover, both are very different from the more familiar category Set, whose objects are sets and whose morphisms are functions. Elsewhere we have argued at great length that this is important for better understanding quantum gravity [8] and even the foundations of quantum theory [9]. The idea is that if Hilb is more like nCob than Set, maybe we should stop thinking of a quantum process as a function from one set of states to another. Instead, maybe we should think of it as resembling a 'spacetime' going between spaces of dimension one less.

This idea sounds strange, but the simplest example is something very practical, used by physicists every day: a Feynman diagram. This is a 1-dimensional graph going between 0-dimensional collections of points, with edges and vertices labelled in certain ways. Feynman diagrams are topological entities, but they describe linear operators. String theory uses 2-dimensional cobordisms equipped with extra structure — string worldsheets — to do a similar job. Loop quantum gravity uses 2d generalizations of Feynman diagrams called 'spin foams' [7]. Topological quantum field theory uses higher-dimensional cobordisms [11]. In each case, processes are described by morphisms in a special sort of category: a 'compact symmetric monoidal category'.

In what follows, we shall not dwell on puzzles from quantum theory or quantum gravity. Instead we take a different tack, simply explaining some basic concepts from category theory and showing how Set, Hilb, nCob and categories of tangles give examples. A recurring theme, however, is that Set is very different from the other examples.

To help the reader safely navigate the sea of jargon, here is a chart of the concepts we shall explain in this section:



The category Set is cartesian closed, while Hilb and nCob are compact symmetric monoidal.

1.2.2 Categories

Category theory was born around 1945, with Eilenberg and Mac Lane [39] defining 'categories', 'functors' between categories, and 'natural transformations' between functors. By now there are many introductions to the subject [33, 69, 72], including some available for free online [18, 47]. Nonetheless, we begin at the beginning:

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

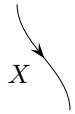
- *a collection of* **objects**, *where if* X *is an object of* C *we write* $X \in C$, *and*
- *for every pair of objects* (X, Y), *a set* $\hom(X, Y)$ *of* **morphisms** *from* X *to* Y. *We call this set* $\hom(X, Y)$ *a* **homset**. *If* $f \in \hom(X, Y)$, *then we write* $f \colon X \to Y$.

such that:

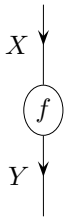
- *for every object* X *there is an* **identity morphism** $1_X \colon X \to X$;

- *morphisms are composable: given* $f\colon X \to Y$ *and* $g\colon Y \to Z$*, there is a* **composite morphism** $gf\colon X \to Z$*; sometimes also written* $g \circ f$*.*
- *an identity morphism is both a* **left and a right unit** *for composition: if* $f\colon X \to Y$*, then* $f1_X = f = 1_Y f$*; and*
- *composition is* **associative***:* $(hg)f = h(gf)$ *whenever either side is well-defined.*

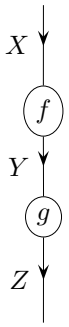
A category is the simplest framework where we can talk about systems (objects) and processes (morphisms). To visualize these, we can use 'Feynman diagrams' of a very primitive sort. In applications to linear algebra, these diagrams are often called 'spin networks', but category theorists call them 'string diagrams', and that is the term we will use. The term 'string' here has little to do with string theory: instead, the idea is that objects of our category label 'strings' or 'wires':



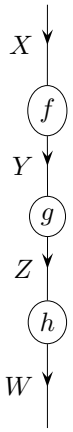
and morphisms $f\colon X \to Y$ label 'black boxes' with an input wire of type X and an output wire of type Y:



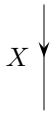
We compose two morphisms by connecting the output of one black box to the input of the next. So, the composite of $f\colon X \to Y$ and $g\colon Y \to Z$ looks like this:



Associativity of composition is then implicit:



is our notation for both $h(gf)$ and $(hg)f$. Similarly, if we draw the identity morphism $1_X \colon X \to X$ as a piece of wire of type X:



then the left and right unit laws are also implicit.

There are countless examples of categories, but we will focus on four:

- Set: the category where objects are sets.
- Hilb: the category where objects are finite-dimensional Hilbert spaces.
- nCob: the category where morphisms are n-dimensional cobordisms.
- Tang_k: the category where morphisms are k-codimensional tangles.

As we shall see, all four are closed symmetric monoidal categories, at least when k is big enough. However, the most familiar of the lot, namely Set, is the odd man out: it is 'cartesian'.

Traditionally, mathematics has been founded on the category Set, where the objects are *sets* and the morphisms are *functions*. So, when we study systems and processes in physics, it is tempting to specify a system by giving its set of states, and a process by giving a function from states of one system to states of another.

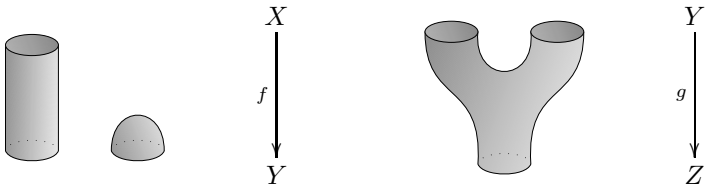
However, in quantum physics we do something subtly different: we use categories where objects are *Hilbert spaces* and morphisms are *bounded linear operators*. We specify a system by giving a Hilbert space, but this Hilbert space is not really the set of states of the system: a state is actually a ray in Hilbert space. Similarly, a bounded linear operator is not precisely a function from states of one system to states of another.

In the day-to-day practice of quantum physics, what really matters is not sets of states and functions between them, but Hilbert space and operators. One of the virtues of category

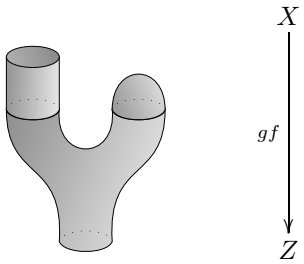
theory is that it frees us from the 'Set-centric' view of traditional mathematics and lets us view quantum physics on its own terms. As we shall see, this sheds new light on the quandaries that have always plagued our understanding of the quantum realm [9].

To avoid technical issues that would take us far afield, we will take Hilb to be the category where objects are *finite-dimensional Hilbert spaces* and morphisms are *linear operators* (automatically bounded in this case). Finite-dimensional Hilbert spaces suffice for some purporses; infinite-dimensional ones are often important, but treating them correctly would require some significant extensions of the ideas we want to explain here.

In physics we also use categories where the objects represent choices of *space*, and the morphisms represent choices of *spacetime*. The simplest is nCob, where the objects are $(n-1)$-*dimensional manifolds*, and the morphisms are n-*dimensional cobordisms*. Glossing over some subtleties that a careful treatment would discuss [79], a cobordism $f\colon X \to Y$ is an n-dimensional manifold whose boundary is the disjoint union of the $(n-1)$-dimensional manifolds X and Y. Here are a couple of cobordisms in the case $n = 2$:



We compose them by gluing the 'output' of one to the 'input' of the other. So, in the above example $gf\colon X \to Z$ looks like this:

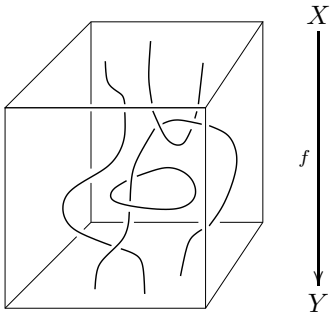


Another kind of category important in physics has objects representing *collections of particles*, and morphisms representing their *worldlines and interactions*. Feynman diagrams are the classic example, but in these diagrams the 'edges' are not taken literally as particle trajectories. An example with closer ties to topology is Tang_k.

Very roughly speaking, an object in Tang_k is a collection of points in a k-dimensional cube, while a morphism is a 'tangle': a collection of arcs and circles smoothly embedded in a $(k+1)$-dimensional cube, such that the circles lie in the interior of the cube, while the arcs touch the boundary of the cube only at its top and bottom, and only at their endpoints. A bit more precisely, tangles are 'isotopy classes' of such embedded arcs and circles: this

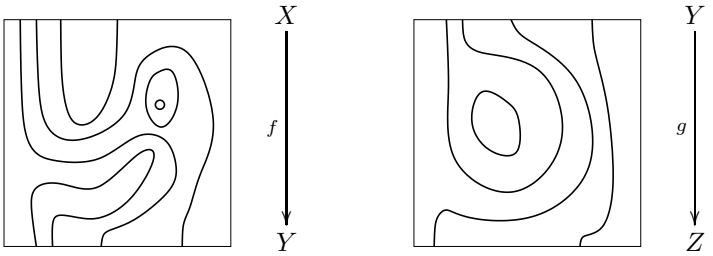
equivalence relation means that only the topology of the tangle matters, not its geometry. We compose tangles by attaching one cube to another top to bottom.

More precise definitions can be found in many sources, at least for $k = 2$, which gives tangles in a 3-dimensional cube [41, 55, 79, 86, 93, 97]. But since a picture is worth a thousand words, here is a picture of a morphism in Tang_2:

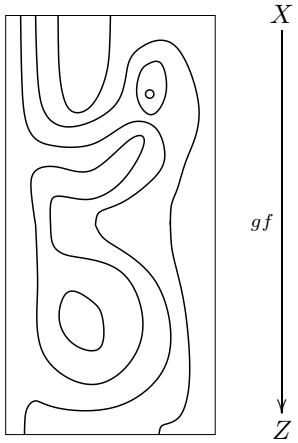


Note that we can think of a morphism in Tang_k as a 1-dimensional cobordism *embedded in a k-dimensional cube*. This is why Tang_k and nCob behave similarly in some respects.

Here are two composable morphisms in Tang_1:



and here is their composite:



Since only the tangle's topology matters, we are free to squash this rectangle into a square if we want, but we do not need to.

It is often useful to consider tangles that are decorated in various ways. For example, in an 'oriented' tangle, each arc and circle is equipped with an orientation. We can indicate this by drawing a little arrow on each curve in the tangle. In applications to physics, these curves represent worldlines of particles, and the arrows say whether each particle is going forwards or backwards in time, following Feynman's idea that antiparticles are particles going backwards in time. We can also consider 'framed' tangles. Here each curve is replaced by a 'ribbon'. In applications to physics, this keeps track of how each particle twists. This is especially important for fermions, where a 2π twist acts nontrivially. Mathematically, the best-behaved tangles are both framed and oriented [11, 86], and these are what we should use to define Tang_k. The category nCob also has a framed oriented version. However, these details will barely matter in what is to come.

It is difficult to do much with categories without discussing the maps between them. A map between categories is called a 'functor':

Definition 2. *A* **functor** $F\colon C \to D$ *from a category* C *to a category* D *is map sending:*

- *any object* $X \in C$ *to an object* $F(X) \in D$*,*
- *any morphism* $f\colon X \to Y$ *in* C *to a morphism* $F(f)\colon F(X) \to F(Y)$ *in* D*,*

in such a way that:

- F **preserves identities***: for any object* $X \in C$*,* $F(1_X) = 1_{F(X)}$*;*
- F **preserves composition***: for any pair of morphisms* $f\colon X \to Y$*,* $g\colon Y \to Z$ *in* C*,* $F(gf) = F(g)F(f)$*.*

In the sections to come, we will see that functors and natural transformations are useful for putting extra structure on categories. Here is a rather different use for functors: we can think of a functor $F\colon C \to D$ as giving a picture, or 'representation', of C in D. The idea is that F can map objects and morphisms of some 'abstract' category C to objects and morphisms of a more 'concrete' category D.

For example, consider an abstract group G. This is the same as a category with one object and with all morphisms invertible. The object is uninteresting, so we can just call it $\bullet$, but the morphisms are the elements of G, and we compose them by multiplying them. From this perspective, a **representation** of G on a finite-dimensional Hilbert space is the same as a functor $F\colon G \to \mathrm{Hilb}$. Similarly, an **action** of G on a set is the same as a functor $F\colon G \to \mathrm{Set}$. Both notions are ways of making an abstract group more concrete.

Ever since Lawvere's 1963 thesis on functorial semantics [66], the idea of functors as representations has become pervasive. However, the terminology varies from field to field. Following Lawvere, logicians often call the category C a 'theory', and call the functor $F\colon C \to D$ a 'model' of this theory. Other mathematicians might call F an 'algebra' of the theory. In this work, the default choice of D is usually the category Set.

In physics, it is the functor $F\colon C \to D$ that is called the 'theory'. Here the default choice of D is either the category we are calling Hilb or a similar category of *infinite-dimensional* Hilbert spaces. For example, both 'conformal field theories' [83] and topological quantum field theories [6] can be seen as functors of this sort.

If we think of functors as models, natural transformations are maps between models:

Definition 3. *Given two functors $F, F'\colon C \to D$, a* **natural transformation** *$\alpha\colon F \Rightarrow F'$ assigns to every object X in C a morphism $\alpha_X\colon F(X) \to F'(X)$ such that for any morphism $f\colon X \to Y$ in C, the equation $\alpha_Y\, F(f) = F'(f)\, \alpha_X$ holds in D. In other words, this square commutes:*

$$\begin{array}{ccc} F(X) & \xrightarrow{F(f)} & F(Y) \\ {\scriptstyle \alpha_X}\downarrow & & \downarrow{\scriptstyle \alpha_Y} \\ F'(X) & \xrightarrow[F'(f)]{} & F'(Y) \end{array}$$

(Going across and then down equals going down and then across.)

Definition 4. *A* **natural isomorphism** *between functors $F, F'\colon C \to D$ is a natural transformation $\alpha\colon F \Rightarrow F'$ such that α_X is an isomorphism for every $X \in C$.*

For example, suppose $F, F'\colon G \to$ Hilb are functors where G is a group, thought of as a category with one object, say $\bullet$. Then, as already mentioned, F and F' are secretly just representations of G on the Hilbert spaces $F(\bullet)$ and $F'(\bullet)$. A natural transformation $\alpha\colon F \Rightarrow F'$ is then the same as an **intertwining operator** from one representation to another: that is, a linear operator

$$A\colon F(\bullet) \to F'(\bullet)$$

satisfying

$$AF(g) = F'(g)A$$

for all group elements g.

1.2.3 Monoidal Categories

In physics, it is often useful to think of two systems sitting side by side as forming a single system. In topology, the disjoint union of two manifolds is again a manifold in its own right. In logic, the conjunction of two statement is again a statement. In programming we can combine two data types into a single 'product type'. The concept of 'monoidal category' unifies all these examples in a single framework.

A monoidal category C has a functor $\otimes\colon C \times C \to C$ that takes two objects X and Y and puts them together to give a new object $X \otimes Y$. To make this precise, we need the cartesian product of categories:

Definition 5. *The* **cartesian product** $C \times C'$ *of categories* C *and* C' *is the category where:*

- *an object is a pair* (X, X') *consisting of an object* $X \in C$ *and an object* $X' \in C'$*;*
- *a morphism from* (X, X') *to* (Y, Y') *is a pair* (f, f') *consisting of a morphism* $f \colon X \to Y$ *and a morphism* $f' \colon X' \to Y'$*;*
- *composition is done componentwise:* $(g, g')(f, f') = (gf, g'f')$*;*
- *identity morphisms are defined componentwise:* $1_{(X,X')} = (1_X, 1_{X'})$*.*

Mac Lane [68] defined monoidal categories in 1963. The subtlety of the definition lies in the fact that $(X \otimes Y) \otimes Z$ and $X \otimes (Y \otimes Z)$ are not usually equal. Instead, we should specify an isomorphism between them, called the 'associator'. Similarly, while a monoidal category has a 'unit object' I, it is not usually true that $I \otimes X$ and $X \otimes I$ equal X. Instead, we should specify isomorphisms $I \otimes X \cong X$ and $X \otimes I \cong X$. To be manageable, all these isomorphisms must then satisfy certain equations:

Definition 6. *A* **monoidal category** *consists of:*

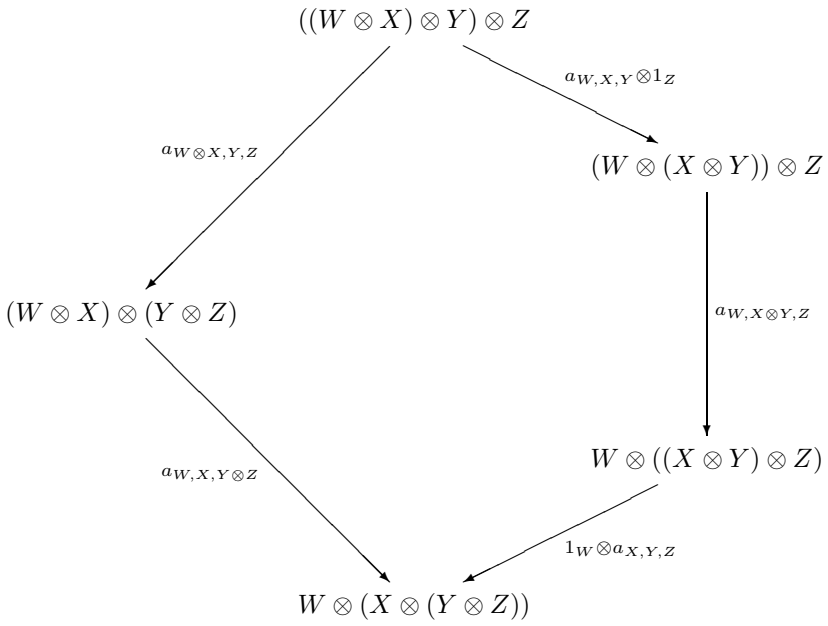
- *a category* C*,*
- *a* **tensor product** *functor* $\otimes \colon C \times C \to C$*,*
- *a* **unit object** $I \in C$*,*
- *a natural isomorphism called the* **associator***, assigning to each triple of objects* $X, Y, Z \in C$ *an isomorphism*
$$a_{X,Y,Z} : (X \otimes Y) \otimes Z \xrightarrow{\sim} X \otimes (Y \otimes Z),$$
- *natural isomorphisms called the* **left** *and* **right unitors***, assigning to each object* $X \in C$ *isomorphisms*
$$l_X : I \otimes X \xrightarrow{\sim} X$$
$$r_X : X \otimes I \xrightarrow{\sim} X,$$

such that:

- *for all* $X, Y \in C$ *the* **triangle equation** *holds:*

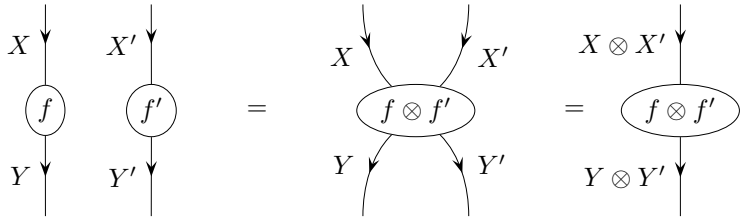
$$\begin{array}{ccc} (X \otimes I) \otimes Y & \xrightarrow{a_{X,I,Y}} & X \otimes (I \otimes Y) \\ & {\scriptstyle r_X \otimes 1_Y} \searrow \quad \swarrow {\scriptstyle 1_X \otimes l_Y} & \\ & X \otimes Y & \end{array}$$

- *for all* $W, X, Y, Z \in C$*, the* **pentagon equation** *holds:*



When we have a tensor product of four objects, there are five ways to parenthesize it, and at first glance the associator lets us build two isomorphisms from $W \otimes (X \otimes (Y \otimes Z))$ to $((W \otimes X) \otimes Y) \otimes Z$. But, the pentagon equation says these isomorphisms are equal. When we have tensor products of even more objects there are even more ways to parenthesize them, and even more isomorphisms between them built from the associator. However, Mac Lane showed that the pentagon identity implies these isomorphisms are all the same. Similarly, if we also assume the triangle equation, all isomorphisms with the same source and target built from the associator, left and right unit laws are equal.

