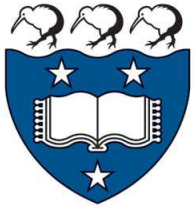
The Department of Computer Science
The University of Auckland
New Zealand

Physics and Computation

Michael Stay
January 2015

Supervisors:
Cristian Calude
John C. Baez



A thesis submitted in partial fulfillment of the requirements of Doctor of Philosophy in Computer Science

Abstract

This thesis is an exploration of some places where ideas in computer science and physics share a common mathematical structure. The first part of the thesis deals with partition functions in physics and algorithmic information theory. In physics, partition functions are used to encode information about statistical systems in thermal equilibrium; in algorithmic information theory, they are used to encode information about the probability that a Turing machine will halt given a random program. We derive analogues of Maxwell's relations in the algorithmic setting and consider thermodynamic cycles such as the Carnot cycle or Stoddard cycle. We also show that given a program and a probability P, we can effectively compute a time after which the probability that the program will eventually halt is less than P. This idea of a time cutoff is reminiscent of a high-energy cutoff in renormalization.

The second part of the thesis reviews symmetric monoidal closed categories and bicategories. We begin with an expository chapter on symmetric monoidal closed categories and demonstrate how they form a broad generalization of the Curry-Howard isomorphism, including string diagrams in physics, cobordisms in topology, multiplicative intuitionistic linear logic, and the simply-typed lambda calculus in computer science. We then go up one dimension and present the complete definition of a special kind of symmetric monoidal closed bicategory called a compact closed bicategory. We emphasize the combinatorial aspects and prove that given a 2-category T with finite products and weak pullbacks, the bicategory $\mathrm{Span}_2(T)$ of objects of T, spans, and isomorphism classes of maps of spans is compact closed. As a corollary, certain bicategories of "resistor networks" are compact closed.

Acknowledgements

Thanks to Google for funding and 20% time. Thanks to Cristian Calude for forgiveness of unintended consequences. Thanks to John Baez for teaching so that even I could understand. Thanks to both for years of friendship, good advice, patience, and faith in me. Thanks most of all to my wife Miriam, for loving me despite everything.

Contents

Introduction

There is a nice analogy between the set of possible arrangements of gas particles in a piston and the set of halting programs written in a given programming language. In Chapter I.1, we talk about "observables" in each case: just as we can ask for the volume of the convex hull of the particles, their total kinetic energy, or how many particles there are, we can ask for the length of a program, its runtime, or the amount of memory it uses. We associate thermodynamic conjugate variables to each observable: just as pressure is conjugate to volume, we can talk about the "algorithmic pressure" that is conjugate to the size of the program. We single out one observable property of a program to play the role of internal energy and then define the entropy of a distribution on programs and analogues of Maxwell's equations. Finally, by considering loops in the pressure/volume space, we describe an "algorithmic heat engine".

Algorithmic entropy is a special case of the entropy as studied in statistical mechanics. A Gibbs ensemble is a probability measure that maximizes the entropy subject to constraints on the average values of some observables. In most work on algorithmic entropy, the relevant observable is the length of a program; however, the full structure of thermodynamics only appears when we consider multiple observables. We focus on the log of the runtime E, the length V, and the output N. The Gibbs ensemble is of the form

$$p = \frac{1}{Z} e^{-\beta E(x) - \gamma V(x) - \delta N(x)}$$

for certain β, γ, δ, where

$$Z = \sum_{x \in X} e^{-\beta E(x) - \gamma V(x) - \delta N(x)}$$

is called the 'partition function' of the ensemble and X is the domain of some universal Turing machine U. The partition function reduces to the halting probability for U when $\beta = \delta = 0$ and $\gamma = \ln 2$.

We derive an algorithmic analogue of the basic thermodynamic relation

$$dE = TdS - PdV + dN,$$

where S is the entropy, $T = 1/\beta, P = \gamma/\beta$, and $\mu = -\delta/\beta$ are algorithmic versions of temperature, pressure, and chemical potential, respectively. Starting from this relation, we derive analogues of Maxwell's equations and consider thermodynamic cycles like the Carnot cycle or Stoddard cycle.

The halting probability of a universal Turing machine is uncomputable. However, in Chapter I.2, we show that we can effectively bound the probability that a particular program halts. Consider an N-bit program p that runs for a long time without stopping—say, 2^{2^N} steps. There is a program

not much longer than N bits that will run p and, if it ever halts, will output the number of steps t required for p to halt. This is a short program of length $N + c$ that produces a large number t whose length is greater than $2N$, so t must be nonrandom. The density of nonrandom numbers near t goes to zero as t goes to infinity, so the density of times at which the program might halt also goes to zero. By choosing a computably enumerable distribution on natural numbers, we turn densities into probabilities: given an N-bit program p and a natural number k, we show that we can effectively compute a critical time t_0 such that the probability that p halts after running at least t_0 steps is less than 2^{-k}. Manin [146] referenced the published version of this chapter and showed that this critical time is formally equivalent to a high-energy cutoff when renormalizing quantum field theory.

It is well-known that Wick rotation turns equations describing thermal systems into equations describing quantum systems. By replacing the energy scale kT in a classical partition function by the imaginary energy $i\hbar/t$, we get a quantum partition function. Consider a large collection of harmonic oscillators at temperature T. The relative probability of finding any given oscillator with energy E is $\exp(-E/k_BT)$, where k_B is Boltzmann's constant. The average value of an observable Q is, up to a normalizing constant,

$$\sum_j Q_j e^{-E_j/(k_BT)}.$$

Now consider a single quantum harmonic oscillator in a superposition of basis states, evolving for a time t under a Hamiltonian H. The relative phase change of the basis state with energy E is $\exp(-Eit/\hbar)$, where $\hbar$ is Planck's constant. The probability amplitude that a uniform superposition of states $|\psi\rangle = \sum_j |j\rangle$ evolves to an arbitrary superposition $|Q\rangle = \sum_j Q_j|j\rangle$ is, up to a normalizing constant,

$$\begin{aligned}
&\langle Q|e^{-iHt/\hbar}|\psi\rangle \\
&= \sum_j Q_j e^{-E_j it/\hbar}\langle j|j\rangle \\
&= \sum_j Q_j e^{-E_j it/\hbar}.
\end{aligned}$$

Feynman's path integral formulation of quantum mechanics considers a sum over paths γ rather than a sum over states, each weighted by a phase $e^{-iS(\gamma)/\hbar}$, where $S(\gamma)$ is the classical action of the path. When we move from quantum mechanics to quantum field theory, the partition function sums over diagrams rather than paths. Feynman diagrams form a category: there is a trivial diagram for any set of particles where they do not interact at all, and we can compose any two diagrams where the output particles of one diagram match the input particles of the next. This category is equipped with certain structure that also appears in programming languages, linear logic, and topology.