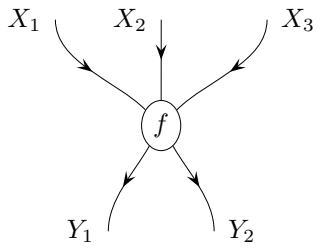
In a monoidal category we can do processes in 'parallel' as well as in 'series'. Doing processes in series is just composition of morphisms, which works in any category. But in a monoidal category we can also tensor morphisms $f \colon X \to Y$ and $f' \colon X' \to Y'$ and obtain a 'parallel process' $f \otimes f' \colon X \otimes X' \to Y \otimes Y'$. We can draw this in various ways:



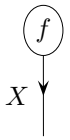
More generally, we can draw any morphism

$$f \colon X_1 \otimes \cdots \otimes X_n \to Y_1 \otimes \cdots \otimes Y_m$$

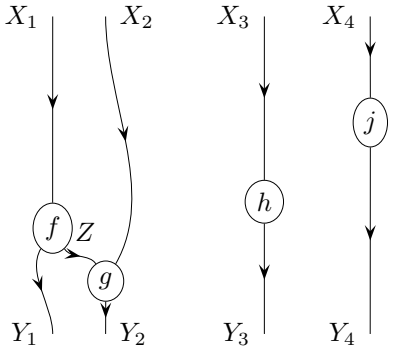
as a black box with n input wires and m output wires:



We draw the unit object I as a blank space. So, for example, we draw a morphism $f\colon I \to X$ as follows:

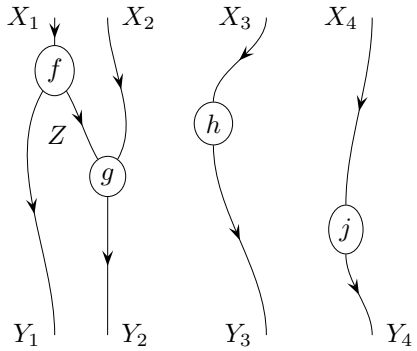


By composing and tensoring morphisms, we can build up elaborate pictures resembling Feynman diagrams:

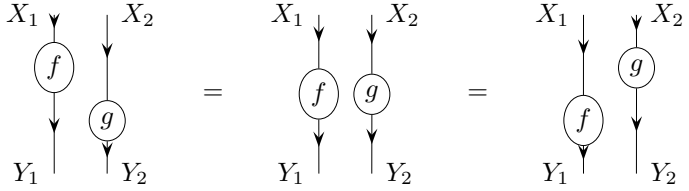


The laws governing a monoidal category allow us to neglect associators and unitors when drawing such pictures, without getting in trouble. The reason is that Mac Lane's Coherence Theorem says any monoidal category is 'equivalent', in a suitable sense, to one where all associators and unitors are identity morphisms [68].

We can also deform the picture in a wide variety of ways without changing the morphism it describes. For example, the above morphism equals this one:



Everyone who uses string diagrams for calculations in monoidal categories starts by worrying about the rules of the game: *precisely how* can we deform these pictures without changing the morphisms they describe? Instead of stating the rules precisely — which gets a bit technical — we urge you to explore for yourself what is allowed and what is not. For example, show that we can slide black boxes up and down like this:



For a formal treatment of the rules governing string diagrams, try the original papers by Joyal and Street [52] and the book by Yetter [97].

Now let us turn to examples. Here it is crucial to realize that the same category can often be equipped with different tensor products, resulting in different monoidal categories:

- There is a way to make Set into a monoidal category where $X \otimes Y$ is the cartesian product $X \times Y$ and the unit object is any one-element set. Note that this tensor product is not strictly associative, since $(x, (y, z)) \neq ((x, y), z)$, but there's a natural isomorphism $(X \times Y) \times Z \cong X \times (Y \times Z)$, and this is our associator. Similar considerations give the left and right unitors. In this monoidal category, the tensor product of $f\colon X \to Y$ and $f'\colon X' \to Y'$ is the function

$$\begin{aligned} f \times f' : X \times X' &\to Y \times Y' \\ (x, x') &\mapsto (f(x), f'(x')). \end{aligned}$$

  There is also a way to make Set into a monoidal category where $X \otimes Y$ is the disjoint union of X and Y, which we shall denote by $X + Y$. Here the unit object is the empty set. Again, as indeed with all these examples, the associative law and left/right unit laws hold only up to natural isomorphism. In this monoidal category, the tensor product of $f\colon X \to Y$ and $f'\colon X' \to Y'$ is the function

$$\begin{aligned} f + f'\colon X + X' &\to Y + Y' \\ x &\mapsto \begin{cases} f(x) \text{ if } x \in X, \\ f'(x) \text{ if } x \in X'. \end{cases} \end{aligned}$$

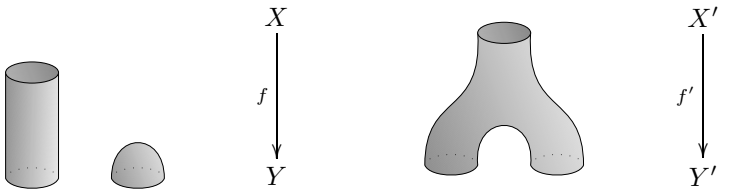
However, in what follows, when we speak of Set *as a monoidal category, we always use the cartesian product!*

- There is a way to make Hilb into a monoidal category with the usual tensor product of Hilbert spaces: $\mathbb{C}^n \otimes \mathbb{C}^m \cong \mathbb{C}^{nm}$. In this case the unit object I can be taken to be an 1-dimensional Hilbert space, for example $\mathbb{C}$.

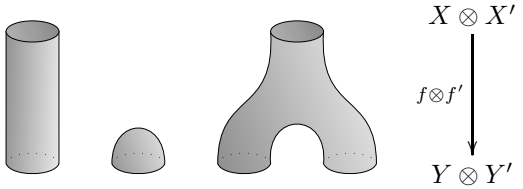
  There is also way to make Hilb into a monoidal category where the tensor product is the direct sum: $\mathbb{C}^n \oplus \mathbb{C}^m \cong \mathbb{C}^{n+m}$. In this case the unit object is the zero-dimensional Hilbert space, $\{0\}$.

  However, in what follows, when we speak of Hilb *as a monoidal category, we always use the usual tensor product!*

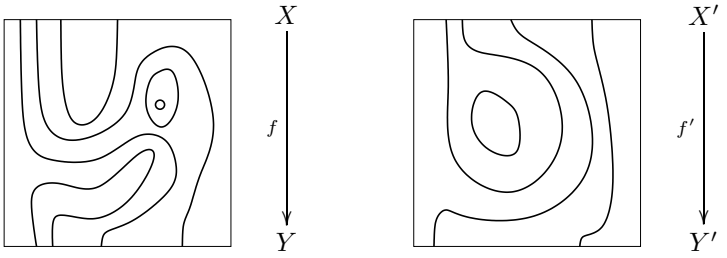
- The tensor product of objects and morphisms in nCob is given by disjoint union. For example, the tensor product of these two morphisms:



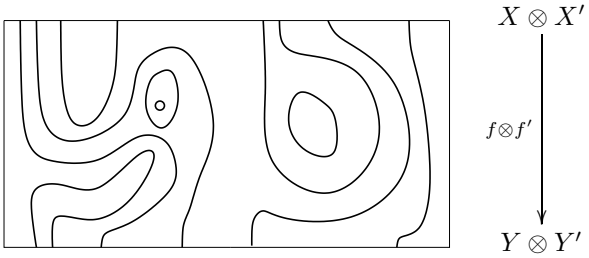
  is this:



- The category Tang_k is monoidal when $k \geq 1$, where the the tensor product is given by disjoint union. For example, given these two tangles:



  their tensor product looks like this:



The example of Set with its cartesian product is different from our other three main examples, because the cartesian product of sets $X \times X'$ comes equipped with functions called 'projections' to the sets X and X':

$$X \xleftarrow{p} X \times X' \xrightarrow{p'} X'$$

Our other main examples lack this feature — though Hilb made into a monoidal category using $\oplus$ has projections. Also, every set has a unique function to the the one-element set:

$$!_X \colon X \to I.$$

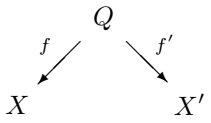
Again, our other main examples lack this feature, though Hilb made into a monoidal category using $\oplus$ has it. A fascinating feature of quantum mechanics is that we make Hilb into a monoidal category using $\otimes$ instead of $\oplus$, even though the latter approach would lead to a category more like Set.

We can isolate the special features of the cartesian product of sets and its projections, obtaining a definition that applies to any category:

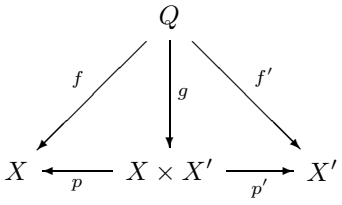
Definition 7. *Given objects X and X' in some category, we say an object $X \times X'$ equipped with morphisms*

$$X \xleftarrow{p} X \times X' \xrightarrow{p'} X'$$

is a **cartesian product** *(or simply* **product***) of X and X' if for any object Q and morphisms*



there exists a unique morphism $g \colon Q \to X \times X'$ making the following diagram commute:



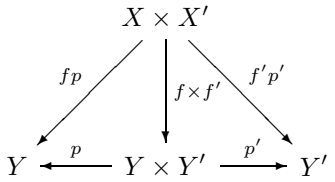
(That is, $f = pg$ and $f' = p'g$.) We say a category has **binary products** *if every pair of objects has a product.*

The product may not exist, and it may not be unique, but when it exists it is unique up to a canonical isomorphism. This justifies our speaking of 'the' product of objects X and Y when it exists, and denoting it as $X \times Y$.

The definition of cartesian product, while absolutely fundamental, is a bit scary at first sight. To illustrate its power, let us do something with it: combine two morphisms $f\colon X \to Y$ and $f'\colon X' \to Y'$ into a single morphism

$$f \times f'\colon X \times X' \to Y \times Y'.$$

The definition of cartesian product says how to build a morphism of this sort out of a pair of morphisms: namely, morphisms from $X \times X'$ to Y and Y'. If we take these to be fp and $f'p'$, we obtain $f \times f'$:



Next, let us isolate the special features of the one-element set:

Definition 8. *An object 1 in a category C is* **terminal** *if for any object $Q \in C$ there exists a unique morphism from Q to 1, which we denote as $!_Q\colon Q \to 1$.*

Again, a terminal object may not exist and may not be unique, but it is unique up to a canonical isomorphism. This is why we can speak of 'the' terminal object of a category, and denote it by a specific symbol, 1.

We have introduced the concept of binary products. One can also talk about n-ary products for other values of n, but a category with binary products has n-ary products for all $n \geq 1$, since we can construct these as iterated binary products. The case $n = 1$ is trivial, since the product of one object is just that object itself (up to canonical isomorphism). The remaining case is $n = 0$. The zero-ary product of objects, if it exists, is just the terminal object. So, we make the following definition:

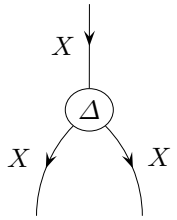
Definition 9. *A category has* **finite products** *if it has binary products and a terminal object.*

A category with finite products can always be made into a monoidal category by choosing a specific product $X \times Y$ to be to the tensor product $X \otimes Y$, and choosing a specific terminal object to be the unit object. It takes a bit of work to show this! A monoidal category of this form is called **cartesian**.

In a cartesian category, we can 'duplicate and delete information'. In general, the definition of cartesian products gives a way to take two morphisms $f_1 \colon Q \to X$ and $f_2 \colon Q \to Y$ and combine them into a single morphism from Q to $X \times Y$. If we take $Q = X = Y$ and take f_1 and f_2 to be the identity, we obtain the **diagonal** or **duplication** morphism:

$$\Delta_X \colon X \to X \times X.$$

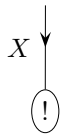
In the category Set one can check that this maps any element $x \in X$ to the pair (x, x). In general, we can draw the diagonal as follows:



Similarly, we call the unique map to the terminal object

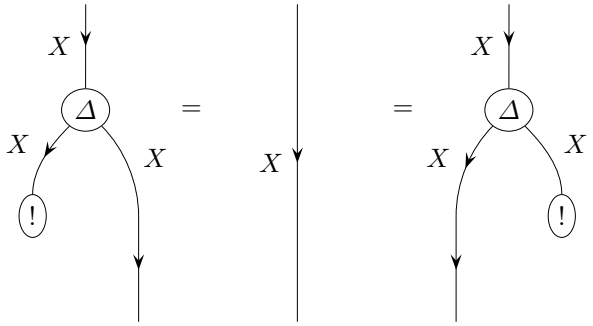
$$!_X \colon X \to 1$$

the **deletion** morphism, and draw it as follows:



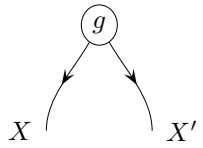
Note that we draw the unit object as an empty space.

A fundamental fact about cartesian categories is that duplicating something and then deleting either copy is the same as doing nothing at all! In string diagrams, this says:

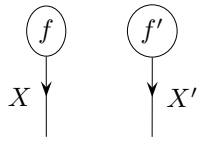


We leave the proof as an exercise for the reader.

Many of the puzzling features of quantum theory come from the noncartesianness of the usual tensor product in Hilb. For example, in a cartesian category, every morphism



is actually of the form

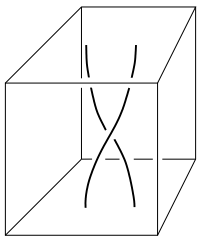


In the case of Set, this says that every point of the set $X \times X'$ comes from a point of X and a point of X'. In physics, this would say that every state g of the combined system $X \otimes X'$ is built by combining states of the systems X and X'. Bell's theorem [17] says that is *not* true in quantum theory. The reason is that quantum theory uses the noncartesian monoidal category Hilb!

Also, in quantum theory we *cannot* freely duplicate or delete information. Wootters and Zurek [96] proved a precise theorem to this effect, focused on duplication: the 'no-cloning theorem'. One can also prove a 'no-deletion theorem'. Again, these results rely on the noncartesian tensor product in Hilb.

1.2.4 Braided Monoidal Categories

In physics, there is often a process that lets us 'switch' two systems by moving them around each other. In topology, there is a tangle that describes the process of switching two points:



In logic, we can switch the order of two statements in a conjunction: the statement 'X and Y' is isomorphic to 'Y and X'. In computation, there is a simple program that switches the order of two pieces of data. A monoidal category in which we can do this sort of thing is called 'braided':

Definition 10. *A* **braided monoidal category** *consists of:*

- *a monoidal category C,*
- *a natural isomorphism called the* **braiding** *that assigns to every pair of objects $X, Y \in C$ an isomorphism*

$$b_{X,Y} \colon X \otimes Y \to Y \otimes X,$$

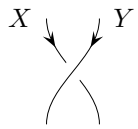
such that the **hexagon equations** *hold:*

$$\begin{array}{ccccc}
X \otimes (Y \otimes Z) & \xrightarrow{a^{-1}_{X,Y,Z}} & (X \otimes Y) \otimes Z & \xrightarrow{b_{X,Y} \otimes 1_Z} & (Y \otimes X) \otimes Z \\
\Big\downarrow{\scriptstyle b_{X,Y \otimes Z}} & & & & \Big\downarrow{\scriptstyle a_{Y,X,Z}} \\
(Y \otimes Z) \otimes X & \xleftarrow[a^{-1}_{Y,Z,X}]{} & Y \otimes (Z \otimes X) & \xleftarrow[1_Y \otimes b_{X,Z}]{} & Y \otimes (X \otimes Z)
\end{array}$$

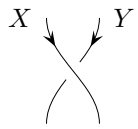
$$\begin{array}{ccccc}
(X \otimes Y) \otimes Z & \xrightarrow{a_{X,Y,Z}} & X \otimes (Y \otimes Z) & \xrightarrow{1_X \otimes b_{Y,Z}} & X \otimes (Z \otimes Y) \\
\Big\downarrow{\scriptstyle b_{X \otimes Y,Z}} & & & & \Big\downarrow{\scriptstyle a^{-1}_{X,Z,Y}} \\
Z \otimes (X \otimes Y) & \xleftarrow[a_{Z,X,Y}]{} & (Z \otimes X) \otimes Y & \xleftarrow[b_{Z,X} \otimes 1_Y]{} & (X \otimes Z) \otimes Y
\end{array}$$

The first hexagon equation says that switching the object X past $Y \otimes Z$ all at once is the same as switching it past Y and then past Z (with some associators thrown in to move the parentheses). The second one is similar: it says switching $X \otimes Y$ past Z all at once is the same as doing it in two steps.

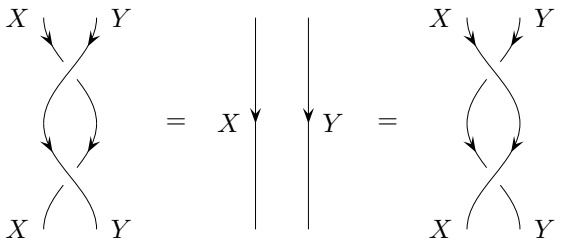
In string diagrams, we draw the braiding $b_{X,Y} \colon X \otimes Y \to Y \otimes X$ like this:



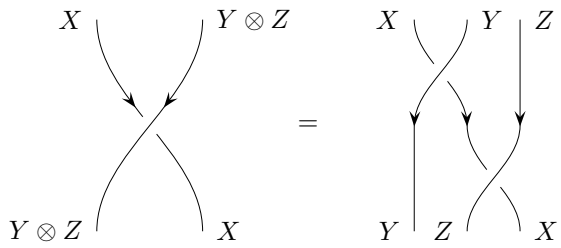
We draw its inverse $b^{-1}_{X,Y}$ like this:

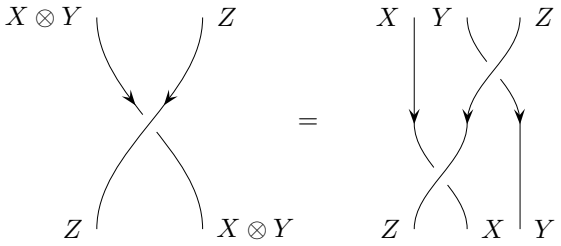


This is a nice notation, because it makes the equations saying that $b_{X,Y}$ and $b^{-1}_{X,Y}$ are inverses 'topologically true':

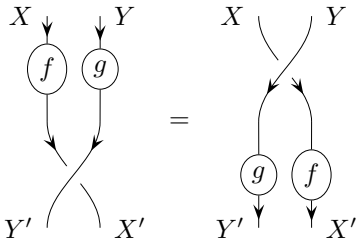


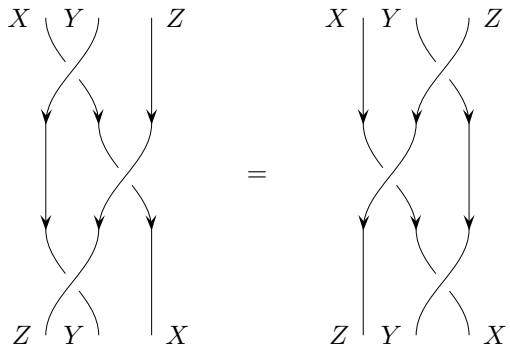
Here are the hexagon equations as string diagrams:





For practice, we urge you to prove the following equations:





If you get stuck, here are some hints. The first equation follows from the naturality of the braiding. The second is called the **Yang–Baxter equation** and follows from a combination of naturality and the hexagon equations [53].

Next, here are some examples. There can be many different ways to give a monoidal category a braiding, or none. However, most of our favorite examples come with well-known 'standard' braidings:

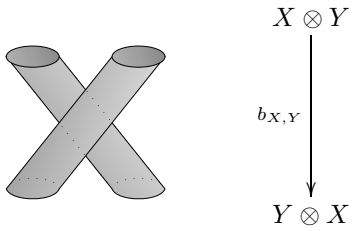
- Any cartesian category automatically becomes braided, and in Set with its cartesian product, this standard braiding is given by:

$$\begin{aligned} b_{X,Y} : X \times Y &\to Y \times X \\ (x, y) &\mapsto (y, x). \end{aligned}$$

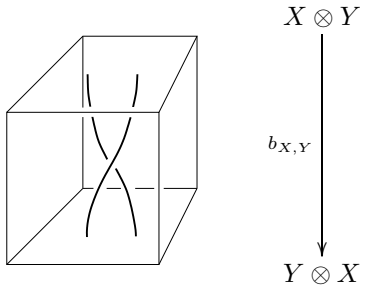
- In Hilb with its usual tensor product, the standard braiding is given by:

$$\begin{aligned} b_{X,Y} : X \otimes Y &\to Y \otimes X \\ x \otimes y &\mapsto y \otimes x. \end{aligned}$$

- The monoidal category nCob has a standard braiding where $b_{X,Y}$ is diffeomorphic to the disjoint union of cylinders $X \times [0,1]$ and $Y \times [0,1]$. For 2Cob this braiding looks as follows when X and Y are circles:



- The monoidal category Tang_k has a standard braiding when $k \geq 2$. For $k = 2$ this looks as follows when X and Y are each a single point:



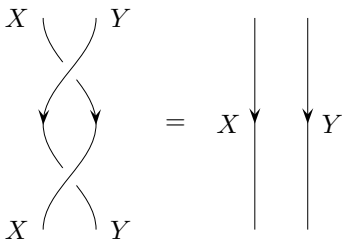
The example of Tang_k illustrates an important pattern. Tang_0 is just a category, because in 0-dimensional space we can only do processes in 'series': that is, compose morphisms. Tang_1 is a monoidal category, because in 1-dimensional space we can also do processes in 'parallel': that is, tensor morphisms. Tang_2 is a braided monoidal category, because in 2-dimensional space there is room to move one object around another. Next we shall see what happens when space has 3 or more dimensions!

1.2.5 Symmetric Monoidal Categories

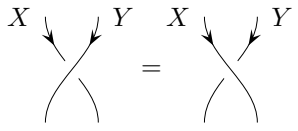
Sometimes switching two objects and switching them again is the same as doing nothing at all. Indeed, this situation is very familiar. So, the first braided monoidal categories to be discovered were 'symmetric' ones [68]:

Definition 11. *A* **symmetric monoidal category** *is a braided monoidal category where the braiding satisfies* $b_{X,Y} = b_{Y,X}^{-1}$.

So, in a symmetric monoidal category,



or equivalently:



Any cartesian category automatically becomes a symmetric monoidal category, so Set is symmetric. It is also easy to check that Hilb, nCob are symmetric monoidal categories. So is Tang_k for $k \geq 3$.

Interestingly, Tang_k 'stabilizes' at $k = 3$: increasing the value of k beyond this value merely gives a category equivalent to Tang_3. The reason is that we can already untie all knots in 4-dimensional space; adding extra dimensions has no real effect. In fact, Tang_k for $k \geq 3$ is equivalent to 1Cob. This is part of a conjectured larger pattern called the 'Periodic Table' of n-categories [11]. A piece of this is shown in Table 1.3.

An n-category has not only morphisms going between objects, but 2-morphisms going between morphisms, 3-morphisms going between 2-morphisms and so on up to n-morphisms. In topology we can use n-categories to describe tangled higher-dimensional surfaces [12], and in physics we can use them to describe not just particles but also strings and higher-dimensional membranes [11, 13]. The Rosetta Stone we are describing concerns only the $n = 1$ column of the Periodic Table. So, it is probably just a fragment of a larger, still buried n-categorical Rosetta Stone.

	$n = \mathbf{0}$	$n = \mathbf{1}$	$n = \mathbf{2}$
$k = \mathbf{0}$	sets	categories	2-categories
$k = \mathbf{1}$	monoids	monoidal categories	monoidal 2-categories
$k = \mathbf{2}$	commutative monoids	braided monoidal categories	braided monoidal 2-categories
$k = \mathbf{3}$	"	symmetric monoidal categories	sylleptic monoidal 2-categories
$k = \mathbf{4}$	"	"	symmetric monoidal 2-categories
$k = \mathbf{5}$	"	"	"
$k = \mathbf{6}$	"	"	"

Table 1.3. The Periodic Table: conjectured descriptions of $(n + k)$-categories with only one j-morphism for $j < k$.

1.2.6 Closed Categories

In quantum mechanics, one can encode a linear operator $f\colon X \to Y$ into a quantum state using a technique called 'gate teleportation' [48]. In topology, there is a way to take any

tangle $f\colon X \to Y$ and bend the input back around to make it part of the output. In logic, we can take a proof that goes from some assumption X to some conclusion Y and turn it into a proof that goes from no assumptions to the conclusion 'X implies Y'. In computer science, we can take any program that takes input of type X and produces output of type Y, and think of it as a piece of data of a new type: a 'function type'. The underlying concept that unifies all these examples is the concept of a 'closed category'.

Given objects X and Y in any category C, there is a *set* of morphisms from X to Y, denoted $\hom(X, Y)$. In a closed category there is also an *object* of morphisms from X to Y, which we denote by $X \multimap Y$. (Many other notations are also used.) In this situation we speak of an 'internal hom', since the object $X \multimap Y$ lives inside C, instead of 'outside', in the category of sets.

Closed categories were introduced in 1966, by Eilenberg and Kelly [38]. While these authors were able to define a closed structure for any category, it turns out that the internal hom is most easily understood for monoidal categories. The reason is that when our category has a tensor product, it is closed precisely when morphisms from $X \otimes Y$ to Z are in natural one-to-one correspondence with morphisms from Y to $X \multimap Z$. In other words, it is closed when we have a natural isomorphism

$$\begin{array}{ccc} \hom(X \otimes Y, Z) & \cong & \hom(Y, X \multimap Z) \\ f & \mapsto & \tilde{f} \end{array}$$

For example, in the category Set, if we take $X \otimes Y$ to be the cartesian product $X \times Y$, then $X \multimap Z$ is just the set of functions from Y to Z, and we have a one-to-one correspondence between

- functions f that eat elements of $X \times Y$ and spit out elements of Z

and

- functions $\tilde{f}$ that eat elements of Y and spit out functions from X to Z.

This correspondence goes as follows:

$$\tilde{f}(x)(y) = f(x, y).$$

Before considering other examples, we should make the definition of 'closed monoidal category' completely precise. For this we must note that for any category C, there is a functor

$$\hom\colon C^{\mathrm{op}} \times C \to \mathrm{Set}.$$

Definition 12. *The* **opposite category** C^{op} *of a category C has the same objects as C, but a morphism $f\colon x \to y$ in C^{op} is a morphism $f\colon y \to x$ in C, and the composite gf in C^{op} is the composite fg in C.*

Definition 13. *For any category C, the* **hom functor**

$$\hom\colon C^{\mathrm{op}} \times C \to \mathrm{Set}$$

sends any object $(X, Y) \in C^{\mathrm{op}} \times C$ to the set $\hom(X, Y)$, and sends any morphism $(f, g) \in C^{\mathrm{op}} \times C$ to the function

$$\begin{aligned} \hom(f, g)\colon \hom(X, Y) &\to \hom(X', Y') \\ h &\mapsto ghf \end{aligned}$$

when $f\colon X' \to X$ and $g\colon Y \to Y'$ as morphisms in C.

Definition 14. *A monoidal category C is* **left closed** *if there is an* **internal hom** *functor*

$$\multimap\colon C^{\mathrm{op}} \times C \to C$$

together with a natural isomorphism c called **currying** *that assigns to any objects $X, Y, Z \in C$ a bijection*

$$c_{X,Y,Z}\colon \hom(X \otimes Y, Z) \xrightarrow{\sim} \hom(X, Y \multimap Z)$$

It is **right closed** *if there is an internal hom functor as above and a natural isomorphism*

$$c_{X,Y,Z}\colon \hom(X \otimes Y, Z) \xrightarrow{\sim} \hom(Y, X \multimap Z).$$

The term 'currying' is mainly used in computer science, after the work of Curry [34]. In the rest of this section we only consider *right* closed monoidal categories. Luckily, there is no real difference between left and right closed for a braided monoidal category, as the braiding gives an isomorphism $X \otimes Y \cong Y \otimes X$.

All our examples of monoidal categories are closed, but we shall see that, yet again, Set is different from the rest:

- The cartesian category Set is closed, where $X \multimap Y$ is just the set of functions from X to Y. In Set or any other cartesian closed category, the internal hom $X \multimap Y$ is usually denoted Y^X. To minimize the number of different notations and emphasize analogies between different contexts, we shall not do this: we shall always use $X \multimap Y$. To treat Set as *left* closed, we define the curried version of $f\colon X \times Y \to Z$ as above:

  $$\tilde{f}(x)(y) = f(x, y).$$

  To treat it as *right* closed, we instead define it by

  $$\tilde{f}(y)(x) = f(x, y).$$

  This looks a bit awkward, but it will be nice for string diagrams.

- The symmetric monoidal category Hilb with its usual tensor product is closed, where $X \multimap Y$ is the set of linear operators from X to Y, made into a Hilbert space in a standard way. In this case we have an isomorphism

  $$X \multimap Y \cong X^* \otimes Y$$

  where X^* is the dual of the Hilbert space X, that is, the set of linear operators $f\colon X \to \mathbb{C}$, made into a Hilbert space in the usual way.

- The monoidal category Tang_k $(k \geq 1)$ is closed. As with Hilb, we have

$$X \multimap Y \cong X^* \otimes Y$$

  where X^* is the orientation-reversed version of X.

- The symmetric monoidal category nCob is also closed; again

$$X \multimap Y \cong X^* \otimes Y$$

  where X^* is the $(n-1)$-manifold X with its orientation reversed.

Except for Set, all these examples are actually 'compact'. This basically means that $X \multimap Y$ is isomorphic to $X^* \otimes Y$, where X^* is some object called the 'dual' of X. But to make this precise, we need to define the 'dual' of an object in an arbitrary monoidal category.

To do this, let us generalize from the case of Hilb. As already mentioned, each object $X \in \mathrm{Hilb}$ has a dual X^* consisting of all linear operators $f\colon X \to I$, where the unit object I is just $\mathbb{C}$. There is thus a linear operator

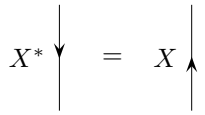
$$\begin{aligned} e_X\colon X \otimes X^* &\to I \\ x \otimes f &\mapsto f(x) \end{aligned}$$

called the **counit** of X. Furthermore, the space of all linear operators from X to $Y \in \mathrm{Hilb}$ can be identified with $X^* \otimes Y$. So, there is also a linear operator called the **unit** of X:

$$\begin{aligned} i_x\colon I &\to X^* \otimes X \\ c &\mapsto c\, 1_X \end{aligned}$$

sending any complex number c to the corresponding multiple of the identity operator.

The significance of the unit and counit become clearer if we borrow some ideas from Feynman. In physics, if X is the Hilbert space of internal states of some particle, X^* is the Hilbert space for the corresponding antiparticle. Feynman realized that it is enlightening to think of antiparticles as particles going backwards in time. So, we draw a wire labelled by X^* as a wire labelled by X, but with an arrow pointing 'backwards in time': that is, up instead of down:



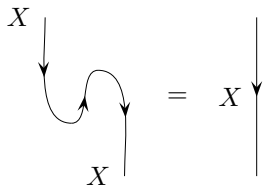
(Here we should admit that most physicists use the opposite convention, where time marches up the page. Since we read from top to bottom, we prefer to let time run down the page.)

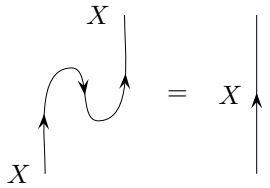
If we draw X^* as X going backwards in time, we can draw the unit as a **cap**:

and the counit as a **cup**:

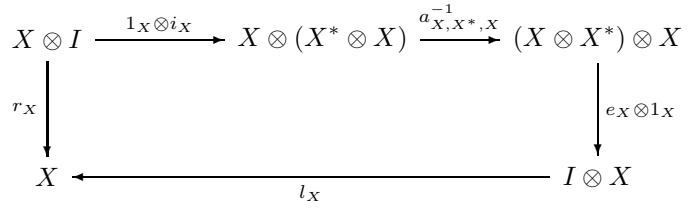
In Feynman diagrams, these describe the *creation* and *annihilation* of virtual particle-antiparticle pairs!

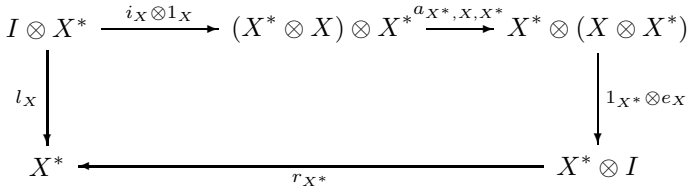
It then turns out that the unit and counit satisfy two equations, the **zig-zag equations**:





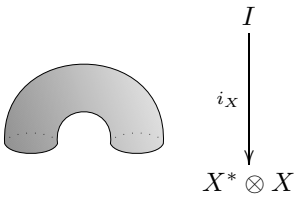
Verifying these is a fun exercise in linear algebra, which we leave to the reader. If we write these equations as commutative diagrams, we need to include some associators and unitors, and they become a bit intimidating:



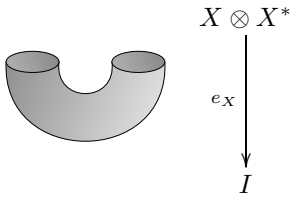


But, they really just say that zig-zags in string diagrams can be straightened out.

This is particularly vivid in examples like Tang_k and nCob. For example, in 2Cob, taking X to be the circle, the unit looks like this:



while the counit looks like this:



In this case, the zig-zag identities say we can straighten a wiggly piece of pipe.

Now we are ready for some definitions:

Definition 15. *Given objects X^* and X in a monoidal category, we call X^* a* **right dual** *of X, and X a* **left dual** *of X^*, if there are morphisms*

$$i_X \colon I \to X^* \otimes X$$

and

$$e_X \colon X \otimes X^* \to I,$$

called the **unit** *and* **counit** *respectively, satisfying the zig-zag equations.*

One can show that the left or right dual of an object is unique up to canonical isomorphism. So, we usually speak of 'the' right or left dual of an object, when it exists.

Definition 16. *A monoidal category C is* **compact** *if every object $X \in C$ has both a left dual and a right dual.*

Often the term 'autonomous' is used instead of 'compact' here. Many authors reserve the term 'compact' for the case where C is symmetric or at least braided; then left duals are the same as right duals, and things simplify [41]. To add to the confusion, compact symmetric monoidal categories are often called simply 'compact closed categories'.

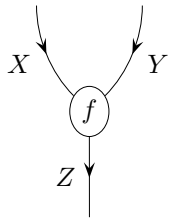
A partial explanation for the last piece of terminology is that any compact monoidal category is automatically closed! For this, we define the internal hom on objects by

$$X \multimap Y = X^* \otimes Y.$$

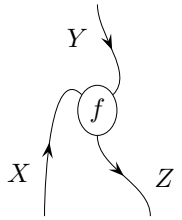
We must then show that the $*$ operation extends naturally to a functor $*\colon C \to C$, so that $\multimap$ is actually a functor. Finally, we must check that there is a natural isomorphism

$$\hom(X \otimes Y, Z) \cong \hom(Y, X^* \otimes Z)$$

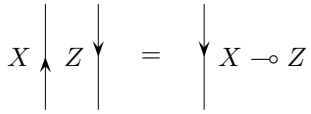
In terms of string diagrams, this isomorphism takes any morphism



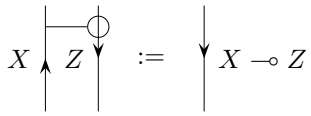
and bends back the input wire labelled X to make it an output:



Now, in a compact monoidal category, we have:

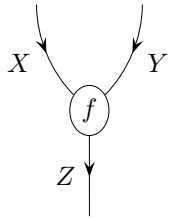


But in general, closed monoidal categories don't allow arrows pointing up! So for these, drawing the internal hom is more of a challenge. We can use the same style of notation as long as we add a decoration — a **clasp** — that binds two strings together:

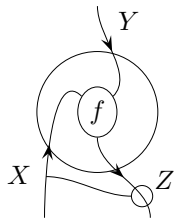


Only when our closed monoidal category happens to be compact can we eliminate the clasp.

Suppose we are working in a closed monoidal category. Since we draw a morphism $f\colon X \otimes Y \to Z$ like this:

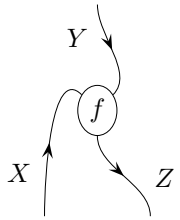


we can draw its curried version $\tilde{f}\colon Y \to X \multimap Z$ by bending down the input wire labelled X to make it part of the output:



Note that where we bent back the wire labelled X, a cap like this appeared:

Closed monoidal categories don't really have a cap unless they are compact. So, we drew a **bubble** enclosing f and the cap, to keep us from doing any illegal manipulations. In the compact case, both the bubble and the clasp are unnecessary, so we can draw $\tilde{f}$ like this:



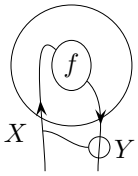
An important special case of currying gives the **name** of a morphism $f\colon X \to Y$,

$$\ulcorner f \urcorner \colon I \to X \multimap Y.$$

This is obtained by currying the morphism

$$fr_x \colon I \otimes X \to Y.$$

In string diagrams, we draw $\ulcorner f \urcorner$ as follows:



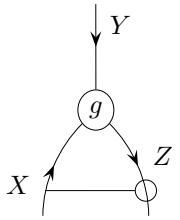
In the category Set, the unit object I is the one-element set. So, a morphism from I to any set Q picks out a point of Q. In particular, the name $\ulcorner f \urcorner \colon I \to X \multimap Y$ picks out the element of $X \multimap Y$ corresponding to the function $f\colon X \to Y$. More generally, in any cartesian closed category, a morphism from 1 to an object Q is called a **point** of Q. So, even in this case, we can say the name of a morphism $f\colon X \to Y$ is a point of $X \multimap Y$.

Something similar works for Hilb, though this example is compact rather than cartesian. In Hilb, the unit object I is just $\mathbb{C}$. So, a nonzero morphism from I to any Hilbert space Q picks out a nonzero vector in Q, which we can normalize to obtain a **state** in Q: that is, a unit vector. In particular, the the name of a nonzero morphism $f\colon X \to Y$ gives a state of $X^* \otimes Y$. This method of encoding operators as states is the basis of 'gate teleportation' [48].

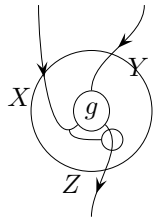
Currying is a bijection, so we can also **uncurry**:

$$\begin{aligned} c^{-1}_{X,Y,Z}\colon \hom(Y, X \multimap Z) &\xrightarrow{\sim} \hom(X \otimes Y, Z) \\ g &\mapsto \underset{\sim}{g}. \end{aligned}$$

Since we draw a morphism $g\colon Y \to X \multimap Z$ like this:



we draw its 'uncurried' version $\underset{\sim}{g}\colon X \otimes Y \to Z$ by bending the output X up to become an input:



Again, we must put a bubble around the 'cup' formed when we bend down the wire labelled Y, unless we are in a compact monoidal category.