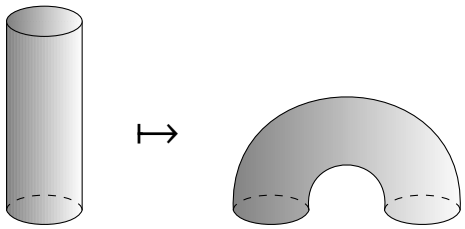
The shared structure is called a "symmetric monoidal closed category"; the contribution of Chapter II.1 of this thesis is an exposition of existing work on symmetric monoidal closed categories and the extension of the appropriate version of the Curry-Howard isomorphism to Feynman diagrams and Hilbert spaces.

A category consists of a collection of "objects" and, for each pair (x, y) of objects, a set of "morphisms" from x to y. Morphisms are composable if the target of one matches the source of the next;

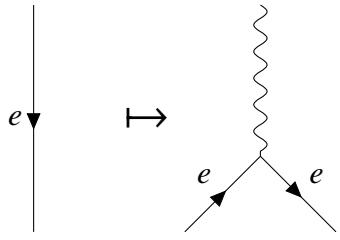
every object has an identity morphism from the object to itself; and composition is associative and unital. Feynman diagrams form a category in which objects are particle types and morphisms are graphs of interactions between them. We interpret Feynman diagrams in the category of Hilbert spaces and linear transformations; that is, we associate to each particle a Hilbert space of states and to each diagram a linear transformation that tells how those states change over time. In computer science, we have a category of data types with typed functions between them; in the Haskell community, this category is called "Hask". In linear logic, we have categories whose objects are propositions and whose morphisms are constructive proofs. In topology, we have categories whose objects are manifolds of a certain dimension and whose morphisms are cobordisms.

A "monoidal" category lets us pair up objects and morphisms. We can juxtapose Feynman diagrams to get a new diagram and take the tensor product of Hilbert spaces to get a new Hilbert space. Programming languages usually provide some way of combining data types into a new one; Python and Scala, for instance, have "tuple" type constructors. In linear logic, the assertion that both the propositions P and Q hold is itself a proposition. In topology, the disjoint union of two manifolds is a manifold. A "braided" monoidal category lets us move objects past each other using an isomorphism called the "braiding". A "symmetric" monoidal category is one in which braiding twice is the identity.

A symmetric monoidal "closed" category has a notion of a "function type" or "implication" or "time reversal". Given two Haskell data types `P` and `Q`, the data type `P -> Q` describes typed functions between them. In linear logic, given two propositions P and Q, the assertion that P implies Q is a proposition. In topology, given any cobordism from a manifold P to a manifold Q, we can "bend" the input around and make it an output, getting a cobordism from the empty manifold to Q and the reverse orientation of P.



Similarly, given a Feynman diagram from a set P of particles to a set Q, we can "bend" the input set around and make it an output, getting a diagram from photons to Q and the antiparticles of P:



Given any linear transformation from a Hilbert space P to a Hilbert space Q, we can use the notion of "gate teleportation" [87] to encode the transformation into a quantum state in $P^* \otimes Q$. When the

function type can be expressed in terms of a dual object and the tensor product, as with Feynman diagrams, the category of cobordisms, or Hilbert spaces, we call the category "compact closed".

Feynman diagrams were designed to represent quantum systems interacting. We assign a Hilbert space of quantum states to each particle such that the pairing and braiding operations are preserved: if we assign the space U to one particle and V to another, then we have to assign $U \otimes V$ to the pair of particles. We assign a linear transformation to each vertex that tells how the quantum states evolve. This kind of structure-preserving map is called a "braided monoidal functor"; every monoidal functor also preserves duals, so the entire structure of a compact closed category is preserved.

We can consider such functors from other compact closed categories into the category Hilb of Hilbert spaces and linear transformations. For example, we can think of manifolds as modeling empty curved space, and cobordisms as modeling spacetime. A braided monoidal functor from the category of manifolds and cobordisms to Hilb assigns a Hilbert space of quantum states to space and a linear transformation to spacetime, giving a toy model of quantum gravity. This toy model does not include matter; it only talks about topological changes in space over time, so the model is called a "topological quantum field theory".

The structure of a compact closed category can be generalized to "bicategories", where in addition to morphisms between objects we have 2-morphisms between morphisms. In Chapter II.2, we lay out the complete definition of a compact closed bicategory (the parts of which have not appeared together in a single place before) and then prove that various useful bicategories are compact closed. We take special note of bicategories of "spans".

In particular, given a category T with pullbacks, we can define a bicategory Span(T) whose

- objects are those of T,
- morphisms from A to B are "spans" consisting of an "apex" object C and an ordered pair of morphisms $(f : C \to A, g : C \to B)$ from T, and
- 2-morphisms are morphisms from C to C' such that the obvious diagram commutes.

If T is the category of finite sets, then we can think of the subset of C consisting of those elements c such that $f(c) = a$ and $g(c) = b$ as being a "matrix element" at row a and column b. If we simply count these subsets, we get a matrix of natural numbers, and the pullback corresponds to matrix multiplication. By using categories T other than the category of finite sets, we get a vast generalization of linear algebra. We conclude Chapter II.2 with a proof that when we take the monoidal tricategory of spans described by Hoffnung [99] and mod out by 3-isomorphisms, we get a compact closed bicategory.

Compact closed bicategories are interesting to us because they open up at least three areas for future research. First, Lawvere showed that we can think of categories with products as a kind of programming language. As programmers, we can write a Java interface that describes the operations on a monoid (identity element and multiplication) and write tests to check the relations (associativity and unit laws). This interface, together with the tests, is a presentation of a category with products, called "the Lawvere theory for monoids". Every implementation of the interface describes a functor from the Lawvere theory for monoids into the category Set and vice versa. The

tao of categorification suggests there should be a "higher Lawvere theory of symmetric monoidal closed categories"; the fact that the currying adjunction between tensor and the internal hom is an isomorphism of profunctors means that this higher theory ought to be a compact closed bicategory and take models in Prof.

Second, Lambek and Scott showed that simply-typed lambda calculus forms a cartesian closed category, but only if we *ignore the process of computation.* This is only possible because lambda calculus is confluent: it does not matter in what order the rewrite rules are applied. Concurrent calculi like Milner's pi calculus are not confluent; once we can express contention for resources—like a deposit to and a withdrawal from the same bank account—it matters a great deal in which order the rewrites occur. This suggests that we need to explicitly account for rewrites using 2-morphisms in a bicategorical setting. The higher theory of a symmetric monoidal closed category above provides many of the pieces we need for the pi calculus: the unit object can be the zero process, the tensor product can be concurrency, and the internal hom should be involved in putting a process under a prefix. The 2-morphisms for adjunctions in this bicategory drawn as string diagrams look amazingly like synchronization on a name.

Finally, this bicategorical approach should also fit better with physics: rewrites are processes that occur over time like particle interactions do. Extended topological quantum field theories should be functors between compact closed bicategories, so it is not unreasonable to expect a nice quantum interpretation of some kind of linear pi calculus.