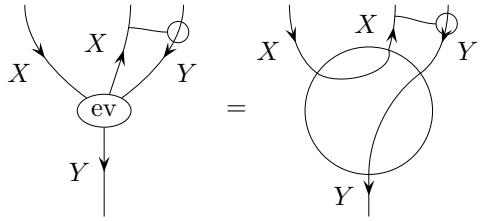
A good example of uncurrying is the **evaluation** morphism:

$$\mathrm{ev}_{X,Y}\colon X \otimes (X \multimap Y) \to Y.$$

This is obtained by uncurrying the identity

$$1_{X \multimap Y}\colon (X \multimap Y) \to (X \multimap Y).$$

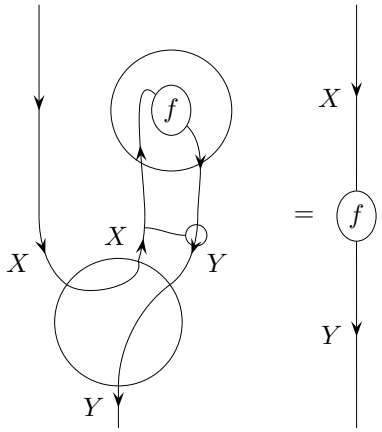
In Set, $\mathrm{ev}_{X,Y}$ takes any function from X to Y and evaluates it at any element of X to give an element of Y. In terms of string diagrams, the evaluation morphism looks like this:



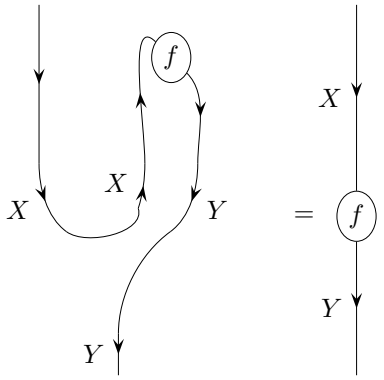
In any closed monoidal category, we can recover a morphism from its name using evaluation. More precisely, this diagram commutes:

$$\begin{array}{ccc} X \otimes I & \xleftarrow{r^{-1}} & X \\ {\scriptstyle 1_X \otimes \ulcorner f \urcorner}\downarrow & & \downarrow{\scriptstyle f} \\ X \otimes (X \multimap Y) & \xrightarrow[\mathrm{ev}_{X,Y}]{} & Y \end{array}$$

Or, in terms of string diagrams:



We leave the proof of this as an exercise. In general, one must use the naturality of currying. In the special case of a compact monoidal category, there is a nice picture proof! Simply pop the bubbles and remove the clasp:



The result then follows from one of the zig-zag identities.

In our rapid introduction to string diagrams, we have not had time to illustrate how these diagrams become a powerful tool for solving concrete problems. So, here are some starting points for further study:

- Representations of Lie groups play a fundamental role in quantum physics, especially gauge field theory. Every Lie group has a compact symmetric monoidal category of finite-dimensional representations. In his book *Group Theory*, Cvitanovic [35] develops detailed string diagram descriptions of these representation categories for the classical Lie groups $\mathrm{SU}(n)$, $\mathrm{SO}(n)$, $\mathrm{SU}(n)$ and also the more exotic 'exceptional' Lie groups. His book also illustrates how this technology can be used to simplify difficult calculations in gauge field theory.
- Quantum groups are a generalization of groups which show up in 2d and 3d physics. The big difference is that a quantum group has compact *braided* monoidal category of finite-dimensional representation. Kauffman's *Knots and Physics* [56] is an excellent introduction to how quantum groups show up in knot theory and physics; it is packed with string diagrams. For more details on quantum groups and braided monoidal categories, see the book by Kassel [55].
- Kauffman and Lins [57] have written a beautiful string diagram treatment of the category of representations of the simplest quantum group, $SU_q(2)$. They also use it to construct some famous 3-manifold invariants associated to 3d and 4d topological quantum field theories: the Witten–Reshetikhin–Turaev, Turaev–Viro and Crane–Yetter invariants. In this example, string diagrams are often called 'q-deformed spin networks' [87]. For generalizations to other quantum groups, see the more advanced texts by Turaev [93] and by Bakalov and Kirillov [14]. The key ingredient is a special class of compact braided monoidal categories called 'modular tensor categories'.
- Kock [61] has written a nice introduction to 2d topological quantum field theories which uses diagrammatic methods to work with 2Cob.
- Abramsky, Coecke and collaborators [2, 3, 4, 29, 31, 32] have developed string diagrams as a tool for understanding quantum computation. The easiest introduction is Coecke's 'Kindergarten quantum mechanics' [30].

1.2.7 Dagger Categories

Our discussion would be sadly incomplete without an important admission: *nothing we have done so far with Hilbert spaces used the inner product!* So, we have not yet touched on the essence of quantum theory.

In fact, everything we have said about Hilb applies equally well to Vect: the category of finite-dimensional *vector spaces* and linear operators. Both Hilb and Vect are compact symmetric monoidal categories. In fact, these compact symmetric monoidal categories are 'equivalent' in a certain precise sense [69].

So, what makes Hilb different? In terms of category theory, the special thing is that we can take the Hilbert space adjoint of any linear operator $f\colon X \to Y$ between finite-dimensional Hilbert spaces, getting an operator $f^\dagger\colon Y \to X$. This ability to 'reverse' morphisms makes Hilb into a 'dagger category':

Definition 17. *A* **dagger category** *is a category C such that for any morphism $f\colon X \to Y$ in C there is a specified morphism $f^\dagger\colon Y \to X$ such that*

$$(gf)^\dagger = f^\dagger g^\dagger$$

for every pair of composable morphisms f and g, and

$$(f^\dagger)^\dagger = f$$

for every morphism f.

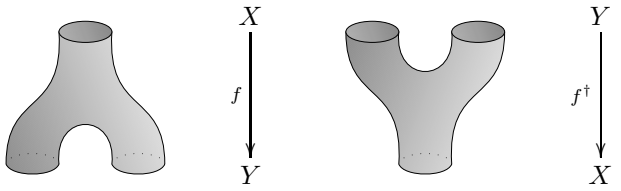
Equivalently, a dagger category is one equipped with a functor $\dagger\colon C \to C^{\mathrm{op}}$ that is the identity on objects and satisfies $(f^\dagger)^\dagger = f$ for every morphism.

In fact, all our favorite examples of categories can be made into dagger categories, except for Set:

- The category Hilb becomes a dagger category as follows. Given any morphism $f\colon X \to Y$ in Hilb, there is a morphism $f^\dagger\colon Y \to X$, the **Hilbert space adjoint** of f, defined by

  $$\langle f^\dagger \psi, \phi \rangle = \langle \psi, f\phi \rangle$$

  for all $\phi \in X$, $\psi \in Y$.
- For any k, the category Tang_k becomes a dagger category where we obtain $f^\dagger\colon Y \to X$ by reflecting $f\colon X \to Y$ in the vertical direction, and then switching the direction of the little arrows denoting the orientations of arcs and circles.
- For any n, the category nCob becomes a dagger category where we obtain $f^\dagger\colon Y \to X$ by switching the input and output of $f\colon X \to Y$, and then switching the orientation of each connected component of f. Again, a picture speaks a thousand words:



In applications to physics, this dagger operation amounts to 'switching the future and the past'.

- There is no way to make Set into a dagger category, since there is a function from the empty set to the 1-element set, but none the other way around.

In all the dagger categories above, the dagger structure interacts in a nice way with the monoidal structure and also, when it exists, the braiding. One can write a list of axioms characterizing how this works [2, 3, 85]. So, it seems that the ability to 'reverse' morphisms is another way in which categories of a quantum flavor differ from the category of sets and functions. This has important implications for the foundations of quantum theory [9] and also for topological quantum field theory [11], where dagger categories seem to be part of larger story involving 'n-categories with duals' [12]. However, this story is still poorly understood — there is much more work to be done.

1.3 Logic

1.3.1 Background

Symmetric monoidal closed categories show up not only in physics and topology, but also in logic. We would like to explain how. To set the stage, it seems worthwhile to sketch a few ideas from 20th-century logic.

Modern logicians study many systems of reasoning beside ordinary classical logic. Of course, even classical logic comes in various degrees of strength. First there is the 'propositional calculus', which allows us to reason with abstract propositions $X, Y, Z, \ldots$ and these logical connectives:

and	$\wedge$
or	$\vee$
implies	$\Rightarrow$
not	$\neg$
true	$\top$
false	$\bot$

Then there is the 'predicate calculus', which also allows variables like $x, y, z, \ldots$, predicates like $P(x)$ and $Q(x, y, z)$, and the symbols 'for all' ($\forall$) and 'there exists' ($\exists$), which allow us to quantify over variables. There are also higher-order systems that allow us to quantify over predicates, and so on. To keep things simple, we mainly confine ourselves to the propositional calculus in what follows. But even here, there are many alternatives to the 'classical' version!

The most-studied of these alternative systems are *weaker* than classical logic: they make it harder or even impossible to prove things we normally take for granted. One reason is that some logicians deny that certain familiar principles are actually valid. But there are also subtler reasons. One is that studying systems with rules of lesser strength allows for a fine-grained study of precisely which methods of reasoning are needed to prove which results. Another reason — the one that concerns us most here — is that dropping familiar rules and then adding them back in one at at time sheds light on the connection between logic and category theory.

For example, around 1907 Brouwer [50] began advocating 'intuitionism'. As part of this, he raised doubts about the law of excluded middle, which amounts to a rule saying that from $\neg\neg X$ we can deduce X. One problem with this principle is that proofs using it are not 'constructive'. For example, we may prove by contradiction that some equation has a solution, but still have no clue how to construct the solution. For Brouwer, this meant the principle was invalid.

Anyone who feels the law of excluded middle is invalid is duty-bound to study intuitionistic logic. But, there is another reason for studying this system. Namely: we do not really *lose* anything by dropping the law of excluded middle! Instead, we *gain* a fine-grained distinction: the distinction between a direct proof of X and a proof by contradiction, which yields merely $\neg\neg X$. If we do not care about this distinction we are free to ignore it, but there is no harm in having it around.

In the 1930's, this idea was made precise by Gödel [46] and Gentzen [43]. They showed that we can embed classical logic in intuitionistic logic. In fact, they found a map sending any formula X of the propositional calculus to a new formula X°, such that X is provable classically if and only if X° is provable intuitionistically. (More impressively, this map also works for the predicate calculus.)

Later, yet another reason for being interested in intuitionistic logic became apparent: its connection to category theory. In its very simplest form, this connection works as follows. Suppose we have a set of propositions $X, Y, Z, \ldots$ obeying the laws of the intuitionistic propositional calculus. We can create a category C where these propositions are objects and there is at most one morphism from any object X to any object Y: a single morphism when X implies Y, and none otherwise!

A category with at most one morphism from any object to any other is called a **preorder**. In the propositional calculus, we often treat two propositions as equal when they both imply each other. If we do this, we get a special sort of preorder: one where isomorphic objects are automatically equal. This special sort of preorder is called a **partially ordered set**, or **poset** for short. Posets abound in logic, precisely because they offer a simple framework for understanding implication.

If we start from a set of propositions obeying the intuitionistic propositional calculus, the resulting category C is better than a mere poset. It is also cartesian, with $X \wedge Y$ as the product of X and Y, and $\top$ as the terminal object! To see this, note that any proposition Q has a unique morphism to $X \wedge Y$ whenever it has morphisms to X and to Y. This is simply a fancy way of saying that Q implies $X \wedge Y$ when it implies X and implies Y. It is also easy to see that $\top$ is terminal: anything implies the truth.

Even better, the category C is cartesian closed, with $X \Rightarrow Y$ as the internal hom. The reason is that

$$X \wedge Y \text{ implies } Z \quad \text{iff} \quad Y \text{ implies } X \Rightarrow Z.$$

This automatically yields the basic property of the internal hom:

$$\hom(X \otimes Y, Z) \cong \hom(Y, X \multimap Z).$$

Indeed, if the reader is puzzled by the difference between 'X implies Y' and $X \Rightarrow Y$, we can now explain this more clearly: the former involves the homset $\hom(X, Y)$ (which has one element when X implies Y and none otherwise), while the latter is the internal hom, an object in C.

So, C is a cartesian closed poset. But, it also has one more nice property, thanks to the presence of $\vee$ and $\bot$ We have seen that $\wedge$ and $\top$ make the category C cartesian; $\vee$ and $\bot$ satisfy exactly analogous rules, but with the implications turned around, so they make C^{op} cartesian.

And that is all! In particular, negation gives nothing more, since we can define $\neg X$ to be $X \Rightarrow F$, and all its intuitionistically valid properties then follow. So, the kind of category we get from the intuitionistic propositional calculus by taking propositions as objects and implications as morphisms is precisely a **Heyting algebra**: a cartesian closed poset C such that C^{op} is also cartesian.

Heyting, a student of Brouwer, introduced Heyting algebras in intuitionistic logic before categories were even invented. So, he used very different language to define them. But, the category-theoretic approach to Heyting algebras illustrates the connection between cartesian closed categories and logic. It also gives more evidence that dropping the law of excluded middle is an interesting thing to try.

Since we have explained the basics of cartesian closed categories, but not said what happens when the *opposite* of such a category is *also* cartesian, in the sections to come we will take a drastic step and limit our discussion of logic even further. We will neglect $\vee$ and $\bot$, and concentrate only on the fragment of the propositional calculus involving $\wedge$, $\top$ and $\Rightarrow$.

Even here, it turns out, there are interesting things to say — and interesting ways to modify the usual rules. This will be the main subject of the sections to come. But to set the stage, we need to say a bit about proof theory.

Proof theory is the branch of mathematical logic that treats proofs as mathematical entities worthy of study in their own right. It lets us dig deeper into the propositional calculus, by studying not merely *whether or not* some assumption X implies some conclusion Y, but the whole *set of proofs* leading from X to Y. This amounts to studying not just posets (or preorders), but categories that allow many morphisms from one object to another.

In Hilbert's approach to proof, there were many axioms and just one rule to deduce new theorems: *modus ponens*, which says that from X and 'X implies Y' we can deduce Y. Most of modern proof theory focuses on another approach, the 'sequent calculus', due to Gentzen [43]. In this approach there are few axioms but many inference rules.

An excellent introduction to the sequent calculus is the book *Proofs and Types* by Girard, Lafont and Taylor, freely available online [45]. Here we shall content ourselves with some sketchy remarks. A 'sequent' is something like this:

$$X_1, \ldots, X_m \vdash Y_1, \ldots, Y_n$$

where X_i and Y_i are propositions. We read this sequent as saying that *all* the propositions X_i, taken together, can be used to prove at least *one* of the propositions Y_i. This strange-sounding convention gives the sequent calculus a nice symmetry, as we shall soon see.

In the sequent calculus, an 'inference rule' is something that produces new sequents from old. For example, here is the **left weakening** rule:

$$\frac{X_1, \ldots, X_m \vdash Y_1, \ldots, Y_n}{X_1, \ldots, X_m, A \vdash Y_1, \ldots, Y_n}$$

This says that from the sequent above the line we can get the sequent below the line: we can throw in the extra assumption A without harm. Thanks to the strange-sounding convention we mentioned, this rule has a mirror-image version called **right weakening**:

$$\frac{X_1, \ldots, X_m \vdash Y_1, \ldots, Y_n}{X_1, \ldots, X_m \vdash Y_1, \ldots, Y_n, A}$$

In fact, Gentzen's whole setup has this mirror symmetry! For example, his rule called **left contraction**:

$$\frac{X_1, \ldots, X_m, A, A \vdash Y_1, \ldots, Y_n}{X_1, \ldots, X_m, A \vdash Y_1, \ldots, Y_n}$$

has a mirror partner called **right contraction**:

$$\frac{X_1, \ldots, X_m \vdash Y_1, \ldots, Y_n, A, A}{X_1, \ldots, X_m \vdash Y_1, \ldots, Y_n, A}$$

Similarly, this rule for 'and'

$$\frac{X_1, \ldots, X_m, A \vdash Y_1, \ldots, Y_n}{X_1, \ldots, X_m, A \wedge B \vdash Y_1, \ldots, Y_n}$$

has a mirror partner for 'or':

$$\frac{X_1, \ldots, X_m, \vdash Y_1, \ldots, Y_n, A}{X_1, \ldots, X_m \vdash Y_1, \ldots, Y_n, A \vee B}$$

Logicians now realize that this mirror symmetry arises from the duality between a category and its opposite. Unfortunately, since we have decided to study $\wedge$ and $\top$ but not their mirror partners $\vee$ and $\bot$, this duality will not be visible in the sections to come.

Gentzen used sequents to write inference rules for the classical propositional calculus, and also the classical predicate calculus. Now, in these forms of logic we have

$$X_1, \ldots, X_m \vdash Y_1, \ldots, Y_n$$

if and only if we have

$$X_1 \wedge \cdots \wedge X_m \vdash Y_1 \vee \cdots \vee Y_n.$$

So, why did Gentzen use sequents with a *list* of propositions on each side of the $\vdash$ symbol, instead just a single proposition? The reason is that this let him use only inference rules having the 'subformula property'. This says that every proposition in the sequent below the line appears as part of some proposition in the sequent above the line. So, a proof built from such inference rules becomes a 'tree' where all the propositions further up the tree are subformulas of those below.

This idea has powerful consequences. For example, in 1936 Gentzen was able prove the consistency of Peano's axioms of arithmetic! His proof essentially used induction on trees (Readers familiar with Gödel's second incompleteness theorem should be reassured that this sort of induction cannot itself be carried out in Peano arithmetic.)

The most famous rule *lacking* the subformula property is the 'cut rule':

$$\frac{X_1, \ldots, X_m \vdash Y_1, \ldots, Y_k, A \qquad X_{m+1}, \ldots, X_n, A \vdash Y_{k+1}, \ldots, Y_\ell}{X_1, \ldots, X_n \vdash Y_1, \ldots, Y_\ell}$$

From the two sequents on top, the cut rule gives us the sequent below. Note that the intermediate step A does not appear in the sequent below. It is 'cut out'. So, the cut rule lacks the subformula property. But, one of Gentzen's great achievements was to show that any proof in the classical propositional (or even predicate) calculus that can be done *with* the cut rule can also be done *without* it. This is called 'cut elimination'.

Gentzen also wrote down inference rules suitable for the intuitionistic propositional and predicate calculi. These rules lack the mirror symmetry of the classical case. But in the 1980s, this symmetry was restored by Girard's invention of 'linear logic' [44].

Linear logic lets us keep track of how many times we use a given premise to reach a given conclusion. To accomplish this, Girard introduced some new logical connectives! For starters, he introduced 'linear' connectives called $\otimes$ and $\multimap$, and a logical constant called I. These act a bit like $\wedge$, $\Rightarrow$ and $\top$. However, they satisfy rules corresponding to a symmetric monoidal category instead of a cartesian closed category. In particular, from X we can prove neither $X \otimes X$ nor I. So, we cannot freely 'duplicate' and 'delete' propositions using these new connectives. This is reflected in the fact that linear logic drops Gentzen's contraction and weakening rules.

By itself, this might seem unbearably restrictive. However, Girard *also* kept the connectives $\wedge$, $\Rightarrow$ and $\top$ in his system, still satisfying the usual rules. And, he introduced an operation called the 'exponential', !, which takes a proposition X and turns it into an 'arbitrary stock of copies of X'. So, for example, from $!X$ we can prove 1, and X, and $X \otimes X$, and $X \otimes X \otimes X$, and so on.

Full-fledged linear logic has even more connectives than we have described here. It seems baroque and peculiar at first glance. It also comes in both classical and intuitionistic versions! But, *just as classical logic can be embedded in intuitionistic logic, intuitionistic*

logic can be embedded in intuitionistic linear logic [44]. So, we do not lose any deductive power. Instead, we gain the ability to make even more fine-grained distinctions.

In what follows, we discuss the fragment of intuitionistic linear logic involving only $\otimes, \multimap$ and I. This is called 'multiplicative intuititionistic linear logic' [49, 80]. It turns out to be the system of logic suitable for closed symmetric monoidal categories — nothing more or less.

1.3.2 Proofs as Morphisms

In Section 1.2 we described categories with various amounts of extra structure, starting from categories pure and simple, and working our way up to monoidal categories, braided monoidal categories, symmetric monoidal categories, and so on. Our treatment only scratched the surface of an enormously rich taxonomy. In fact, each kind of category with extra structure corresponds to a system of logic with its own inference rules!

To see this, we will think of *propositions* as *objects* in some category, and *proofs* as giving *morphisms*. Suppose X and Y are propositions. Then, we can think of a proof starting from the assumption X and leading to the conclusion Y as giving a morphism $f\colon X \to Y$. (In Section 1.3.3 we shall see that a morphism is actually an equivalence class of proofs — but for now let us gloss over this issue.)

Let us write $X \vdash Y$ when, starting from the assumption X, there is a proof leading to the conclusion Y. An inference rule is a way to get new proofs from old. For example, in almost every system of logic, if there is a proof leading from X to Y, and a proof leading from Y to Z, then there is a proof leading from X to Z. We write this inference rule as follows:

$$\frac{X \vdash Y \qquad Y \vdash Z}{X \vdash Z}$$

We can call this **cut rule**, since it lets us 'cut out' the intermediate step Y. It is a special case of Gentzen's cut rule, mentioned in the previous section. It should remind us of composition of morphisms in a category: if we have a morphism $f\colon X \to Y$ and a morphism $g\colon Y \to Z$, we get a morphism $gf\colon X \to Z$.

Also, in almost every system of logic there is a proof leading from X to X. We can write this as an inference rule that starts with *nothing* and concludes the existence of a proof of X from X:

$$\frac{}{X \vdash X}$$

This rule should remind us of how every object in category has an identity morphism: for any object X, we automatically get a morphism $1_X\colon X \to X$. Indeed, this rule is sometimes called the **identity rule**.

If we pursue this line of thought, we can take the definition of a closed symmetric monoidal category and extract a collection of inference rules. Each rule is a way to get new morphisms from old in a closed symmetric monoidal category. There are various superficially different but ultimately equivalent ways to list these rules. Here is one:

$$\frac{}{X \vdash X}\ (\mathrm{i}) \qquad \frac{X \vdash Y \qquad Y \vdash Z}{X \vdash Z}\ (\circ)$$

$$\frac{W \vdash X \qquad Y \vdash Z}{W \otimes Y \vdash X \otimes Z}\ (\otimes) \qquad \frac{W \vdash (X \otimes Y) \otimes Z}{W \vdash X \otimes (Y \otimes Z)}\ (\mathrm{a})$$

$$\frac{X \vdash I \otimes Y}{X \vdash Y}\ (\mathrm{l}) \qquad \frac{X \vdash Y \otimes I}{X \vdash Y}\ (\mathrm{r})$$

$$\frac{W \vdash X \otimes Y}{W \vdash Y \otimes X}\ (\mathrm{b}) \qquad \frac{X \otimes Y \vdash Z}{Y \vdash X \multimap Z}\ (\mathrm{c})$$

Double lines mean that the inverse rule also holds. We have given each rule a name, written to the right in parentheses. As already explained, rules (i) and ($\circ$) come from the presence of identity morphisms and composition in any category. Rules ($\otimes$), (a), (l), and (r) come from tensoring, the associator, and the left and right unitors in a monoidal category. Rule (b) comes from the braiding in a braided monoidal category, and rule (c) comes from currying in a closed monoidal category.

Now for the big question: *what does all this mean in terms of logic?* These rules describe a small fragment of the propositional calculus. To see this, we should read the connective $\otimes$ as 'and', the connective $\multimap$ as 'implies', and the proposition I as 'true'.

In this interpretation, rule (c) says we can turn a proof leading from the assumption 'Y and X' to the conclusion Z into a proof leading from X to 'Y implies Z'. It also says we can do the reverse. This is true in classical, intuitionistic and linear logic, and so are all the other rules. Rules (a) and (b) say that 'and' is associative and commutative. Rule (l) says that any proof leading from the assumption X to the conclusion 'true and Y' can be converted to a proof leading from X to Y, and vice versa. Rule (r) is similar.

What do we do with these rules? We use them to build 'deductions'. Here is an easy example:

$$\cfrac{\cfrac{}{X \multimap Y \vdash X \multimap Y}\ (\mathrm{i})}{X \otimes (X \multimap Y) \vdash Y}\ (\mathrm{c}^{-1})$$

First we use the identity rule, and then the inverse of the currying rule. At the end, we obtain

$$X \otimes (X \multimap Y) \vdash Y.$$

This should remind us of the evaluation morphisms we have in a closed monoidal category:

$$\mathrm{ev}_{X,Y} \colon X \otimes (X \multimap Y) \to Y.$$

In terms of logic, the point is that we can prove Y from X and 'X implies Y'. This fact comes in handy so often that we may wish to abbreviate the above deduction as an extra inference rule — a rule derived from our basic list:

$$\frac{}{X \otimes (X \multimap Y) \vdash Y}\ (\mathrm{ev})$$

This rule is called **modus ponens**.

In general, a deduction is a tree built from inference rules. Branches arise when we use the $(\circ)$ or $(\otimes)$ rules. Here is an example:

$$\dfrac{\dfrac{\dfrac{}{(A \otimes B) \otimes C \vdash ((A \otimes B) \otimes C}\,(\mathrm{i})}{(A \otimes B) \otimes C \vdash A \otimes (B \otimes C)}\,(\mathrm{a}) \qquad A \otimes (B \otimes C) \vdash D}{(A \otimes B) \otimes C \vdash D}\,(\circ)$$

Again we can abbreviate this deduction as a derived rule. In fact, this rule is reversible:

$$\dfrac{A \otimes (B \otimes C) \vdash D}{(A \otimes B) \otimes C \vdash D}\,(\alpha)$$

For a more substantial example, suppose we want to show

$$(X \multimap Y) \otimes (Y \multimap Z) \vdash X \multimap Z.$$

The deduction leading to this will not even fit on the page unless we use our abbreviations:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{X \otimes (X \multimap Y) \vdash Y}\,(\mathrm{ev}) \qquad \dfrac{}{Y \multimap Z \vdash Y \multimap Z}\,(\mathrm{id})}{(X \otimes (X \multimap Y)) \otimes (Y \multimap Z) \vdash Y \otimes (Y \multimap Z)}\,(\otimes) \qquad \dfrac{}{Y \otimes (Y \multimap Z) \vdash Z}\,(\mathrm{ev})}{(X \otimes (X \multimap Y)) \otimes (Y \multimap Z) \vdash Z}}{X \otimes ((X \multimap Y) \otimes (Y \multimap Z)) \vdash Z}\,(\alpha^{-1})}{(X \multimap Y) \otimes (Y \multimap Z) \vdash X \multimap Z}\,(\mathrm{c})$$

Since each of the rules used in this deduction came from a way to get new morphisms from old in a closed monoidal category (we never used the braiding), it follows that in every such category we have **internal composition** morphisms:

$$\bullet_{X,Y,Z} : (X \multimap Y) \otimes (Y \multimap Z) \to X \multimap Z.$$

These play the same role for the internal hom that ordinary composition

$$\circ \colon \hom(X, Y) \times \hom(Y, Z) \to \hom(X, Z)$$

plays for the ordinary hom.

We can go ahead making further deductions in this system of logic, but the really interesting thing is what it omits. For starters, it omits the connective 'or' and the proposition 'false'. It also omits two inference rules we normally take for granted — namely, **contraction**:

$$\dfrac{X \vdash Y}{X \vdash Y \otimes Y}\,(\Delta)$$

and **weakening**:

$$\frac{X \vdash Y}{X \vdash I} \; (!)$$

which are closely related to duplication and deletion in a cartesian category. Omitting these rules is a distinctive feature of linear logic [44]. The word 'linear' should remind us of the category Hilb. As noted in Section 1.2.3, this category with its usual tensor product is noncartesian, so it does not permit duplication and deletion. But, what does omitting these rules mean *in terms of logic?*

Ordinary logic deals with propositions, so we have been thinking of the above system of logic in the same way. Linear logic deals not just with propositions, but also other resources — for example, physical things! Unlike propositions in ordinary logic, we typically can't duplicate or delete these other resources. In classical logic, if we know that a proposition X is true, we can use X as many or as few times as we like when trying to prove some proposition Y. But if we have a cup of milk, we can't use it to make cake and then use it again to make butter. Nor can we make it disappear without a trace: even if we pour it down the drain, it must go somewhere.

In fact, these ideas are familiar in chemistry. Consider the following resources:

$$\begin{aligned} H_2 &= \text{one molecule of hydrogen} \\ O_2 &= \text{one molecule of oxygen} \\ H_2O &= \text{one molecule of water} \end{aligned}$$

We can burn hydrogen, combining one molecule of oxygen with two of hydrogen to obtain two molecules of water. A category theorist might describe this reaction as a morphism:

$$f \colon O_2 \otimes (H_2 \otimes H_2) \to H_2O \otimes H_2O.$$

A linear logician might write:

$$O_2 \otimes (H_2 \otimes H_2) \vdash H_2O \otimes H_2O$$

to indicate the existence of such a morphism. But, we cannot duplicate or delete molecules, so for example

$$H_2 \not\vdash H_2 \otimes H_2$$

and

$$H_2 \not\vdash I$$

where I is the unit for the tensor product: not iodine, but 'no molecules at all'.

In short, ordinary chemical reactions are morphisms in a symmetric monoidal category where objects are collections of molecules. As chemists normally conceive of it, this category is not closed. So, it obeys an even more limited system of logic than the one we have been discussing, a system lacking the connective $\multimap$. To get a closed category — in fact a compact one — we need to remember one of the great discoveries of 20th-century physics: *antimatter*. This lets us define $Y \multimap Z$ to be 'anti-Y and Z':

$$Y \multimap Z = Y^* \otimes Z$$

Then the currying rule holds:

$$\frac{Y \otimes X \vdash Z}{X \vdash Y^* \otimes Z}$$

Most chemists don't think about antimatter very often — but particle physicists do. They don't use the notation of linear logic or category theory, but they know perfectly well that since a neutrino and a neutron can collide and turn into a proton and an electron:

$$\nu \otimes n \vdash p \otimes e,$$

then a neutron can turn into a neutrino together with a proton and an electron:

$$n \vdash \nu^* \otimes (p \otimes e).$$

This is an instance of the currying rule, rule (c).

1.3.3 Logical Theories from Categories

We have sketched how different systems of logic naturally arise from different types of categories. To illustrate this idea, we introduced a system of logic with inference rules coming from ways to get new morphisms from old in a *closed symmetric monoidal category*. One could substitute many other types of categories here, and get other systems of logic.

To tighten the connection between proof theory and category theory, we shall now describe a recipe to get a logical theory from any closed symmetric monoidal category. For this, we shall now use $X \vdash Y$ to denote the *set* of proofs — or actually, equivalence classes of proofs — leading from the assumption X to the conclusion Y. This is a change of viewpoint. Previously we would write $X \vdash Y$ when this set of proofs was nonempty; otherwise we would write $X \not\vdash Y$. The advantage of treating $X \vdash Y$ as a set is that this set is precisely what a category theorist would call $\hom(X, Y)$: a homset in a category.

If we let $X \vdash Y$ stand for a homset, an inference rule becomes a function from a product of homsets to a single homset. For example, the cut rule

$$\frac{X \vdash Y \qquad Y \vdash Z}{X \vdash Z} \; (\circ)$$

becomes another way of talking about the composition function

$$\circ_{X,Y,Z} \colon \hom(X, Y) \times \hom(Y, Z) \to \hom(X, Z),$$

while the identity rule

$$\frac{}{X \vdash X} \; (\mathrm{i})$$

becomes another way of talking about the function

$$\mathrm{i}_X \colon 1 \to \hom(X, X)$$

that sends the single element of the set 1 to the identity morphism of X. (Note: the set 1 is a *zero-fold* product of homsets.)

Next, if we let inference rules be certain functions from products of homsets to homsets, deductions become more complicated functions of the same sort built from these basic ones. For example, this deduction:

$$\dfrac{\dfrac{\dfrac{}{X \otimes I \vdash X \otimes I}\,(\mathrm{i})}{X \otimes I \vdash X}\,(\mathrm{r}) \qquad \dfrac{}{Y \vdash Y}\,(\mathrm{i})}{(X \otimes I) \otimes Y \vdash X \otimes Y}\,(\otimes)$$

specifies a function from 1 to $\hom((X \otimes I) \otimes Y, X \otimes Y)$, built from the basic functions indicated by the labels at each step. This deduction:

$$\dfrac{\dfrac{\dfrac{}{(X \otimes I) \otimes Y \vdash (X \otimes I) \otimes Y}\,(\mathrm{i})}{(X \otimes I) \otimes Y \vdash X \otimes (I \otimes Y)}\,(\mathrm{a}) \qquad \dfrac{\dfrac{\dfrac{}{I \otimes Y \vdash I \otimes Y}\,(\mathrm{i})}{I \otimes Y \vdash Y}\,(\mathrm{r}) \qquad \dfrac{}{X \vdash X}\,(\mathrm{i})}{X \otimes (I \otimes Y) \vdash X \otimes Y}\,(\otimes)}{(X \otimes I) \otimes Y \vdash X \otimes Y}\,(\circ)$$

gives another function from 1 to $\hom((X \otimes I) \otimes Y, X \otimes Y)$.

If we think of deductions as giving functions this way, the question arises when two such functions are equal. In the example just mentioned, the triangle equation in the definition of monoidal category (Definition 6):

$$\begin{array}{ccc} (X \otimes I) \otimes Y & \xrightarrow{a_{X,I,Y}} & X \otimes (I \otimes Y) \\ & {\scriptstyle r_X \otimes 1_Y} \searrow \quad \swarrow {\scriptstyle 1_X \otimes l_Y} & \\ & X \otimes Y & \end{array}$$

says these two functions *are* equal. Indeed, the triangle equation is precisely the statement that these two functions agree! (We leave this as an exercise for the reader.)

So: even though two deductions may look quite different, they may give the same function from a product of homsets to a homset if we demand that these are homsets in a closed symmetric monoidal category. This is why we think of $X \multimap Y$ as a set of *equivalence classes* of proofs, rather than proofs: it is forced on us by our desire to use category theory. We could get around this by using a 2-category with proofs as morphisms and 'equivalences between proofs' as 2-morphisms [82]. This would lead us further to the right in the Periodic Table (Table 1.3). But let us restrain ourselves and make some definitions formalizing what we have done so far.

From now on we shall call the objects $X, Y, \ldots$ 'propositions', even though we have seen they may represent more general resources. Also, purely for the sake of brevity, we use the term 'proof' to mean 'equivalence class of proofs'. The equivalence relation must be coarse enough to make the equations in the following definitions hold:

Definition 18. *A* **closed monoidal theory** *consists of the following:*

- *A collection of* **propositions**. *The collection must contain a proposition I, and if X and Y are propositions, then so are $X \otimes Y$ and $X \multimap Y$.*
- *For every pair of propositions X, Y, a set $X \vdash Y$ of* **proofs** *leading from X to Y. If $f \in X \vdash Y$, then we write $f \colon X \to Y$.*
- *Certain functions, written as* **inference rules**:

$$\frac{}{X \vdash X}\ (\mathrm{i}) \qquad \frac{X \vdash Y \qquad Y \vdash Z}{X \vdash Z}\ (\circ)$$

$$\frac{W \vdash X \qquad Y \vdash Z}{W \otimes Y \vdash X \otimes Z}\ (\otimes) \qquad \frac{W \vdash (X \otimes Y) \otimes Z}{W \vdash X \otimes (Y \otimes Z)}\ (\mathrm{a})$$

$$\frac{X \vdash I \otimes Y}{X \vdash Y}\ (\mathrm{l}) \qquad \frac{X \vdash Y \otimes I}{X \vdash Y}\ (\mathrm{r})$$

$$\frac{X \otimes Y \vdash Z}{Y \vdash X \multimap Z}\ (\mathrm{c})$$

  A double line means that the function is invertible. So, for example, for each triple X, Y, Z we have a function

$$\circ_{X,Y,Z} \colon (X \vdash Y) \times (Y \vdash Z) \ \to\ (X \vdash Z)$$

  and a bijection

$$c_{X,Y,Z} \colon (X \otimes Y \vdash Z) \ \to\ (Y \vdash X \multimap Z).$$

- *Certain equations that must be obeyed by the inference rules. The inference rules* $(\circ)$ *and* (i) *must obey equations describing associativity and the left and right unit laws. Rule* $(\otimes)$ *must obey an equation saying it is a functor. Rules* (a), (l), (r), *and* (c) *must obey equations saying they are natural transformations. Rules* (a), (l), (r) *and* $(\otimes)$ *must also obey the triangle and pentagon equations.*

Definition 19. *A* **closed braided monoidal theory** *is a closed monoidal theory with this additional inference rule:*

$$\frac{W \vdash X \otimes Y}{W \vdash Y \otimes X}\ (\mathrm{b})$$

We demand that this rule give a natural transformation satisfying the hexagon equations.

Definition 20. *A* **closed symmetric monoidal theory** *is a closed braided monoidal theory where the rule* (b) *is its own inverse.*

These are just the usual definitions of various kinds of closed category — monoidal, braided monoidal and symmetric monoidal — written in a new style. This new style lets us *build such categories from logical systems*. To do this, we take the objects to be propositions

and the morphisms to be equivalence classes of proofs, where the equivalence relation is generated by the equations listed in the definitions above.

However, the full advantages of this style only appear when we dig deeper into proof theory, and generalize the expressions we have been considering:

$$X \vdash Y$$

to 'sequents' like this:

$$X_1, \ldots, X_n \vdash Y.$$

Loosely, we can think of such a sequent as meaning

$$X_1 \otimes \cdots \otimes X_n \vdash Y.$$

The advantage of sequents is that they let us use inference rules that — except for the cut rule and the identity rule — have the 'subformula property' mentioned near the end of Section 1.3.1.

Formulated in terms of these inference rules, the logic of closed symmetric monoidal categories goes by the name of 'multiplicative intuitionistic linear logic', or MILL for short [49, 80]. There is a 'cut elimination' theorem for MILL, which says that with a suitable choice of other inference rules, the cut rule becomes redundant: any proof that can be done with it can be done without it. This is remarkable, since the cut rule corresponds to *composition of morphisms* in a category. One consequence is that in the free symmetric monoidal closed category on any set of objects, the set of morphisms between any two objects is *finite*. There is also a decision procedure to tell when two morphisms are equal. For details, see the book by Szabo [90] and Trimble's thesis [91]. For an alternative viewpoint, see Kelly and Mac Lane's coherence theorem for closed symmetric monoidal categories [58], and the related theorem for compact symmetric monoidal categories [59].

MILL is just one of many closely related systems of logic. Most include extra features, but some *subtract* features. Here are just a few examples:

- Algebraic theories. In his famous thesis, Lawvere [66] defined an **algebraic theory** to be a cartesian category where every object is an n-fold cartesian power $X \times \cdots \times X$ ($n \geq 0$) of a specific object X. He showed how such categories regarded as logical theories of a simple sort — the sort that had previously been studied in 'universal algebra' [24]. This work initiated the categorical approach to logic which we have been sketching here. Crole's book [33] gives a gentle introduction to algebraic theories as well as some richer logical systems. More generally, we can think of any cartesian category as a generalized algebraic theory.
- Intuitionistic linear logic (ILL). ILL supplements MILL with the operations familiar from intuitionistic logic, as well as an operation ! turning any proposition (or resource) X into an 'indefinite stock of copies of X'. Again there is a nice category-theoretic interpretation. Bierman's thesis [22] gives a good overview, including a proof of cut elimination for ILL and a proof of the result, originally due to Girard, that intuitionistic logic can be be embedded in ILL.

- Linear logic (LL). For full-fledged linear logic, the online review article by Di Cosmo and Miller [36] is a good place to start. For more, try the original paper by Girard [44] and the book by Troelstra [92]. Blute and Scott's review article [23] serves as a Rosetta Stone for linear logic and category theory, and so do the lectures notes by Schalk [80].
- Intuitionistic Logic (IL). Lambek and Scott's classic book [64] is still an excellent introduction to intuitionistic logic and cartesian closed categories. The online review article by Moschovakis [74] contains many suggestions for further reading.

To conclude, let us say precisely what an 'inference rule' amounts to in the setup we have described. We have said it gives a function from a product of homsets to a homset. While true, this is not the last word on the subject. After all, instead of treating the propositions appearing in an inference rule as *fixed*, we can treat them as *variable*. Then an inference rule is really a 'schema' for getting new proofs from old. How do we formalize this idea?

First we must realize that $X \vdash Y$ is not just a set: it is a set *depending in a functorial way* on X and Y. As noted in Definition 13, there is a functor, the 'hom functor'

$$\hom \colon C^{\mathrm{op}} \times C \to \mathrm{Set},$$

sending (X, Y) to the homset $\hom(X, Y) = X \vdash Y$. To look like logicians, let us write this functor as $\vdash$.