Part I

Algorithmic information theory

Chapter 1

Algorithmic thermodynamics

The first mathematical structure we will examine that appears in both computer science and physics is the partition function. Partition functions encode how likely it is to find a system in a given state. From the partition function, we can compute the "entropy", the number of bits required to pick out the particular state the system is in. In statistical mechanics, the bits describe the positions and momenta of a collection of particles; in algorithmic information theory, the bits describe a program.

Algorithmic entropy can be seen as a special case of entropy as studied in statistical mechanics. This viewpoint allows us to apply many techniques developed for use in thermodynamics to the subject of algorithmic information theory. In particular, suppose we fix a universal prefix-free Turing machine and let X be the set of programs that halt for this machine. Then we can regard X as a set of 'microstates', and treat any function on X as an 'observable'. For any collection of observables, we can study the Gibbs ensemble that maximizes entropy subject to constraints on expected values of these observables. We illustrate this by taking the log runtime, length, and output of a program as observables analogous to the energy E, volume V and number of molecules N in a container of gas. The conjugate variables of these observables allow us to define quantities which we call the 'algorithmic temperature' T, 'algorithmic pressure' P and 'algorithmic potential' μ, since they are analogous to the temperature, pressure and chemical potential. We derive an analogue of the fundamental thermodynamic relation $dE = TdS - PdV + \mu dN$, and use it to study thermodynamic cycles analogous to those for heat engines. We also investigate the values of T, P and μ for which the partition function converges. At some points on the boundary of this domain of convergence, the partition function becomes uncomputable. Indeed, at these points the partition function itself has nontrivial algorithmic entropy.

1.1 Introduction

Many authors [34, 56, 78, 129, 141, 183, 194, 196] have discussed the analogy between algorithmic entropy and entropy as defined in statistical mechanics: that is, the entropy of a probability measure p on a set X. It is perhaps insufficiently appreciated that algorithmic entropy can be seen as a

special case of the entropy as defined in statistical mechanics. We describe how to do this in Section 1.3.

This allows all the basic techniques of thermodynamics to be imported to algorithmic information theory. The key idea is to take X to be some version of 'the set of all programs that eventually halt and output a natural number', and let p be a Gibbs ensemble on X. A Gibbs ensemble is a probability measure that maximizes entropy subject to constraints on the mean values of some observables — that is, real-valued functions on X.

In most traditional work on algorithmic entropy, the relevant observable is the length of the program. However, much of the interesting structure of thermodynamics only becomes visible when we consider several observables. When X is the set of programs that halt and output a natural number, some other important observables include the output of the program and logarithm of its runtime. So, in Section 1.4 we illustrate how ideas from thermodynamics can be applied to algorithmic information theory using these three observables.

To do this, we consider a Gibbs ensemble of programs which maximizes entropy subject to constraints on:

- E, the expected value of the logarithm of the program's runtime (which we treat as analogous to the energy of a container of gas),
- V, the expected value of the length of the program (analogous to the volume of the container), and
- N, the expected value of the program's output (analogous to the number of molecules in the gas).

This measure is of the form

$$p = \frac{1}{Z} e^{-\beta E(x) - \gamma V(x) - \delta N(x)}$$

for certain numbers β, γ, δ, where the normalizing factor

$$Z = \sum_{x \in X} e^{-\beta E(x) - \gamma V(x) - \delta N(x)}$$

is called the 'partition function' of the ensemble. The partition function reduces to Chaitin's number Ω when $\beta = 0$, $\gamma = \ln 2$ and $\delta = 0$. This number is uncomputable [56]. However, we show that the partition function Z is computable when $\beta > 0$, $\gamma \geq \ln 2$, and $\delta \geq 0$.

We derive an algorithmic analogue of the basic thermodynamic relation

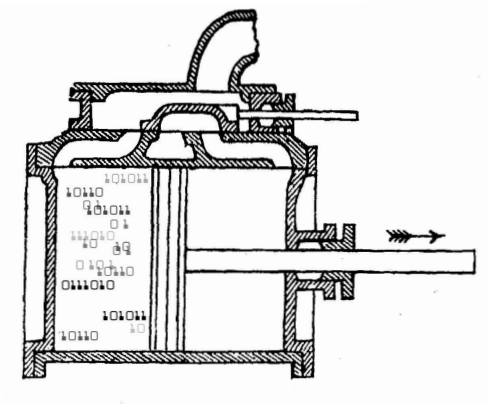
$$dE = TdS - PdV + \mu dN.$$

Here:

- S is the entropy of the Gibbs ensemble,

- $T = 1/\beta$ is the 'algorithmic temperature' (analogous to the temperature of a container of gas). Roughly speaking, this counts how many times you must double the runtime in order to double the number of programs in the ensemble while holding their mean length and output fixed.
- $P = \gamma/\beta$ is the 'algorithmic pressure' (analogous to pressure). This measures the tradeoff between runtime and length. Roughly speaking, it counts how much you need to decrease the mean length to increase the mean log runtime by a specified amount, while holding the number of programs in the ensemble and their mean output fixed.
- $\mu = -\delta/\beta$ is the 'algorithmic potential' (analogous to chemical potential). Roughly speaking, this counts how much the mean log runtime increases when you increase the mean output while holding the number of programs in the ensemble and their mean length fixed.

Starting from this relation, we derive analogues of Maxwell's relations and consider thermodynamic cycles such as the Carnot cycle or Stoddard cycle. For this we must introduce concepts of 'algorithmic heat' and 'algorithmic work'.



Charles Babbage described a computer powered by a steam engine; we describe a heat engine powered by programs! We admit that the significance of this line of thinking remains a bit mysterious. However, we hope it points the way toward a further synthesis of algorithmic information theory and thermodynamics. We call this hoped-for synthesis 'algorithmic thermodynamics'.

1.2 Related Work

Li and Vitányi use the term 'algorithmic thermodynamics' for describing physical states using a universal prefix-free Turing machine U. They look at the smallest program p that outputs a description x of a particular microstate to some accuracy, and define the physical entropy to be

$$S_A(x) = (k \ln 2)(K(x) + H_x),$$

where $K(x) = |p|$ and H_x embodies the uncertainty in the actual state given x. They summarize their own work and subsequent work by others in chapter eight of their book [142]. Whereas they

consider $x = U(p)$ to be a microstate, we consider p to be the microstate and x the value of the observable U. Then their observables $O(x)$ become observables of the form $O(U(p))$ in our model.

Tadaki [195] generalized Chaitin's number Ω to a function Ω^D and showed that the value of this function is compressible by a factor of exactly D when D is computable. Calude and Stay [52] pointed out that this generalization was formally equivalent to the partition function of a statistical mechanical system where temperature played the role of the compressibility factor, and studied various observables of such a system. Tadaki [196] then explicitly constructed a system with that partition function: given a total length E and number of programs N, the entropy of the system is the log of the number of E-bit strings in $\mathrm{dom}(U)^N$. The temperature is

$$\frac{1}{T} = \left.\frac{\Delta E}{\Delta S}\right|_N .$$

In a follow-up paper [197], Tadaki showed that various other quantities like the free energy shared the same compressibility properties as Ω^D. In this thesis, we consider multiple variables, which is necessary for thermodynamic cycles, chemical reactions, and so forth.

Manin and Marcolli [147] derived similar results in a broader context and studied phase transitions in those systems. Manin [145, 146] also outlined an ambitious program to treat the infinite runtimes one finds in undecidable problems as singularities to be removed through the process of renormalization. In a manner reminiscent of hunting for the proper definition of the "one-element field" F_{un}, he collected ideas from many different places and considered how they all touch on this central theme. While he mentioned a runtime cutoff as being analogous to an energy cutoff, the renormalizations he presented are uncomputable. In this thesis, we take the log of the runtime as being analogous to the energy; the randomness described by Chaitin and Tadaki then arises as the infinite-temperature limit.

1.3 Algorithmic Entropy

To see algorithmic entropy as a special case of the entropy of a probability measure, it is useful to follow Solomonoff [183] and take a Bayesian viewpoint. In Bayesian probability theory, we always start with a probability measure called a 'prior', which describes our assumptions about the situation at hand before we make any further observations. As we learn more, we may update this prior. This approach suggests that we should define the entropy of a probability measure *relative to another probability measure* — the prior.

A probability measure p on a finite set X is simply a function $p\colon X \to [0, 1]$ whose values sum to 1, and its entropy is defined as follows:

$$S(p) = -\sum_{x \in X} p(x) \ln p(x).$$

But we can also define the entropy of p relative to another probability measure q:

$$S(p, q) = -\sum_{x \in X} p(x) \ln \frac{p(x)}{q(x)}.$$

This **relative entropy** has been extensively studied and goes by various other names, including 'Kullback–Leibler divergence' [130] and 'information gain' [165].

The term 'information gain' is nicely descriptive. Suppose we initially assume the outcome of an experiment is distributed according to the probability measure q. Suppose we then repeatedly do the experiment and discover its outcome is distributed according to the measure p. Then the information gained is $S(p, q)$.

Why? We can see this in terms of coding. Suppose X is a finite set of signals which are randomly emitted by some source. Suppose we wish to encode these signals as efficiently as possible in the form of bit strings. Suppose the source emits the signal x with probability $p(x)$, but we erroneously believe it is emitted with probability $q(x)$. Then $S(p, q)/\ln 2$ is the expected extra message-length per signal that is required if we use a code that is optimal for the measure q instead of a code that is optimal for the true measure, p.

The ordinary entropy $S(p)$ is, up to a constant, just the relative entropy in the special case where the prior assigns an equal probability to each outcome. In other words:

$$S(p) = S(p, q_0) + S(q_0)$$

when q_0 is the so-called 'uninformative prior', with $q_0(x) = 1/|X|$ for all $x \in X$.

We can also define relative entropy when the set X is countably infinite. As before, a probability measure on X is a function $p\colon X \to [0, 1]$ whose values sum to 1. And as before, if p and q are two probability measures on X, the entropy of p relative to q is defined by

$$S(p, q) = -\sum_{x \in X} p(x) \ln \frac{p(x)}{q(x)}. \tag{1.1}$$

But now the role of the prior becomes more clear, because there is no probability measure that assigns the same value to each outcome!

In what follows we will take X to be — roughly speaking — the set of all programs that eventually halt and output a natural number. As we shall see, while this set is countably infinite, there are still some natural probability measures on it, which we may take as priors.

To make this precise, we recall the concept of a universal prefix-free Turing machine. In what follows we use **string** to mean a bit string, that is, a finite, possibly empty, list of 0's and 1's. If x and y are strings, let $x\|y$ be the concatenation of x and y. A **prefix** of a string z is a substring beginning with the first letter, that is, a string x such that $z = x\|y$ for some y. A **prefix-free** set of strings is one in which no element is a prefix of any other. The **domain** $\mathrm{dom}(M)$ of a Turing machine M is the set of strings that cause M to eventually halt. We call the strings in $\mathrm{dom}(M)$ **programs**. We assume that when the M halts on the program x, it outputs a natural number $M(x)$. Thus we may think of the machine M as giving a function $M\colon \mathrm{dom}(M) \to \mathbb{N}$.

A **prefix-free Turing machine** is one whose halting programs form a prefix-free set. A prefix-free machine U is **universal** if for any prefix-free Turing machine M there exists a constant c such that for each string x, there exists a string y with

$$U(y) = M(x) \text{ and } |y| < |x| + c.$$

Let U be a universal prefix-free Turing machine. Then we can define some probability measures on $X = \mathrm{dom}(U)$ as follows. Let

$$|\cdot| \colon X \to \mathbb{N}$$

be the function assigning to each bit string its length. Then there is for any constant $\gamma \geq \ln 2$ a probability measure p given by

$$p(x) = \frac{1}{Z} e^{-\gamma |x|}.$$

Here the normalization constant Z is chosen to make the numbers $p(x)$ sum to 1:

$$Z = \sum_{x \in X} e^{-\gamma |x|}.$$

It is worth noting that for computable real numbers $\gamma \geq \ln 2$, the normalization constant Z is uncomputable [195]. Indeed, when $\gamma = \ln 2$, Z is Chaitin's famous number Ω. We return to this issue in Section 1.4.5.

Let us assume that each program prints out some natural number as its output. Thus we have a function

$$N \colon X \to \mathbb{N}$$

where $N(x)$ equals i when program x prints out the number i. We may use this function to 'push forward' p to a probability measure q on the set $\mathbb{N}$. Explicitly:

$$q(i) = \sum_{x \in X : N(x) = i} e^{-\gamma |x|} \ .$$

In other words, if i is some natural number, $q(i)$ is the probability that a program randomly chosen according to the measure p will print out this number.

Given any natural number n, there is a probability measure δ_n on $\mathbb{N}$ that assigns probability 1 to this number:

$$\delta_n(m) = \begin{cases} 1 & \text{if } m = n \\ 0 & \text{otherwise.} \end{cases}$$

We can compute the entropy of δ_n relative to q:

$$\begin{aligned} S(\delta_n, q) &= -\sum_{i \in \mathbb{N}} \delta_n(i) \ \ln \frac{\delta_n(i)}{q(i)} \\ &= -\ln \left(\sum_{x \in X \colon \ N(x) = n} e^{-\gamma |x|} \right) + \ln Z. \end{aligned} \tag{1.2}$$

Since the quantity $\ln Z$ is independent of the number n, and uncomputable, it makes sense to focus attention on the other part of the relative entropy:

$$-\ln \left(\sum_{x \in X \colon \ N(x) = n} e^{-\gamma |x|} \right).$$

If we take $\gamma = \ln 2$, this is precisely the **algorithmic entropy** [57, 141] of the number n. So, up to the additive constant $\ln Z$, we have seen that *algorithmic entropy is a special case of relative entropy*.

One way to think about entropy is as a measure of surprise: if you can predict what comes next — that is, if you have a program that can compute it for you — then you are not surprised. For example, the first 2000 bits of the binary fraction for 1/3 can be produced with this short Python program:

```
print "01" * 1000
```

But if the number is complicated, if every bit is surprising and unpredictable, then the shortest program to print the number does not do any computation at all! It just looks something like

```
print "1010000110010100101001010001011111011011010010l0"
```

Levin's coding theorem [140] says that the difference between the algorithmic entropy of a number and its **Kolmogorov complexity** — the length of the shortest program that outputs it — is bounded by a constant that only depends on the programming language.