Viewed in this light, most of our inference rules are *natural transformations*. For example, rule (a) is a natural transformation between two functors from $C^{\mathrm{op}} \times C^3$ to Set, namely the functors

$$(W, X, Y, Z) \mapsto W \vdash (X \otimes Y) \otimes Z)$$

and

$$(W, X, Y, Z) \mapsto W \vdash X \otimes (Y \otimes Z).$$

This natural transformation turns any proof

$$f \colon W \to (X \otimes Y) \otimes Z)$$

into the proof

$$a_{X,Y,Z} f \colon W \to X \otimes (Y \otimes Z).$$

The fact that this transformation is *natural* means that it changes in a systematic way as we vary W, X, Y and Z. The commuting square in the definition of natural transformation, Definition 3, makes this precise.

Rules (l), (r), (b) and (c) give natural transformations in a very similar way. The $(\otimes)$ rule gives a natural transformation between two functors from $C^{\mathrm{op}} \times C \times C^{\mathrm{op}} \times C$ to Set, namely

$$(W, X, Y, Z) \mapsto (W \vdash X) \ \times \ (Y \vdash Z)$$

and

$$(W, X, Y, Z) \mapsto W \otimes Y \vdash X \otimes Z$$

This natural transformation sends any element $(f, g) \in \hom(W, X) \times \hom(Y, Z)$ to $f \otimes g$.

The identity and cut rules are different: they *do not* give natural transformations, because the top line of these rules has a different number of variables than the bottom line! Rule (i) says that for each $X \in C$ there is a function

$$\mathrm{i}_X \colon 1 \;\to\; X \vdash X$$

picking out the identity morphism 1_X. What would it mean for this to be natural in X? Rule ($\circ$) says that for each triple $X, Y, Z \in C$ there is a function

$$\circ \colon (X \vdash Y) \;\times\; (Y \vdash Z) \;\to\; X \vdash Z.$$

What would it mean for this to be natural in X, Y and Z? The answer to both questions involves a generalization of natural transformations called 'dinatural' transformations [68].

As noted in Definition 3, a natural transformation $\alpha \colon F \Rightarrow G$ between two functors $F, G \colon C \to D$ makes certain squares in D commute. If in fact $C = C_1^{\mathrm{op}} \times C_2$, then we actually obtain commuting cubes in D. Namely, the natural transformation α assigns to each object (X_1, X_2) a morphism α_{X_1,X_2} such that for any morphism $(f_1 \colon Y_1 \to X_1, f_2 \colon X_2 \to Y_2)$ in C, the cube shown in Figure 1.1 commutes.

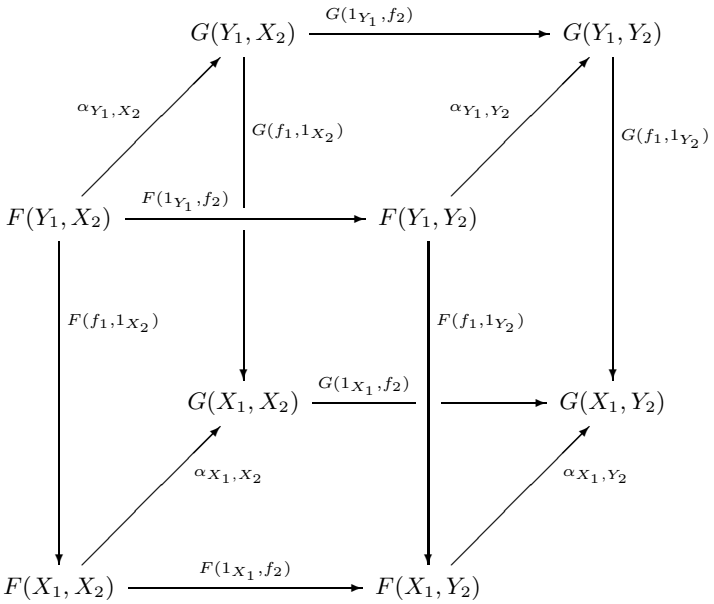


Fig. 1.1. A natural transformation between functors $F, G \colon C_1^{\mathrm{op}} \times C_2 \to \mathcal{D}$ gives a commuting cube in D for any morphisms $f_i \colon X_i \to Y_i$ in C_i.

If $C_1 = C_2$, we can choose a single object X and a single morphism $f \colon X \to Y$ and use it in both slots. As shown in Figure 1.2, there are then two paths from one corner of the

cube to the antipodal corner that only involve α for repeated arguments: that is, $\alpha_{X,X}$ and $\alpha_{Y,Y}$, but not $\alpha_{X,Y}$ or $\alpha_{Y,X}$. These paths give a commuting hexagon.

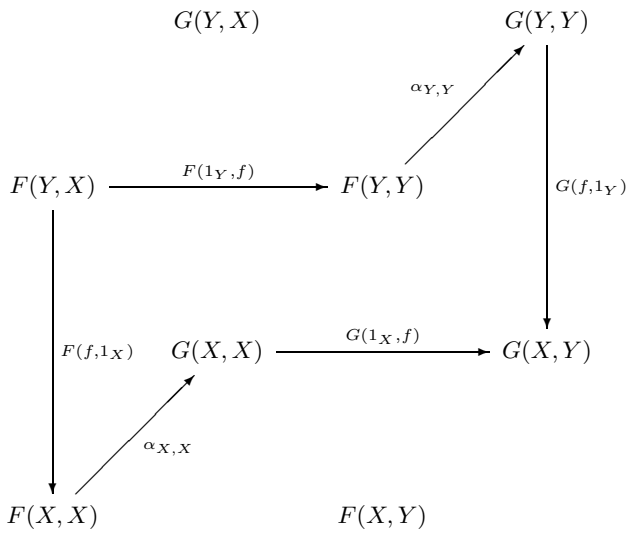


Fig. 1.2. A natural transformation between functors $F, G\colon C^{\mathrm{op}} \times C \to \mathcal{D}$ gives a commuting cube in D for any morphism $f\colon X \to Y$, and there are two paths around the cube that only involve α for repeated arguments.

This motivates the following:

Definition 21. *A* **dinatural transformation** $\alpha\colon F \Rightarrow G$ *between functors* $F, G\colon C^{\mathrm{op}} \times C \to D$ *assigns to every object* X *in* C *a morphism* $\alpha_X\colon F(X,X) \to G(X,X)$ *in* D *such that for every morphism* $f\colon X \to Y$ *in* C*, the hexagon in Figure 1.2 commutes.*

In the case of the identity rule, this commuting hexagon says that the identity morphism is a left and right unit for composition: see Figure 1.3. For the cut rule, this commuting hexagon says that composition is associative: see Figure 1.4.

So, in general, the sort of logical theory we are discussing involves:

- A *category* C of propositions and proofs.
- A *functor* $\vdash\colon C^{\mathrm{op}} \times C \to$ Set sending any pair of propositions to the set of proofs leading from one to the other.
- A set of *dinatural transformations* describing inference rules.

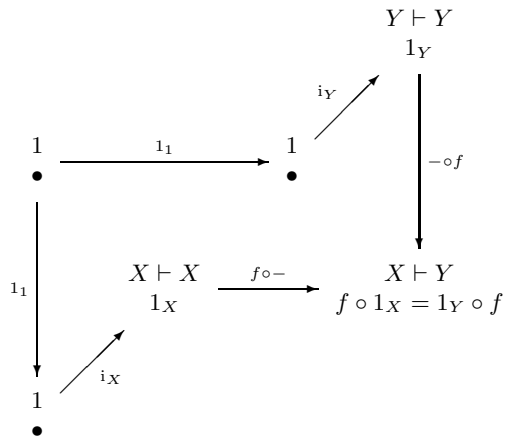


Fig. 1.3. Dinaturality of the (i) rule, where $f\colon X \to Y$. Here $\bullet \in 1$ denotes the one element of the one-element set.

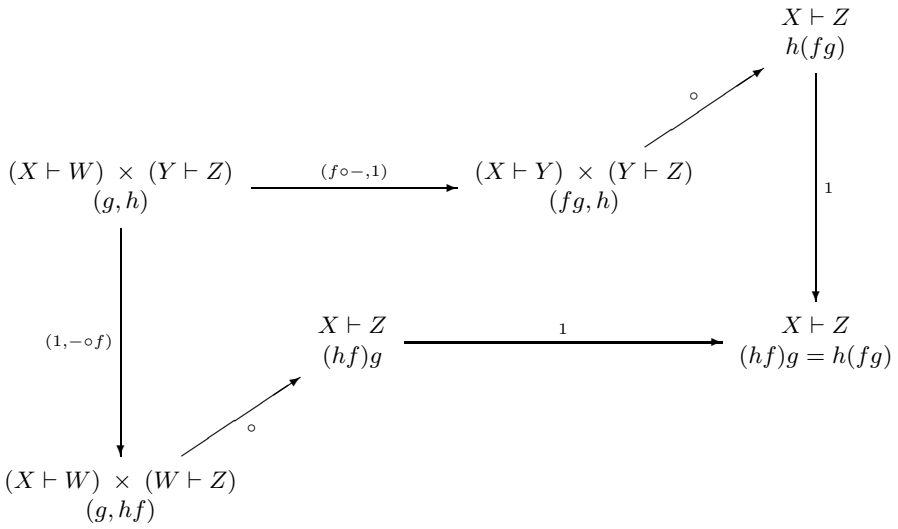


Fig. 1.4. Dinaturality of the cut rule, where $f\colon W \to Y$, $g\colon X \to W$, $h\colon Y \to Z$.

1.4 Computation

1.4.1 Background

In the 1930s, while Turing was developing what are now called 'Turing machines' as a model for computation, Church and his student Kleene were developing a different model, called the 'lambda calculus' [28, 60]. While a Turing machine can be seen as an idealized, simplified model of computer *hardware*, the lambda calculus is more like a simple model of *software*.

By now the are many careful treatments of the lambda calculus in the literature, from Barendregt's magesterial tome [15] to the classic category-theoretic treatment of Lambek and Scott [64], to Selinger's elegant online notes [84]. So, we shall content ourselves with a quick sketch.

Poetically speaking, the lambda calculus describes a universe where everything is a program and everything is data — *programs are data* — and we can take any program and apply it to any piece of data to get a new piece of data. More prosaically, everything is a 'λ-term', or 'term' for short. These are defined inductively:

- **Variables:** there is a countable set of 'variables', which are all terms.
- **Application:** if f and g are terms, we can 'apply' f to g and obtain a term (fg).
- **Lambda-abstraction:** if x is a variable and t is a term, there is a term $(\lambda x.t)$.

We think of $(\lambda x.t)$ as the program that, given x as input, returns t as output. For example, if x is a variable and f is a term, the term

$$(\lambda x.(fx))$$

stands for the program that, given x as input, returns (fx) as output. But this is just a fancy way of talking about f! So, the lambda calculus has a rule saying

$$(\lambda x.(fx)) = f.$$

But beware: this rule is not an equation in the usual mathematical sense. Instead, it is a 'rewrite rule': given the term on the left, we are allowed to rewrite it and get the term on the right. There are also other rewrite rules. Starting with a term and repeatedly applying rewrite rules is how we take a program and letting it run.

The lambda calculus is a very simple formalism. However, starting from just this, Church and Kleene were able to build up Boolean logic, the natural numbers and their usual operations, and so on. For example, they defined 'Church numerals' as follows:

$$\begin{aligned}
\overline{0} &= (\lambda x.(\lambda y.y)) \\
\overline{1} &= (\lambda x.(\lambda y.(xy))) \\
\overline{2} &= (\lambda x.(\lambda y.(x(xy)))) \\
\overline{3} &= (\lambda x.(\lambda y.(x(x(xy)))))
\end{aligned}$$

and so on. Thus, the Church numeral $\overline{n}$ is the program that 'takes any program to the nth power': if you give it any program x as input, it returns the program that applies x n times to whatever input y it receives. This makes it easy to define terms for addition, multiplication, and so on, and recover their usual properties. With more cleverness, Church and Kleene were able to write terms corresponding to more complicated functions. They eventually came to believe that *all* computable functions $f\colon \mathbb{N} \to \mathbb{N}$ can be defined in the lambda calculus.

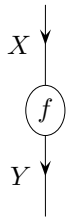
Meanwhile, Gödel was developing another approach to computability, the theory of 'recursive functions'. Around 1936, Kleene proved that the functions definable in the lambda calculus were the same as Gödel's recursive functions. In 1937 Turing described his 'Turing machines', and used these to give yet another definition of computable functions. This definition was later shown to agree with the other two. Thanks to this and other evidence, it is now widely accepted that the lambda calculus can define *any* function that can be computed by *any* systematic method. We say it is 'Turing complete'.

It took a while for computer scientists to profit from Church and Kleene's insights. However, in 1965 Landin pointed out a powerful analogy between the lambda calculus and the programming language ALGOL [65]. Landin's paper was very influential. It led to a renewed burst of work on the lambda calculus which continues to this day. By now, a number of computer languages are explicitly based on ideas from the lambda calculus. The most famous of these include Lisp, ML and Haskell. These languages, called 'functional programming languages', are beloved by theoretical computer scientists for their conceptual clarity. In fact, for many years, everyone majoring in computer science at MIT has been required to take an introductory course that involves programming in Scheme, a dialect of LISP. The cover of the textbook for this course [1] even has a big λ on the cover!

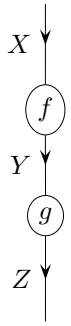
Languages of a different sort — 'imperative programming languages' — are more popular among working programmers. Examples include FORTRAN, BASIC, and C. In imperative programming, a program consists of statements that tell the computer what to do. In functional programming, a program describes a function, and running it evaluates this function.

Here we are mainly interested in 'typed' functional programming languages. These are more regimented than the original lambda calculus. In the original lambda calculus, any term can be applied to any other. In real-world programming, such an unstructured framework easily leads to mistakes. It is better to treat data as coming in various 'types', such as integers, floating-point numbers, alphanumeric strings, and so on. Then, we can demand that each program only accept input of a specified type and produce output of a specified type.

This idea is formalized by the 'typed' lambda calculus, where every term has a type. It also corresponds to a basic idea in category theory, where every morphism is like a black box with input and output wires of specified types:



and it makes no sense to hook two black boxes together unless the output of the first has the same type as the input of the next:



However, in the lambda calculus the basic operation is not *composition*, but *application*. As we shall see, this fits nicely into the framework of *closed* categories.

Indeed, in 1980 Lambek found a way to interpret the typed lambda calculus in terms of cartesian closed categories [63]. This idea led to a productive line of research blending category theory and computer science. There is no way we can summarize the resulting enormous body of work, though it constitutes a crucial aspect of the Rosetta Stone. Two good starting points for further reading are the textbook by Crole [33] and the online review article by Scott [78].

Our goal is more limited: we want to explain how morphisms in a closed symmetric monoidal category can be seen as programs written in a very limited sort of programming language. Unlike the lambda calculus, this language forbids duplication and deletion of data except when expressly permitted. The reason is that while every object in a cartesian category comes equipped with 'duplication' morphisms, a monoidal category typically lacks these. As we saw in Section 1.2.3, a great example is the category Hilb with its usual tensor product. So, in quantum computation we cannot freely duplicate or delete data. This makes it interesting to envisage programming languages with this constraint built in.

Various flavors of 'linear' or 'quantum' lambda calculus have already been studied, for example by Benton, Bierman de Paiva and Hyland [20], Dorca and van Tonder [94], and Selinger [84]. In the sections that follow we shall take a slightly different tack and consider a linear version, not of the lambda calculus, but of 'combinatory logic'.

Combinatory logic was born in a 1924 paper by Schönfinkel [81], and extensively developed by Curry [34]. In retrospect, we can see it as a stripped-down version of the lambda

calculus that completely avoids the use of variables. Starting from a basic stock of terms called 'combinators', the only way to build new ones is application: we can apply any term a to any term b and get a term (ab).

To build a Turing-complete programming language in such a impoverished setup, we need a sufficient stock of combinators. In fact, three are enough. The first, called I, acts like the identity, since it comes with the rewrite rule:

$$(Ia) = a$$

for every term a. The second, called K, gives a constant function (Ka) for each term a. In other words, it comes with a rewrite rule saying

$$(Ka)b = a.$$

for every term b. The third, called S, is the tricky one. It takes three terms, applies the first to the third, and applies the result to the second applied to the third:

$$(((Sa)b)c) = ((ac)(bc)).$$

As an illustration of how these rules work, let us apply $((SK)K)$ to any term a:

$$(((SK)K)a) = ((Ka)(Ka)) = a.$$

This is the same as (Ia). So, we say $((SK)K)$ and I are 'extensionally equivalent'. This means that I is redundant! We included it just to make this point. Or, consider $((SI)I)$:

$$(((SI)I)a) = ((Ia)(Ia)) = (aa)$$

So, $((SI)I)$ takes any term a and applies it to itself. This is a very powerful feature, but it leads to an infinite loop when we apply $((SI)I)$ to itself, since we get... nothing but $((SI)I))$ applied to itself! We can avoid infinite loops in a 'typed' combinatory logic, but there is a price to pay, since any Turing complete programming langage must allow nonterminating programs. A good compromise is to use a typed system with extra features that permit looping.

We can embed combinatory logic into the untyped lambda calculus by defining I, K, and S as follows:

$$\begin{aligned} I &= (\lambda x.x) \\ K &= (\lambda x.(\lambda y.x)) \\ S &= (\lambda x.(\lambda y.(\lambda z.((xz)(yz))))). \end{aligned}$$

The rewrite rules for these combinators then follow from rewrite rules in the lambda calculus. The more suprising fact is that any function computable using the lambda calculus can also be computed using just I, K and S!

We can make this a bit more precise as follows. All the variables in the lambda calculus formulas for I, K, and S are dummy variables. More generally, in the lambda calculus we define a 'combinator' to be a term in which all variables are dummy variables. Two

combinators c and d are 'extensionally equivalent' if they give the same result on any input: that is, for any term t, we can apply lambda calculus rewrite rules to (ct) and (dt) in a way that leads to the same term. It is then a theorem that any combinator in the lambda calculus is extensionally equivalent to one built from I, K, and S using just application.

In the sections to come, we shall describe a linear version of typed combinatory logic, suitable for closed symmetric monoidal categories. It is a bit irksome to avoid duplication or deletion of data in the lambda calculus, since we need to count how many times each variable gets used. It is easier in combinatory logic: we just need to avoid combinators that duplicate or delete data. For example, the combinator I is okay:

$$(Ia) = a.$$

The combinator K is not, since it deletes b here:

$$((Ka)b) = a.$$

The combinator S is not okay either, since it duplicates c here:

$$(((Sa)b)c) = ((ac)(bc)).$$

Luckily, we can read off a list of acceptable combinators directly from the definition of 'closed symmetric monoidal category'. The resulting system is not very powerful — but it will let us show how all the key concepts from previous sections have analogues in the world of computation.

1.4.2 Categories in Functional Programming

In functional programming, the fundamental operation is *application*. Mathematicians apply a function f to an argument x and write the result as $f(x)$; in functional programming we apply a program `f` to a piece of data `x` and write the result as `(f x)`.

To actually evaluate expressions like `(f x)`, we need 'rewrite rules'. For example, suppose we have programs `double` and `increment`, which compute the obvious functions from integers to integers. Then we might have rewrite rules

```
(increment 3) = 4
```

and

```
(double 4) = 8
```

We can simplify more complex expressions by repeatedly using these rules:

```
(double (increment 3)) = (double 4) = 8
```

We can handle functions that take more than one argument using a trick discussed in Section 1.2.6: 'currying'. This turns a function of several arguments into a function that takes the first argument and returns a function of the remaining arguments. So, for example, we could have a program `plus` that adds integers as follows:

```
((plus 3) 5) = 8
```

where `(plus 3)` stands for a program that adds 3 to its input.

With currying, we can define `increment` and `double` in terms of addition and multiplication. In other words, we can have rewrite rules

```
increment = (plus 1)
double = (times 2)
```

Then, for example, we have

```
(double (increment 3))
= (double ((plus 1) 3))
= ((times 2) ((plus 1) 3))
= ((times 2) 4)
= 8
```

So far we have focused on programs that take integers as input. In reality, programming involves many other kinds of data. For example, suppose we are writing a program that also involves days of the week. It would not make sense to write `(double Tuesday)`, because Tuesday is not a number. We might choose to represent Tuesday by a number in some program, but doubling that number doesn't have a good interpretation: is the first day of the week Sunday or Monday? Is the week indexed from zero or one? These are arbitrary choices that affect the result, so the expression `(double Tuesday)` has no well-defined meaning.