So, up to some error bounded by a constant, *algorithmic information is information gain*. The algorithmic entropy is the information gained upon learning a number, if our prior assumption was that this number is the output of a randomly chosen program — randomly chosen according to the measure p where $\gamma = \ln 2$.

So, algorithmic entropy is not just *analogous* to entropy as defined in statistical mechanics: it is a *special case*, as long as we take seriously the Bayesian philosophy that entropy should be understood as relative entropy. This realization opens up the possibility of taking many familiar concepts from thermodynamics, expressed in the language of statistical mechanics, and finding their counterparts in the realm of algorithmic information theory.

But to proceed, we must also understand more precisely the role of the measure p. In the next section, we shall see that this type of measure is already familiar in statistical mechanics: it is a Gibbs ensemble.

1.4 Algorithmic Thermodynamics

Suppose we have a countable set X, finite or infinite, and suppose $C_1, \ldots, C_n \colon X \to \mathbb{R}$ is some collection of functions. Then we may seek a probability measure p that maximizes entropy subject to the constraints that the mean value of each observable C_i is a given real number $\overline{C}_i$:

$$\sum_{x \in X} p(x)\, C_i(x) = \overline{C}_i.$$

As nicely discussed by Jaynes [106, 107], the solution, if it exists, is the so-called **Gibbs ensemble**:

$$p(x) = \frac{1}{Z} e^{-(s_1 C_1(x) + \cdots + s_n C_n(x))}$$

for some numbers $s_i \in \mathbb{R}$ depending on the desired mean values $\overline{C}_i$. Here the normalizing factor Z is called the **partition function**:

$$Z = \sum_{x \in X} e^{-(s_1 C_1(x) + \cdots + s_n C_n(x))} .$$

In thermodynamics, X represents the set of **microstates** of some physical system. A probability measure on X is also known as an **ensemble**. Each function $C_i \colon X \to \mathbb{R}$ is called an **observable**, and the corresponding quantity s_i is called the **conjugate variable** of that observable. For example, the conjugate of the energy E is the inverse of temperature T, in units where Boltzmann's constant equals 1. The conjugate of the volume V — of a piston full of gas, for example — is the pressure P divided by the temperature. And in a gas containing molecules of various types, the conjugate of the number N_i of molecules of the ith type is minus the 'chemical potential' μ_i, again divided by temperature. For easy reference, we list these observables and their conjugate variables below.

THERMODYNAMICS

Observable	Conjugate Variable
energy: E	$\frac{1}{T}$
volume: V	$\frac{P}{T}$
number: N_i	$-\frac{\mu_i}{T}$

Now let us return to the case where $X = \mathrm{dom}(U)$. Recalling that programs are bit strings, one important observable for programs is the length:

$$|\cdot| \colon X \to \mathbb{N}.$$

We have already seen the measure

$$p(x) = \frac{1}{Z} e^{-\gamma |x|}.$$

Now its significance should be clear! This is the probability measure on programs that maximizes entropy subject to the constraint that the mean length is some constant ℓ:

$$\sum_{x \in X} p(x)\, |x| = \ell.$$

So, γ is the conjugate variable to program length.

There are, however, other important observables that can be defined for programs, and each of these has a conjugate quantity. To make the analogy to thermodynamics as vivid as possible, let us arbitrarily choose two more observables and treat them as analogues of energy and the

number of some type of molecule. Two of the most obvious observables are 'output' and 'runtime'. Since Levin's computable complexity measure [139] uses the logarithm of runtime as a kind of 'cutoff' reminiscent of an energy cutoff in renormalization, we shall arbitrarily choose the log of the runtime to be analogous to the energy, and denote it as

$$E\colon X \to [0,\infty)$$

Following the chart above, we use $1/T$ to stand for the variable conjugate to E. We arbitrarily treat the output of a program as analogous to the number of a certain kind of molecule, and denote it as

$$N\colon X \to \mathbb{N}.$$

We use $-\mu/T$ to stand for the conjugate variable of N. Finally, as already hinted, we denote program length as

$$V\colon X \to \mathbb{N}$$

so that in terms of our earlier notation, $V(x) = |x|$. We use P/T to stand for the variable conjugate to V.

ALGORITHMS

Observable	Conjugate Variable
log runtime: E	$\frac{1}{T}$
length: V	$\frac{P}{T}$
output: N	$-\frac{\mu}{T}$

Before proceeding, we wish to emphasize that the analogies here were chosen somewhat arbitrarily. They are merely meant to illustrate the application of thermodynamics to the study of algorithms. There may or may not be a specific 'best' mapping between observables for programs and observables for a container of gas! Indeed, Tadaki [196] has explored another analogy, where length rather than log run time is treated as the analogue of energy. There is nothing wrong with this. However, he did not introduce enough other observables to see the whole structure of thermodynamics, as developed in Sections 1.4.1-1.4.2 below.

Having made our choice of observables, we define the partition function by

$$Z = \sum_{x\in X} e^{-\frac{1}{T}(E(x)+PV(x)-\mu N(x))} .$$

When this sum converges, we can define a probability measure on X, the Gibbs ensemble, by

$$p(x) = \frac{1}{Z} e^{-\frac{1}{T}(E(x)+PV(x)-\mu N(x))} .$$

Both the partition function and the probability measure are functions of T, P and μ. From these we can compute the mean values of the observables to which these variables are conjugate:

$$\begin{aligned} \overline{E} &= \sum_{x \in X} p(x)\, E(x) \\ \overline{V} &= \sum_{x \in X} p(x)\, V(x) \\ \overline{N} &= \sum_{x \in X} p(x)\, N(x) \end{aligned}$$

In certain ranges, the map $(T, P, \mu) \mapsto (\overline{E}, \overline{V}, \overline{N})$ will be invertible. This allows us to alternatively think of Z and p as functions of $\overline{E}, \overline{V}$, and $\overline{N}$. In this situation it is typical to abuse language by omitting the overlines which denote 'mean value'.

1.4.1 Elementary Relations

The entropy S of the Gibbs ensemble is given by

$$S = -\sum_{x \in X} p(x)\, \ln p(x).$$

We may think of this as a function of T, P and μ, or alternatively — as explained above — as functions of the mean values E, V, and N. Then simple calculations, familiar from statistical mechanics [164], show that

$$\left.\frac{\partial S}{\partial E}\right|_{V,N} = \frac{1}{T} \tag{1.3}$$

$$\left.\frac{\partial S}{\partial V}\right|_{E,N} = \frac{P}{T} \tag{1.4}$$

$$\left.\frac{\partial S}{\partial N}\right|_{E,V} = -\frac{\mu}{T}. \tag{1.5}$$

We may summarize all these by writing

$$dS = \frac{1}{T} dE + \frac{P}{T} dV - \frac{\mu}{T} dN$$

or equivalently

$$dE = T dS - P dV + \mu dN. \tag{1.6}$$

Starting from the latter equation we see:

$$\left.\frac{\partial E}{\partial S}\right|_{V,N} = T \tag{1.7}$$

$$\left.\frac{\partial E}{\partial V}\right|_{S,N} = -P \tag{1.8}$$

$$\left.\frac{\partial E}{\partial N}\right|_{S,V} = \mu. \tag{1.9}$$

With these definitions, we can start to get a feel for what the conjugate variables are measuring. To build intuition, it is useful to think of the entropy S as roughly the logarithm of the number of programs whose log runtimes, length and output lie in small ranges $E \pm \Delta E$, $V \pm \Delta V$ and $N \pm \Delta N$. This is at best approximately true, but in ordinary thermodynamics this approximation is commonly employed and yields spectacularly good results. That is why in thermodynamics people often say the entropy is the logarithm of the number of microstates for which the observables E, V and N lie within a small range of their specified values [164].

If you allow programs to run longer, more of them will halt and give an answer. The **algorithmic temperature**, T, is roughly the number of times you have to double the runtime in order to double the number of ways to satisfy the constraints on length and output.

The **algorithmic pressure**, P, measures the tradeoff between runtime and length [51]: if you want to keep the number of ways to satisfy the constraints constant, then the freedom gained by having longer runtimes has to be counterbalanced by shortening the programs. This is analogous to the pressure of gas in a piston: if you want to keep the number of microstates of the gas constant, then the freedom gained by increasing its energy has to be counterbalanced by decreasing its volume.

Finally, the **algorithmic potential** describes the relation between log runtime and output: it is a quantitative measure of the principle that most large outputs must be produced by long programs.

1.4.2 Thermodynamic Cycles

One of the first applications of thermodynamics was to the analysis of heat engines. The underlying mathematics applies equally well to algorithmic thermodynamics. Suppose C is a loop in (T, P, μ) space. Assume we are in a region that can also be coordinatized by the variables E, V, N. Then the change in **algorithmic heat** around the loop C is defined to be

$$\Delta Q = \oint_C T dS.$$

Suppose the loop C bounds a surface Σ. Then Stokes' theorem implies that

$$\Delta Q = \oint_C T dS = \int_\Sigma dT dS.$$

However, Equation (1.6) implies that

$$dT dS = d(TdS) = d(dE + PdV - \mu dN) = +dPdV - d\mu dN$$

since $d^2 = 0$. So, we have

$$\Delta Q = \int_\Sigma (dPdV - d\mu dN)$$

or using Stokes' theorem again

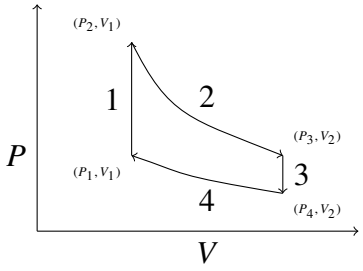
$$\Delta Q = \int_C (PdV - \mu dN). \tag{1.10}$$

In ordinary thermodynamics, N is constant for a heat engine using gas in a sealed piston. In this situation we have

$$\Delta Q = \int_C PdV.$$

This equation says that the change in heat of the gas equals the work done on the gas — or equivalently, minus the work done *by* the gas. So, in algorithmic thermodynamics, let us define $\int_C PdV$ to be the **algorithmic work** done on our ensemble of programs as we carry it around the loop C. Beware: the algorithmic work has the same units as the observable we choose to play the role of internal energy. The algorithmic work is only the same as the 'computational work', meaning the amount of computation done by a program as it runs, if we choose to use the runtime as the analogue of internal energy. We have chosen to use the log runtime instead, so the concepts are related, but not the same. If we had chosen to use the length of a program to represent the internal energy, then algorithmic work would be measured in bits and the two concepts would be completely distinct.

To see an example of a cycle in algorithmic thermodynamics, consider the analogue of the heat engine patented by Stoddard in 1919 [192]. Here we fix N to a constant value and consider the following loop in the PV plane:



We start with an ensemble with algorithmic pressure P_1 and mean length V_1. We then trace out a loop built from four parts:

1. *Isometric*. We increase the pressure from P_1 to P_2 while keeping the mean length constant. No algorithmic work is done on the ensemble of programs during this step.

2. *Isentropic*. We increase the length from V_1 to V_2 while keeping the number of halting programs constant. High pressure means that we're operating in a range of runtimes where if we increase the length a little bit, many more programs halt. In order to keep the number of halting programs constant, we need to shorten the runtime significantly. As we gradually increase the length and lower the runtime, the pressure drops to P_3. The total difference in log runtime is the algorithmic work done on the ensemble during this step.

3. *Isometric.* Now we decrease the pressure from P_3 to P_4 while keeping the length constant. No algorithmic work is done during this step.

4. *Isentropic.* Finally, we decrease the length from V_2 back to V_1 while keeping the number of halting programs constant. Since we're at low pressure, we need only increase the runtime a little. As we gradually decrease the length and increase the runtime, the pressure rises slightly back to P_1. The total increase in log runtime is minus the algorithmic work done on the ensemble of programs during this step.

The total algorithmic work done on the ensemble per cycle is the difference in log runtimes between steps 2 and 4.

1.4.3 Further Relations

From the elementary thermodynamic relations in Section 1.4.1, we can derive various others. For example, the so-called 'Maxwell relations' are obtained by computing the second derivatives of thermodynamic quantities in two different orders and then applying the basic derivative relations, Equations (1.7-1.9). While trivial to prove, these relations say some things about algorithmic thermodynamics which may not seem intuitively obvious.

We give just one example here. Since mixed partials commute, we have:

$$\left.\frac{\partial^2 E}{\partial V \partial S}\right|_N = \left.\frac{\partial^2 E}{\partial S \partial V}\right|_N .$$

Using Equation (1.7), the left side can be computed as follows:

$$\left.\frac{\partial^2 E}{\partial V \partial S}\right|_N = \left.\frac{\partial}{\partial V}\right|_{S,N} \left.\frac{\partial E}{\partial S}\right|_{V,N} = \left.\frac{\partial T}{\partial V}\right|_{S,N}$$

Similarly, we can compute the right side with the help of Equation (1.8):

$$\left.\frac{\partial^2 E}{\partial S \partial V}\right|_N = \left.\frac{\partial}{\partial S}\right|_{V,N} \left.\frac{\partial E}{\partial V}\right|_{S,N} = -\left.\frac{\partial P}{\partial S}\right|_{V,N} .$$

As a result, we obtain:

$$\left.\frac{\partial T}{\partial V}\right|_{S,N} = -\left.\frac{\partial P}{\partial S}\right|_{V,N} .$$

We can also derive interesting relations involving derivatives of the partition function. These become more manageable if we rewrite the partition function in terms of the conjugate variables of the observables E, V, and N:

$$\beta = \frac{1}{T}, \quad \gamma = \frac{P}{T}, \quad \delta = -\frac{\mu}{T}. \tag{1.11}$$