We can avoid ill-defined expressions of this sort by 'type checking.' To do this, every piece of data should have a 'type,' and our type checker (usually part of a compiler) should complain if we try to apply a function to a piece of data of the wrong type. For example, the number 248 is an integer, while `Tueday` is a day of the week; in programming we denote this by something like:

```
248:integer
Tuesday:day
```

(This notation is used in Ada, Pascal and some other languages. Other notations are also in widespread use.)

Given some types, we can build up other types in a variety of ways. For example, given types `X` and `Y`, there is a new type for functions that take data of type `X` as input and return data of type `Y` as output. This sort of type is called a **function type**. In the computer science literature, it would be denoted `X -> Y`. However, to be consistent with the rest of

our paper, we will write this function type as `X ⊸ Y`. The functions `increment` and `double` map integers to integers, so we write

```
increment:integer ⊸ integer
double:integer ⊸ integer
```

Addition and multiplication both have two inputs and one output:

```
plus:integer ⊸ (integer ⊸ integer)
times:integer ⊸ (integer ⊸ integer)
```

(The parentheses here don't stand for application; they are used for grouping.)

In fact, everything we have done so far can be formalized in terms of a closed monoidal category. Given such a category, we can call its objects **data types**. We can call a morphism $f\colon X \to Y$ a **program** that takes data of type X as input and produces data of type Y as output. The simplest sort of program is a piece of data: we call $x\colon I \to X$ is a **datum** of type X, and indicate this by writing `x:X`.

So, when we wrote `Tuesday:day` above, we can think of this as another way of saying

$$\text{Tuesday}\colon I \to \text{day}.$$

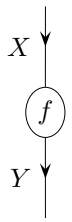
This funny-looking morphism has an easy interpretation in the category Set: here I would be the one-element set, 'day' would be the 7-element set consisting of days of the week, and 'Tuesday' would be the function picking out this particular day of the week. But, we could equally well be working in some category of data types and programs, and that is the interpretation we are trying to stress here.

We explained in Section 1.2.6 that in a closed monoidal category, any morphism $f\colon X \to Y$ has a 'name':

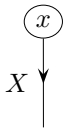
$$\ulcorner f \urcorner \colon I \to X \multimap Y.$$

Conversely, any morphism $g\colon I \to X \multimap Y$ is the name of a unique morphism $f\colon X \to Y$. This makes precise the idea that 'programs are data'. Namely: any program that takes data of type `X` as input and outputs data of type `Y` can be reinterpreted as a datum of type `X ⊸ Y`, and conversely.

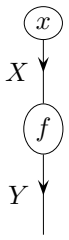
In functional programming, we use this fact to systematically work with the names of morphisms rather than the morphisms themselves. This is why 'application' becomes more fundamental than 'composition'. For example, suppose we have a program:



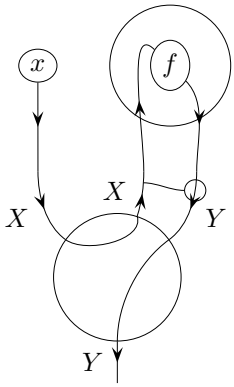
and this datum `x` of type `X`:



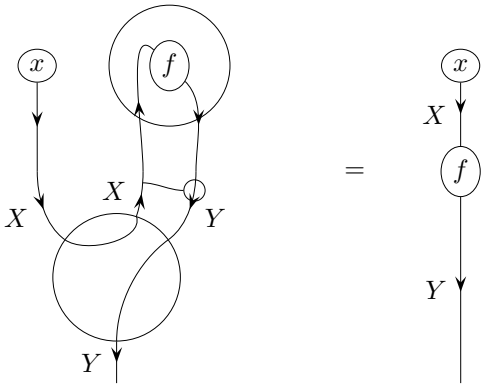
In category theory, we can feed the datum `x` into the program by composing them:



In functional programming, we instead 'apply' the name of f to x, as follows:



However, the results agree:



The functional programming approach looks awkward when drawn using string diagrams, but simple when described on its own terms, especially if we write something short for $\ulcorner f \urcorner$, say `g`. Then we are simply applying `g` to `x` and obtaining `(g x)`. In the language of category theory, this amounts to using a natural transformation called **application**:

$$\begin{aligned} \mathrm{app}_{X,Y} \colon \hom(1, X \multimap Y) \times \hom(1, X) &\to \hom(1, Y) \\ (\ulcorner f \urcorner, x) &\mapsto fx \end{aligned}$$

which can be defined in any closed monoidal category.

Computer scientists call ways of getting new types from old 'type constructors'. From the category-theoretic viewpoint these correspond to functors (though in practice, often just their value on *objects* is given). For example, we have already seen that for any pair of types `X` and `Y` there is a function type `X` $\multimap$ `Y`. In category theory, this corresponds to the internal hom functor

$$\multimap \colon C^{\mathrm{op}} \times C \to C.$$

But there is also another functor inherent in the definition of 'closed monoidal category', the tensor product:

$$\otimes \colon C \times C \to C.$$

This gives another type constructor: for any types `X` and `Y` there is a **product type**, denoted `X` $\otimes$ `Y`.

Product types give another way to deal with programs that take more than one input. We have already mentioned a program

```
plus:integer ⊸ (integer ⊸ integer)
```

With the help of product types, we could also have a program

```
+: (integer ⊗ integer) ⊸ integer
```

Even more importantly, product types let us deal with programs that produce more than one output. So, for example, we might have a program that takes an integer and duplicates it:

```
duplicate: integer ⊸ (integer ⊗ integer)
```

Indeed, we would necessarily have such a program in any *cartesian* closed category — but not necessarily in quantum computation [27, 96].

In computer science, 'polymorphism' refers to the ability of certain programs to accept data, not just of a single fixed type, but of a variable type. The definition of closed monoidal category leads us naturally to polymorphism. For example, given data `x:X` and `y:Y`, we would like a program that combines them into an ordered pair: a datum `x` $\otimes$ `y: X` $\otimes$ `Y`. But, it would be tiresome to use a separate program to do this for each choice of types `X` and `Y`. So, we will use just one program: a 'polymorphic combinator'. We call it $\otimes$, and write:

```
⊗:(X ⊸ (Y ⊸ (X ⊗ Y))
```

Here `X` and `Y` are not fixed types, but variables standing for *arbitrary* types. To define how $\otimes$ behaves, we use the following rewrite rule:

```
((⊗ x) y) = x ⊗ y
```

All this seems reasonable, but how was it forced upon us by the the definition of 'closed monoidal category'? To answer this, we need to think about dinatural transformations. In Section 1.3.3 we saw that in logic, dinatural transformations give us inference schemas. In computation, they give us polymorphic combinators!

For example, $\otimes$ comes from a dinatural transformation which can be defined in any closed monoidal category. To see how this works, it will be slightly prettier if we henceforth switch to using *right* closed monoidal categories, where currying goes like this:

$$c_{X,Y,Z} \colon \hom(X \otimes Y, Z) \xrightarrow{\sim} \hom(X, Y \multimap Z).$$

With this convention, if we curry the identity

$$1_{X \otimes Y} \colon X \otimes Y \to X \otimes Y$$

we get a morphism called **coevaluation**:

$$\mathrm{coev}_{X,Y} \colon X \to (Y \multimap (X \otimes Y)).$$

Currying once more, we get a morphism

$$\otimes_{X,Y} \colon I \to (X \multimap (Y \multimap (X \otimes Y))).$$

We urge the reader to check that this extends to a dinatural transformation! Translating into the language of computer science, this gives the polymorphic combinator

```
⊗:(X ⊸ (Y ⊸ (X ⊗ Y))
```

We can continue along these lines, transcribing all the dinatural transformations inherent in the definition of 'closed monoidal category' into polymorphic combinators. We can also work out rewrite rules for these. For example, the dinatural transformation for composition gives us this:

```
compose:(Y ⊸ Z) ⊸ ((X ⊸ Y) ⊸ (X ⊸ Z))
(((compose g) f) x) = (g (f x))
```

while the one for identity morphisms gives us this:

```
id:X ⊸ X
(id x) = x
```

We should also have a polymorphic combinator called `curry`:

```
curry:((X ⊗ Y) ⊸ Z) ⊸ (X ⊸ (Y ⊸ Z))
(((curry f) x) y) = (f (x ⊗ y))
```

Since currying is a bijection, this should have an inverse, `uncurry`:

```
uncurry:(X ⊸ (Y ⊸ Z)) ⊸ ((X ⊗ Y) ⊸ Z)
((uncurry f) (x ⊗ y)) = ((f y) x)
```

This lets us start with a program

```
plus:integer ⊸ (integer ⊸ integer)
```

and define a new program

```
+: (integer ⊗ integer) ⊸ integer
```

using the rewrite rule

```
+ = (uncurry plus)
```

Then we have

```
(+ (1 ⊗ 3))
= ((uncurry plus) (1 ⊗ 3))
= ((plus 1) 3)
= 4
```

The associator also gives a polymorphic combinator:

```
assoc:((X ⊗ Y) ⊗ Z) ⊸ (X ⊗ (Y ⊗ Z))
(assoc (x ⊗ y) ⊗ z)) = x ⊗ (y ⊗ z)
```

And, since the associator is invertible, we should also have

```
unassoc:(X ⊗ (Y ⊗ Z)) ⊸ ((X ⊗ Y) ⊗ Z)
(assoc x ⊗ (y ⊗ z)) = (x ⊗ y) ⊗ z
```

Furthermore, corresponding to the unit object in a monoidal category, we should have a type `I` called the **unit type**. This type comes with a datum `unit:I`, and we have combinators corresponding to the right and left unit laws, along with the obvious rewrite rules:

```
left:(I ⊗ X) ⊸ X
(left (unit ⊗ x)) = x

right:(X ⊗ I) ⊸ X
(right (X ⊗ unit)) = x
```

These too need inverses.

So far we have not mentioned the braiding. But at this point, we hope the reader sees the pattern and is ready for a more formal treatment.

1.4.3 Combinator Calculi from Categories

In Section 1.3.3 we translated the concept of 'closed symmetric monoidal category' into the language of logic. We obtained a system of logic that works in any such category. Like pidgin English, this system is not very expressive. Its only virtue is that being so rudimentary, it works in a vast variety of situations: not just 'classical' ones, but also intuitionistic, linear and quantum ones — and even purely topological ones, like the category nCob.

A similar translation into the language of computer science will give us a rudimentary programming language. This will let us see any morphism in any closed symmetric monoidal category as a kind of 'program'.

We have already sketched how this should work. Suppose C is a closed symmetric monoidal category. Then any *object* $X \in C$ gives a *type* `X`. Any *morphism* $x\colon I \to X$ gives a *datum* of type `X`. All the *functors* inherent in the definition of 'symmetric closed monoidal category' give *type constructors*, and all the *dinatural transformations* give *polymorphic combinators*. Finally, all the *equations* between morphisms in C give *rewrite rules*.

We begin by doing this translation for just a closed monoidal category.

Definition 22. *A* **closed monoidal combinator calculus** *consists of:*

- *a collection of* **types** *closed under the following* **type constructors***:*
  - `I` *is a type;*
  - *if* `X` *and* `Y` *are types, then* `X ⊸ Y` *is a type;*
  - *if* `X` *and* `Y` *are types, then* `X ⊗ Y` *is a type.*
- *a collection of* **terms** *such that:*
  - *if* `X` *and* `Y` *are types and* `f:X ⊸ Y` *and* `x:X` *are terms, then* `(f x):Y` *is a term;*
  - *if* `X` *and* `Y` *are types and* `x:X` *and* `y:Y` *are terms, then* `x ⊗ y:X ⊗ Y` *is a term;*
  - `unit:I` *is a term.*
  - *if* `X`*,* `Y`*, and* `Z` *are types, then the following* **polymorphic combinators** *give terms:*
    - `id:X ⊸ X`
    - `compose:(Y ⊸ Z) ⊸ ((X ⊸ Y) ⊸ (X ⊸ Z))`
    - `curry:((X ⊗ Y) ⊸ Z) ⊸ (X ⊸ (Y ⊸ Z))`
    - `uncurry:((X ⊸ Y) ⊸ Z) ⊸ ((X ⊗ Y) ⊸ Z)`
    - `left:(I ⊗ X) ⊸ X`
    - `unleft:X ⊸ (I ⊗ X)`
    - `right:(X ⊗ I) ⊸ X`
    - `unright:X ⊸ (X ⊗ I)`
- *a collection of type-preserving* **rewrite rules***, including:*

- `(id t) = t`
- `(((compose t) u) v) = (t (u v))`
- `(((curry t) u) v) = (t (u ⊗ v))`
- `((uncurry t) (u ⊗ v)) = ((t u) v)`
- `(left (unit ⊗ t)) = t`
- `(unleft t) = unit ⊗ t`
- `(right (t ⊗ unit)) = t`
- `(unright t) = t ⊗ unit`

where the variables `t, u, v` *are understood to refer to the subterms in those positions.*

We can freely pass back and forth between closed monoidal categories and closed monoidal combinator calculi. However, the details deserve a little explanation, since morphisms come from *equivalence classes* of terms.

First, given a closed monoidal category C, there is a way to build a closed monoidal combinator calculus. The types in this calculus are the objects of C and all expressions generated from these using the type constructors $\multimap$ and $\otimes$. When X is an object of C, the terms `x:X` are the morphisms $x\colon I \to X$. Terms of other types are generated following the rules listed above. In addition to the rewrite rules listed above, we need rules that say how application and tensoring work in C. Given morphisms $x\colon I \to X$ and $g\colon I \to X \multimap Y$ in C, we can apply g to x and get a morphism $y\colon I \to Y$. These give terms `x:X`, `g:X ⊸ Y`, and `y:Y`, and we include a rewrite rule:

```
(g x) = y
```

Similarly, given morphisms $x\colon I \to X$ and $y\colon I \to Y$, tensoring gives an object $Z = X \otimes Y$ and a morphism $z\colon I \to Z$. These give terms `x:X`, `y:Y`, and `z:Z`, and we include a rewrite rule:

```
x ⊗ y = z
```

Second, from a closed monoidal combinator calculus, there is a way to build a closed monoidal category C. An object $X \in C$ is just a type `X`. A morphism of the form $x\colon X \to Y$ is an equivalence class of terms of type `X ⊸ Y`. The equivalence relation is generated as follows:

- if the terms `x:X` and `x′:X` are related by a rewrite rule, they are equivalent;
- if the terms `x:X` and `x′:X` are equivalent and the terms `f:X ⊸ Y` and `f′:X ⊸ Y` are equivalent, then `(f x):Y` and `(f′ x′):Y` are equivalent;
- if the terms `x:X` and `x′:X` are equivalent and the terms `y:Y` and `y′:Y` are equivalent, then `x ⊗ y: X ⊗ Y` and `x′ ⊗ y′: X ⊗ Y` are equivalent;

- if terms are 'extensionally equivalent', they are equivalent. Here we say `f:X ⊸ Y` and `f′:X ⊸ Y` are **extensionally equivalent** if `(f x):Y` and `(f′ x):Y` are equivalent for all terms `x:X`.

We define composition of morphisms using `compose`, define identity morphisms using `id`, and make C into a closed monoidal category using the other features of the calculus: the term constructors $\otimes$ and $\multimap$, and the various polymorphic combinators and rewrite rules.

Next we include the braiding. For this, we follow an approach going back to Schönfinkel. Namely, we use the braiding to construct a natural transformation

$$((X \otimes Y) \multimap Z) \to ((Y \otimes X) \multimap Z)$$

which in turn gives a natural transformation

$$(X \multimap (Y \multimap Z)) \to (Y \multimap (X \multimap Z))$$

which we can curry to give a dinatural transformation:

$$\mathrm{braid}_{X,Y,Z} \colon I \to (X \multimap (Y \multimap Z)) \multimap (Y \multimap (X \multimap Z))$$

This gives a polymorphic combinator which Schönfinkel called C. We instead call it `braid`:

```
braid:(X ⊸ (Y ⊸ Z)) ⊸ (Y ⊸ (X ⊸ Z))
```

At this point, a subtlety arises. The obvious rewrite rule for this combinator is

```
(((braid f) x) y) = ((f y) x)
```

But, this is only suited to *symmetric* monoidal categories, since the terms `(braid (braid f))` and `f` are equivalent in the sense described above. To describe nonsymmetric braided monoidal categories using combinatory logic, it seems we would need to specify rewrite rules in a way that depends explicitly on the desired braiding. To keep things simple, we limit ourselves to the symmetric case:

Definition 23. *A closed monoidal combinator calculus is* **symmetric** *if whenever* `X`*,* `Y`*, and* `Z` *are types, then*

```
braid:(X ⊸ (Y ⊸ Z)) ⊸ (Y ⊸ (X ⊸ Z))
```

is a term, and there is a rewrite rule:

```
(((braid f) x) y) = ((f y) x)
```

Such calculi give closed symmetric monoidal categories and vice versa. For more details, see the work of Abramsky, Haghverdi and Scott [5] on 'linear combinatory algebra', and Bierman's reformulation of intuitionistic linear logic in terms of a 'linear combinator calculus' [22]. Our setup is just a piece of what they have done.

1.5 Conclusions

In this paper we sketched how category theory can serve to clarify the analogies between physics, topology, logic and computation. Each field has its own concept of 'thing' (object) and 'process' (morphism) — and these things and processes are organized into categories that share many common features. To keep our task manageable, we focused on those features that are present in every closed symmetric monoidal category. Table 1.4, an expanded version of the Rosetta Stone, shows some of the analogies we found.

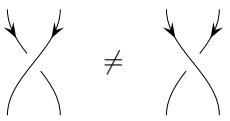
Category Theory	Physics	Topology	Logic	Computation
object X	Hilbert space X	manifold X	proposition X	data type `X`
morphism $f\colon X \to Y$	operator $f\colon X \to Y$	cobordism $f\colon X \to Y$	proof $f\colon X \to Y$	program `f: X -> Y`
tensor product of objects: $X \otimes Y$	Hilbert space of joint system: $X \otimes Y$	disjoint union of manifolds: $X \otimes Y$	conjunction of propositions: $X \otimes Y$	product of data types: `X ⊗ Y`
tensor product of morphisms: $f \otimes g$	parallel processes: $f \otimes g$	disjoint union of cobordisms: $f \otimes g$	proofs carried out in parallel: $f \otimes g$	programs executing in parallel: `f ⊗ g`
internal hom: $X \multimap Y$	Hilbert space of 'anti-X and Y': $X^* \otimes Y$	disjoint union of orientation-reversed X and Y: $X^* \otimes Y$	conditional proposition: $X \multimap Y$	function type: `X -> Y`

Table 1.4. The Rosetta Stone (larger version)

However, we only scratched the surface! There is much more to say about categories equipped with extra structure, and how we can use them to strengthen the ties between physics, topology, logic and computation — not to mention what happens when we go from categories to n-categories. But the real fun starts when we exploit these analogies to come up with new ideas and surprising connections. Here is an example.

In the late 1980s, Witten [95] realized that string theory was deeply connected to a 3d topological quantum field theory and thus the theory of knots and tangles [62]. This led to a huge explosion of work, which was ultimately distilled into a beautiful body of results focused on a certain class of compact braided monoidal categories called 'modular tensor categories' [14, 93].

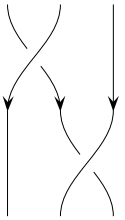
All this might seem of purely theoretical interest, were it not for the fact that superconducting thin films in magnetic fields seem to display an effect — the 'fractional quantum Hall effect' — that can be nicely modelled with the help of such categories [88, 89]. In a nutshell, the idea is that excitations of these films can act like particles, called 'anyons'. When two anyons trade places, the result depends on how they go about it:



So, collections of anyons are described by objects in a braided monoidal category! The details depend on things like the strength of the magnetic field; the range of possibilities can be worked out with the help of modular tensor categories [73, 77].