Then we have

$$Z = \sum_{x \in X} e^{-\beta E(x) - \gamma V(x) - \delta N(x)}$$

Simple calculations, standard in statistical mechanics [164], then allow us to compute the mean values of observables as derivatives of the logarithm of Z with respect to their conjugate variables. Here let us revert to using overlines to denote mean values:

$$\begin{aligned}
\overline{E} &= \sum_{x \in X} p(x)\, E(x) &&= -\frac{\partial}{\partial \beta} \ln Z \\
\overline{V} &= \sum_{x \in X} p(x)\, V(x) &&= -\frac{\partial}{\partial \gamma} \ln Z \\
\overline{N} &= \sum_{x \in X} p(x)\, N(x) &&= -\frac{\partial}{\partial \delta} \ln Z
\end{aligned}$$

We can go further and compute the variance of these observables using second derivatives:

$$(\Delta E)^2 = \sum_{x \in X} p(x)(E(x)^2 - \overline{E}^2) = \frac{\partial^2}{\partial^2 \beta} \ln Z$$

and similarly for V and N. Higher moments of E, V and N can be computed by taking higher derivatives of $\ln Z$.

1.4.4 Convergence

So far we have postponed the crucial question of convergence: for which values of T, P and μ does the partition function Z converge? For this it is most convenient to treat Z as a function of the variables β, γ and δ introduced in Equation (1.11). For which values of β, γ and δ does the partition function converge?

First, when $\beta = \gamma = \delta = 0$, the contribution of each program is 1. Since there are infinitely many halting programs, $Z(0, 0, 0)$ does not converge.

Second, when $\beta = 0, \gamma = \ln 2$, and $\delta = 0$, the partition function converges to Chaitin's number

$$\Omega = \sum_{x \in X} 2^{-V(x)}.$$

To see that the partition function converges in this case, consider this mapping of strings to segments of the unit interval:

<table>
<tr><td colspan="8">empty</td></tr>
<tr><td colspan="4">0</td><td colspan="4">1</td></tr>
<tr><td colspan="2">00</td><td colspan="2">01</td><td colspan="2">10</td><td colspan="2">11</td></tr>
<tr><td>000</td><td>001</td><td>010</td><td>011</td><td>100</td><td>101</td><td>110</td><td>111</td></tr>
</table>

⋮

Each segment consists of all the real numbers whose binary expansion begins with that string; for example, the set of real numbers whose binary expansion begins $0.101\ldots$ is $[0.101, 0.110)$ and has measure $2^{-|101|} = 2^{-3} = 1/8$. Since the set of halting programs for our universal machine is prefix-free, we never count any segment more than once, so the sum of all the segments corresponding to halting programs is at most 1.

Third, Tadaki has shown [195] that the expression

$$\sum_{x \in X} e^{-\gamma V(x)}$$

converges for $\gamma \geq \ln 2$ but diverges for $\gamma < \ln 2$. It follows that $Z(\beta, \gamma, \delta)$ converges whenever $\gamma \geq \ln 2$ and $\beta, \delta \geq 0$.

Fourth, when $\beta > 0$ and $\gamma = \delta = 0$, convergence depends on the machine. There are machines where infinitely many programs halt immediately. For these, $Z(\beta, 0, 0)$ does not converge. However, there are also machines where program x takes at least $V(x)$ steps to halt; for these machines $Z(\beta, 0, 0)$ will converge when $\beta \geq \ln 2$. Other machines take much longer to run. For these, $Z(\beta, 0, 0)$ will converge for even smaller values of β.

Fifth and finally, when $\beta = \gamma = 0$ and $\delta > 0$, $Z(\beta, \gamma, \delta)$ fails to converge, since there are infinitely many programs that halt and output 0.

1.4.5 Computability

Even when the partition function Z converges, it may not be computable. The theory of computable real numbers was independently introduced by Church, Post, and Turing, and later blossomed into the field of computable analysis [161]. We will only need the basic definition: a real number a is **computable** if there is a recursive function that maps any natural number $n > 0$ to an integer $f(n)$ such that

$$\frac{f(n)}{n} \leq a \leq \frac{f(n) + 1}{n}.$$

In other words, for any $n > 0$, we can compute a rational number that approximates a with an error of at most $1/n$. This definition can be formulated in various other equivalent ways: for example, the computability of binary digits.

Chaitin [56] proved that the number

$$\Omega = Z(0, \ln 2, 0)$$

is uncomputable. In fact, he showed that for any universal machine, the values of all but finitely many bits of Ω are not only uncomputable, but random: knowing the value of some of them tells you nothing about the rest. They're independent, like separate flips of a fair coin.

More generally, for any computable number $\gamma \geq \ln 2$, $Z(0, \gamma, 0)$ is 'partially random' in the sense of Tadaki [49, 195]. This deserves a word of explanation. A fixed formal system with finitely many axioms can only prove finitely many bits of $Z(0, \gamma, 0)$ have the values they do; after that, one has to add more axioms or rules to the system to make any progress. The number Ω is completely random

in the following sense: for each bit of axiom or rule one adds, one can prove at most one more bit of its binary expansion has the value it does. So, the most efficient way to prove the values of these bits is simply to add them as axioms! But for $Z(0, \gamma, 0)$ with $\gamma > \ln 2$, the ratio of bits of axiom per bits of sequence is less than than 1. In fact, Tadaki showed that for any computable $\gamma \geq \ln 2$, the ratio can be reduced to exactly $(\ln 2)/\gamma$.

On the other hand, $Z(\beta, \gamma, \delta)$ is computable for all computable real numbers $\beta > 0$, $\gamma \geq \ln 2$ and $\delta \geq 0$. The reason is that $\beta > 0$ exponentially suppresses the contribution of machines with long runtimes, eliminating the problem posed by the undecidability of the halting problem. The fundamental insight here is due to Levin [139]. His idea was to 'dovetail' all programs: on turn n, run each of the first n programs a single step and look to see which ones have halted. As they halt, add their contribution to the running estimate of Z. For any $k \geq 0$ and turn $t \geq 0$, let k_t be the location of the first zero bit after position k in the estimation of Z. Then because $-\beta E(x)$ is a monotonically decreasing function of the runtime and decreases faster than k_t, there will be a time step where the total contribution of all the programs that have not halted yet is less than 2^{-k_t}.

1.5 Conclusions

There are many further directions to explore. Here we mention just three. First, as already mentioned, the 'Kolmogorov complexity' [129] of a number n is the number of bits in the shortest program that produces n as output. However, a very short program that runs for a million years before giving an answer is not very practical. To address this problem, the **Levin complexity** [140] of n is defined using the program's length plus the logarithm of its runtime, again minimized over all programs that produce n as output. Unlike the Kolmogorov complexity, the Levin complexity is computable. But like the Kolmogorov complexity, the Levin complexity can be seen as a *relative entropy*—at least, up to some error bounded by a constant. The only difference is that now we compute this entropy relative to a different probability measure: instead of using the Gibbs distribution at infinite algorithmic temperature, we drop the temperature to $\ln 2$. Indeed, the Kolmogorov and Levin complexities are just two examples from a continuum of options. By adjusting the algorithmic pressure and temperature, we get complexities involving other linear combinations of length and log runtime. The same formalism works for complexities involving other observables: for example, the maximum amount of memory the program uses while running.