So far this is all about physics and topology. Computation entered the game around 2000, when Freedman, Kitaev, Larsen and Wang [40] showed that certain systems of anyons could function as 'universal quantum computers'. This means that, in principle, arbitrary computations can be carried out by moving anyons around. Doing this *in practice* will be far from easy. However, Microsoft has set up a research unit called Project Q attempting to do just this. After all, a working quantum computer could have huge practical consequences.

But regardless of whether topological quantum computation ever becomes practical, the implications are marvelous. A simple diagram like this:



can now be seen as a *quantum process*, a *tangle*, a *computation* — or an abstract morphism in any braided monoidal category! This is just the sort of thing one would hope for in a general science of systems and processes.

Acknowledgements

We owe a lot to participants of the seminar at UCR where some of this material was first presented: especially David Ellerman, Larry Harper, Tom Payne — and Derek Wise, who took notes [10]. MS would like to thank Google for letting him devote 20% of his time to this research, and Ken Shirriff for helpful corrections. Finally, this paper was vastly improved by comments at the n-Category Café, especially from Andrej Bauer, Tim Chevalier, Derek Elkins, Matt Hellige, Robin Houston, Todd Trimble, and Dave Tweed.

References

1. H. Abelson, G. J. Sussman and J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, 1996. Available at `http://mitpress.mit.edu/sicp/`.
2. S. Abramsky, Abstract scalars, loops, and free traced and strongly compact closed categories, in *Proceedings of CALCO 2005*, Lecture Notes in Computer Science **3629**, Springer, Berlin, 2005, 1–31. Also available at `http://web.comlab.ox.ac.uk/oucl/work/samson.abramsky/calco05.pdf`.
3. S. Abramsky and B. Coecke, A categorical semantics of quantum protocols, available at `arXiv:quant-ph/0402130`.

4. S. Abramsky and R. Duncan, A categorical quantum logic, to appear in *Mathematical Structures in Computer Science*, 2006. Also available as `arXiv:quant-ph/0512114`.
5. S. Abramsky, E. Haghverdi and P. Scott, Geometry of interaction and linear combinatory algebras, *Math. Struct. Comp. Sci.* **12** (2002), 625–665. Also available at `http://citeseer.ist.psu.edu/491623.html`.
6. M. F. Atiyah, Topological quantum field theories, *Publ. Math. IHES Paris* **68** (1989), 175–186. M. F. Atiyah, *The Geometry and Physics of Knots*, Cambridge U. Press, Cambridge, 1990.
7. J. Baez, An introduction to spin foam models of quantum gravity and BF theory, in *Geometry and Quantum Physics*, eds. H. Gausterer and H. Grosse, Springer, Berlin, 2000, pp. 25–93. Also available at `arXiv:gr-qc/9905087`.
8. J. Baez, Higher-dimensional algebra and Planck-scale physics, in *Physics Meets Philosophy at the Planck Length*, eds. C. Callender and N. Huggett, Cambridge U. Press, Cambridge, 2001, pp. 177–195. Also available as `arXiv:gr-qc/9902017`.
9. J. Baez, Quantum quandaries: a category-theoretic perspective, in *Structural Foundations of Quantum Gravity*, eds. S. French, D. Rickles and J. Saatsi, Oxford U. Press, Oxford, 2006, pp. 240-265. Also available as `arXiv:quant-ph/0404040`.
10. J. Baez, Classical versus quantum computation. Seminar notes by D. Wise available at `http://math.ucr.edu/home/baez/qg-fall2006/` and `http://math.ucr.edu/home/baez/qg-winter2007`.
11. J. Baez and J. Dolan, Higher-dimensional algebra and topological quantum field theory, *Jour. Math. Phys.* **36** (1995), 6073–6105. Also available as `arXiv:q-alg/9503002`.
12. J. Baez and L. Langford, Higher-dimensional algebra IV: 2-tangles, *Adv. Math.* **180** (2003), 705–764.
13. J. Baez and A. Lauda, A prehistory of n-categorical physics, to appear in proceedings of Deep Beauty: Mathematical Innovation and the Search for an Underlying Intelligibility of the Quantum World, Princeton, October 3, 2007, ed. Hans Halvorson. Also available at `http://math.ucr.edu/home/baez/history.pdf`.
14. B. Bakalov and A. Kirillov, Jr., *Lectures on Tensor Categories and Modular Functors*, American Mathematical Society, Providence, Rhode Island, 2001. Preliminary version available at http://www.math.sunysb.edu/∼kirillov/tensor/tensor.html.
15. H. Barendregt, *The Lambda Calculus, its Syntax and Semantics*, North–Holland, Amsterdam, 1984.
16. M. Barr and C. Wells, *Toposes, Triples and Theories*, Springer Verlag, Berlin, 1983. Revised and corrected version available at `http://www.cwru.edu/artsci/math/wells/pub/ttt.html`.
17. J. S. Bell, On the Einstein-Podolsky-Rosen paradox, *Physics* **1** (1964), 195–200.
18. J. L. Bell, *The Development of Categorical Logic*, available at `http://publish.uwo.ca/∼jbell/catlogprime.pdf`.
19. C. H. Bennett, P. G'acs, M. Li, P. Vit'anyi, and W. Zurek, Information distance, *IEEE Trans. Inform. Theory*, **44** (1998), 1407–1423. Also available at `http://citeseer.ist.psu.edu/166286.html`.
20. N. Benton, G. M. Bierman, V. de Paiva and J. M. E. Hyland, Linear lambda-calculus and categorical models revisited, in *Computer Science Logic (CSL'92), Selected Papers*, Lecture Notes in Computer Science **702**, Springer, Berlin, 1992, pp. 61–84. Also available at `http://citeseer.ist.psu.edu/benton92linear.html`
21. N. Benton, G. Bierman, V. de Paiva and M. Hyland, *Term Assignment for Intuitionistic Linear Logic*, Technical Report 262, University of Cambridge Computer Laboratory, August 1992. Also available at `http://citeseer.ist.psu.edu/1273.html`.
22. G. Bierman, *On Intuitionistic Linear Logic*, PhD Thesis, Cambridge University. Available at `http://research.microsoft.com/∼gmb/Papers/thesis.pdf`.

23. R. Blute and P. Scott, Category theory for linear logicians, in *Linear Logic in Computer Science*, eds. T. Ehrhard, J.-Y. Girard, P. Ruet, P. Scott, Cambridge U. Press, Cambridge, 2004, pp. 3–64. Also available as `http://www.site.uottawa.ca/~phil/papers/catsurv.web.pdf`.
24. S. N. Burris and H. P. Sankappanavar, *A Course in Universal Algebra*, Springer, Berlin, 1981. Also available as `http://www.math.uwaterloo.ca/~snburris/htdocs/ualg.html`.
25. V. Chari and A. Pressley, *A Guide to Quantum Groups*, Cambridge University Press, Cambridge, 1995.
26. E. Cheng and A. Lauda, *Higher-Dimensional Categories: an Illustrated Guidebook*. Available at `http://www.dpmms.cam.ac.uk/~elgc2/guidebook/`.
27. M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, Cambridge U. Press, Cambridge, 2000.
28. A. Church, An unsolvable problem of elementary number theory, *Amer. Jour. Math.* **58** (1936), 345–363.
29. B. Coecke, De-linearizing linearity: projective quantum axiomatics from strong compact closure, *Proceedings of the 3rd International Workshop on Quantum Programming Languages (QPL 2005)*, Elsevier, 2007, pp. 49–72. Also available as `arXiv:quant-ph/0506134`.
30. B. Coecke, Kindergarten quantum mechanics, to appear in *Proceedings of QTRF-III*. Also available as `arXiv:quant-ph/0510032`.
31. B. Coecke and E. O. Paquette, POVMs and Naimark's theorem without sums, to appear in *Proceedings of the 4th International Workshop on Quantum Programming Languages*. Also available as `arXiv:quant-ph/0608072`.
32. B. Coecke and D. Pavlovic, Quantum measurements without sums, to appear in *The Mathematics of Quantum Computation and Technology*, eds. Chen, Kauffman and Lomonaco, Taylor and Francis. Also available as `arXiv:quant-ph/0608035`.
33. R. L. Crole, *Categories for Types*, Cambridge U. Press, Cambridge, 1993.
34. H. B. Curry and R. Feys, *Combinatory Logic* Vol. I, North–Holland, Amsterdam, 1958.
35. P. Cvitanovic, *Group Theory*, Princeton U. Press, Princeton, 2003. Available at `http://www.nbi.dk/GroupTheory/`.
36. R. Di Cosmo and D. Miller, Linear logic, *Stanford Encyclopedia of Philosophy*, available at `http://plato.stanford.edu/entries/logic-linear/`.
37. M. Dorca and A. van Tonder, Quantum computation, categorical semantics and linear logic, available as `arXiv:quant-ph/0312174`.
38. S. Eilenberg and G. M. Kelly, Closed categories, in *Proceedings of the Conference on Categorical Algebra (La Jolla, 1965)*, Springer, Berlin, 1966, pp. 421–562.
39. S. Eilenberg and S. Mac Lane, General theory of natural equivalences, *Trans. Amer. Math. Soc.* **58** (1945), 231–294.
40. M. Freedman, A. Kitaev, M. Larsen and Z. Wang, Topological quantum computation, available as `arXiv:quant-ph/0101025`.
    M. Freedman, A. Kitaev and Z. Wang, Simulation of topological field theories by quantum computers, *Comm. Math. Phys.* **227** (2002), 587–603. Also available as `arXiv:quant-ph/0001071`.
    M. Freedman, A. Kitaev and Z. Wang, A modular functor which is universal for quantum computation, *Comm. Math. Phys.* **227** (2002), 605–622. Also available as `arXiv:quant-ph/0001108`.
41. P. Freyd and D. Yetter, Braided compact monoidal categories with applications to low dimensional topology, *Adv. Math.* **77** (1989), 156–182.
42. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*, Addison–Wesley, New York, 1994.
43. G. Gentzen, *Collected Papers of Gerhard Gentzen*, ed. M. E. Szabo, North–Holland, Amsterdam, 1969.

44. J.-Y. Girard, Linear logic, *Theor. Comp. Sci.* **50** (1987), 1–102. Also available at `http://iml.univ-mrs.fr/ girard/linear.pdf`.
45. J.-Y. Girard, Y. Lafont and P. Taylor, *Proofs and Types*, Cambridge U. Press, Cambridge, 1990. Also available at `http://www.monad.me.uk/stable/Proofs%2BTypes.html`.
46. K. Gödel, Zur intuitionistischen Arithmetik und Zahlentheorie, *Ergebnisse eines mathematischen Kolloquiums* **4** (1933), 34–38.
47. R. Goldblatt, *Topoi: the Categorial Analysis of Logic*, North–Holland, New York, 1984. Also available at `http://cdl.library.cornell.edu/cgi-bin/cul.math/docviewer?did=Gold010`.
48. D. Gottesman and I. L. Chuang, Quantum teleportation is a universal computational primitive, *Nature* **402** (1999), 390–393. Also available as `arXiv:quant-ph/9908010`.
49. M. Hasegawa, Logical predicates for intuitionistic linear type theories, *Typed Lambda Calculi and Applications: 4th International Conference, TLCA '99*, ed. J.-Y. Girard, Lecture Notes in Computer Science **1581**, Springer, Berlin, 1999. Also available at `http://citeseer.ist.psu.edu/187161.html`.
50. A. Heyting, ed., *L. E. J. Brouwer: Collected Works 1: Philosophy and Foundations of Mathematics*, Elsevier, Amsterdam, 1975.
51. W. A. Howard, The formulae-as-types notion of constructions, in *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, eds. J. P. Seldin and J. R. Hindley, Academic Press, New York, 1980, pp. 479–490.
52. A. Joyal and R. Street, The geometry of tensor calculus I, *Adv. Math.* **88** (1991), 55–113.
A. Joyal and R. Street, The geometry of tensor calculus II. Available at `http://www.math.mq.edu.au/ street/GTCII.pdf`.
53. A. Joyal and R. Street, Braided monoidal categories, *Macquarie Math Reports* 860081 (1986). Available at `http://rutherglen.ics.mq.edu.au/~street/JS86.pdf`.
A. Joyal and R. Street, Braided tensor categories, *Adv. Math.* **102** (1993), 20–78.
54. D. Kaiser, *Drawing Theories Apart: The Dispersion of Feynman Diagrams in Postwar Physics*, U. Chicago Press, Chicago, 2005.
55. C. Kassel, *Quantum Groups*, Springer, Berlin, 1995.
56. L. H. Kauffman, *Knots and Physics*, World Scientific, Singapore, 1991.
57. L. H. Kauffman and S. Lins, *Temperley–Lieb Recoupling Theory and Invariants of 3-Manifolds*, Princeton U. Press, Princeton, 1994.
58. G. M. Kelly and S. Mac Lane, Coherence in closed categories, *Jour. Pure Appl. Alg.* **1** (1971), 97–140 and 219.
59. G. M. Kelly and M. L. Laplaza, Coherence for compact closed categories, *Jour. Pure Appl. Alg.* **19** (1980), 193–213.
60. S. Kleene, λ-definability and recursiveness, *Duke Math. Jour.* **2** (1936), 340–353.
61. J. Kock, *Frobenius Algebras and 2D Topological Quantum Field Theories*, London Mathematical Society Student Texts **59**, Cambridge U. Press, Cambridge, 2004.
62. T. Kohno, ed., *New Developments in the Theory of Knots*, World Scientific, Singapore, 1990.
63. J. Lambek, From λ-calculus to cartesian closed categories, in *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, eds. J. P. Seldin and J. R. Hindley, Academic Press, New York, 1980, pp. 375–402.
64. J. Lambek and P. J. Scott, *Introduction to Higher-order Categorical Logic*, Cambridge U. Press, Cambridge, 1986.
65. P. Landin, A correspondence between ALGOL 60 and Church's lambda-notation, *Comm. ACM* **8** (1965), 89–101, 158–165.
66. F. W. Lawvere, *Functorial Semantics of Algebraic Theories*, Ph.D. Dissertation, Columbia University, 1963. Also available at `http://www.tac.mta.ca/tac/reprints/articles/5/tr5abs.html`.

67. T. Leinster, A survey of definitions of n-category, *Th. Appl. Cat.* **10** (2002), 1–70. Also available as `arXiv:math/0107188`.
68. S. Mac Lane, Natural associativity and commutativity, *Rice Univ. Stud.* **49** (1963) 28–46.
69. S. Mac Lane, *Categories for the Working Mathematician*, Springer, Berlin, 1998.
70. S. Mac Lane and I. Moerdijk, *Sheaves in Geometry and Logic: a First Introduction to Topos Theory*, Springer Verlag, Berlin, 1992.
71. M. Markl, S. Shnider and J. Stasheff, *Operads in Algebra, Topology and Physics*, American Mathematical Society, Providence, Rhode Island, 2002.
72. C. McLarty, *Elementary Categories, Elementary Toposes*, Clarendon Press, Oxford, 1995.
73. G. Moore and N. Read, Nonabelions in the the fractional quantum Hall effect, *Nucl. Phys. B* **360** (1991), 362–396.
74. J. Moschovakis, Intuitionistic logic *Stanford Encyclopedia of Philosophy*, available at `http://plato.stanford.edu/entries/logic-intuitionistic/`.
75. R. Penrose, Applications of negative dimensional tensors, in *Combinatorial Mathematics and its Applications*, ed. D. Welsh. Academic Press, 1971, pp. 221–244.
R. Penrose, Angular momentum: an approach to combinatorial space-time, in *Quantum Theory and Beyond*, ed. T. Bastin. Cambridge U. Press, 1971, pp. 151–180.
R. Penrose, On the nature of quantum geometry, in *Magic Without Magic*, ed. J. Klauder. Freeman, 1972, pp. 333–354.
R. Penrose, Combinatorial quantum theory and quantized directions, in *Advances in Twistor Theory*, eds. L. Hughston and R. Ward. Pitman Advanced Publishing Program, 1979, pp. 301–317.
76. G. Restall, *An Introduction to Substructural Logics*, Routledge, London, 2000.
G. Restall, Substructural logics, *Stanford Encyclopedia of Philosophy*, available at `http://plato.stanford.edu/entries/logic-substructural/`.
77. E. Rowell, R. Stong and Z. Wang, On classification of modular tensor categories, available as `arXiv:0712.1377`.
78. P. Scott, Some aspects of categories in computer science, in *Handbook of Algebra*, Vol. 2, ed. M. Hazewinkel, Elsevier, Amsterdam, 2000. Also available at `http://www.site.uottawa.ca/ phil/papers/handbook.ps`.
79. S. Sawin, Links, quantum groups and TQFTs, *Bull. Amer. Math. Soc.* **33** (1996), 413-445. Also available as `arXiv:q-alg/9506002`.
80. A. Schalk, What is a categorical model for linear logic? Available at `http://www.cs.man.ac.uk/~schalk/notes/llmodel.pdf`.
81. M. Schönfinkel, Über die Bausteine der mathematischen Logik, *Math. Ann.* **92** (1924), 305–316. Also available as On the building blocks of mathematical logic, trans. S. Bauer-Mengelberg, in *A Source Book in Mathematical Logic, 1879-1931*, ed. J. van Heijenoort, Harvard U. Press, Cambridge, Massachusetts, 1967, pp. 355–366.
82. R. A. G. Seely, Weak adjointness in proof theory, *Applications of Sheaves*, Lecture Notes in Mathematics **753**, Springer, Berlin, 697–701. Also available at `http://www.math.mcgill.ca/~rags/WkAdj/adj.pdf`.
83. G. Segal, The definition of a conformal field theory, in *Topology, Geometry and Quantum Field Theory: Proceedings of the 2002 Oxford Symposium in Honour of the 60th Birthday of Graeme Segal,* ed. U. L. Tillmann, Cambridge U. Press, 2004.
84. P. Selinger, Lecture notes on the lambda calculus, available at `http://www.mscs.dal.ca/~selinger/papers/#lambdanotes`.
85. P. Selinger, Dagger compact closed categories and completely positive maps, *Proceedings of the 3rd International Workshop on Quantum Programming Languages (QPL 2005)*, Elsevier, 2007, pp. 139–163. Also available at `http://www.mscs.dal.ca/~selinger/papers.html/#dagger`.

86. M.C. Shum, Tortile tensor categories, *Jour. Pure Appl. Alg.* **93** (1994), 57–110.
87. L. Smolin, The future of spin networks, *The Geometric Universe: Science, Geometry, and the Work of Roger Penrose*, eds. S. Hugget, P. Tod, and L. J. Mason, Oxford U. Press, Oxford, 1998. Also available as `arXiv:gr-qc/9702030`.
88. A. Stern, Anyons and the quantum Hall effect – a pedagogical review, *Ann. Phys.* **323** (2008), 204–249. available as `arXiv:0711.4697`.
89. M. Stone, ed., *Quantum Hall Effect*, World Scientific, Singapore, 1992.
90. M. Szabo, *Algebra of Proofs*, North–Holland, Amsterdam, 1978.
91. T. Trimble, *Linear Logic, Bimodules, and Full Coherence for Autonomous Categories*, Ph.D. thesis, Rutgers University, 1994.
92. A. S. Troelstra, *Lectures on Linear Logic*, Center for the Study of Language and Information, Stanford, California, 1992.
93. V. G. Turaev, *Quantum Invariants of Knots and 3-Manifolds*, de Gruyter, Berlin, 1994.
94. A. van Tonder, A lambda calculus for quantum computation, *SIAM Jour. Comput.* **33** (2004), 1109–1135. Also available as `arXiv:quant-ph/0307150`.
95. E. Witten, Quantum field theory and the Jones polynomial, *Comm. Math. Phys.* **121** (1989), 351–399.
96. W. K. Wootters and W. H. Zurek, A single quantum cannot be cloned, *Nature* **299** (1982), 802–803.
97. D. N. Yetter, *Functorial Knot Theory: Categories of Tangles, Coherence, Categorical Deformations, and Topological Invariants*, World Scientific, Singapore, 2001.