Second, instead of considering Turing machines that output a single natural number, we can consider machines that output a finite list of natural numbers $(N_1, \ldots, N_j)$; we can treat these as populations of different "chemical species" and define algorithmic potentials for each of them. Processes analogous to chemical reactions are paths through this space that preserve certain invariants of the lists. With chemical reactions we can consider things like internal combustion cycles.

Finally, in ordinary thermodynamics the partition function Z is simply a number after we fix values of the conjugate variables. The same is true in algorithmic thermodynamics. However, in algorithmic thermodynamics, it is natural to express this number in binary and inquire about the algorithmic entropy of the first n bits. For example, we have seen that for suitable values of temperature, pressure and chemical potential, Z is Chaitin's number Ω. For each universal machine there

exists a constant c such that the first n bits of the number Ω have at least $n - c$ bits of algorithmic entropy with respect to that machine. Tadaki [195] generalized this computation to other cases.

So, *in algorithmic thermodynamics, the partition function itself has nontrivial entropy*. Tadaki has shown that the same is true for algorithmic pressure (which in his analogy he calls 'temperature'). This reflects the self-referential nature of computation.

The bits of Ω are uncomputable; we can never be *sure* that a computed estimate is correct in its initial digits. However, in the next chapter we show that we can make the probability that we are wrong an arbitrarily small positive number.

Chapter 2

Critical time

In the last chapter, we looked at partition functions and how they relate the entropy of physical systems like a piston of gas to the algorithmic entropy of a particular output of a computer. A special output of the partition function for a prefix-free universal Turing machine is the machine's "Ω number," the probability that it will run to completion and produce an output, as opposed to failing with a syntax error or going into an infinite loop. Given the first n digits of a machine's Ω number, there is a program that will compute whether or not any program up to n bits long halts or not. However, the bits of the Ω number for a prefix-free universal Turing machine are uncomputable; past a finite initial segment, there is no way to know for certain that the bits are correct.

The aim of this chapter is to provide a probabilistic, but non-quantum, analysis of the halting problem. Our approach is to have the probability space extend over both space and time and to consider the probability that a random N-bit program has halted by a random time. We postulate *an a priori computably enumerable probability distribution on all possible runtimes* and we prove that given an integer $k > 0$, we can effectively compute a time bound T such that the probability that an N-bit program will eventually halt given that it has not halted by T is smaller than 2^{-k}.

We also show that the set of halting programs (which is computably enumerable, but not computable) can be written as a disjoint union of a computable set and a set of effectively vanishing probability.

Finally, we show that "long" runtimes are effectively rare. More formally, the set of times at which an N-bit program can stop after the time $2^{N+\text{constant}}$ has effectively zero density.

2.1 Introduction

The Halting Problem for Turing machines is to decide whether an arbitrary Turing machine M eventually halts on an arbitrary input x. As a Turing machine M can be coded by a finite string—say, $code(M)$—one can ask whether there is a Turing machine M_{halt} which, given $code(M)$ and the input x, eventually stops and produces 1 if $M(x)$ stops, and 0 if $M(x)$ does not stop. Turing's

famous result states that this problem cannot be solved by any Turing machine, i.e. there is no such M_{halt}. Halting computations can be recognised by simply running them; the main difficulty is to detect non-halting programs. In what follows time is discrete.

Since many real-world problems arising in the fields of compiler optimisation, automatised software engineering, formal proof systems, and so forth are equivalent to the Halting Problem, there is a strong interest—not merely academic!—in understanding the problem better and in providing alternative solutions.

There are two approaches we can take to calculating the probability that an N-bit program will halt. The first approach, initiated by Chaitin [56], is to have the probability space range only over programs; this is the approach taken when computing the Omega number, [45, 43]. In that case, the probability is $Prob_N = \#\{p \in \{0,1\}^N \mid p \text{ halts}\}/2^N$. For a prefix-free machine, $Prob_N$ goes to zero when N tends to infinity, since it becomes more and more likely that any given N-bit string is an extension of a shorter halting program. For a universal non-prefix-free Turing machine, the probability is always nonzero for large enough N: after some point, the universal non-prefix-free Turing machine will simulate a total Turing machine (one that halts on all inputs), so some fixed proportion of the space will always contribute. The probability in this case is uncomputable, machine-dependent; in general, 1 is the best computable upper bound one can find. In this approach it matters only whether a program halts or not; the time at which a halting program stops is irrelevant.

Our approach is to have the probability space extend over both space and time, and to consider the probability that a random N-bit program—which has not stopped by some given time—will halt by a random later time. In this approach, the stopping time of a halting program is paramount. The problem is that there is no uniform distribution on the integers, so we must choose some kind of distribution on times as well. Any distribution we choose must have that most long times are rare, so in the limit, which distribution we choose does not matter very much.

The new approach was proposed by Calude and Pavlov [48] (see also [7]) where a mathematical quantum "device" was constructed to probabilistically solve the Halting Problem. The procedure has two steps: a) based on the length of the program and an *a priori* admissible error 2^{-k}, a finite time T is effectively computed, b) a quantum "device", designed to work on a randomly chosen test-vector is run for the time T; if the device produces a click, then the program halts; otherwise, the program probably does not halt, with probability higher than $1 - 2^{-k}$. This result uses an unconventional model of quantum computing, an infinite dimensional Hilbert space. This quantum proposal has been discussed in Ziegler [206].

It is natural to ask whether the quantum mechanics machinery is essential for obtaining the result. In this thesis we discuss a method to "de-quantize" the algorithm. We have been motivated by some recent approximate solutions to the Halting Problem obtained by Köhler, Schindelhauer and M. Ziegler [127] and experimental work [45, 135].* Different approaches were proposed by Hamkins and Miasnikov [92], and D'Abramo [70].

Our working hypothesis, crucial for this approach, is to postulate *an a priori computably enu-*

*For example, Langdon and Poli [135] suggest that, for a specific universal machine that they describe, about $N^{-1/2}$ programs of length N halt.

merable probability distribution on all possible runtimes. Consequently, the probability space is the product of the space of programs of fixed length (or of all possible lengths), where programs are uniformly distributed, and the time space, which is discrete and has an *a priori* computable probability distribution. In this context we show that given an integer $k > 0$, we can effectively compute a time bound T such that the probability on the product space that an N-bit program will eventually halt given that it not stopped by T is smaller than 2^{-k}. This phenomenon is similar to the one described for proofs in formal axiomatic systems [46].

We also show that for every integer $k > 0$, the set of halting programs (which is computably enumerable, but not computable) can be written as a disjoint union of a computable set and a set of probability effectively smaller than 2^{-k}.

Of course, an important question is to what extent the postulated hypothesis is acceptable or realistic. The next part of the chapter deals with this question offering an argument in favor of the hypothesis. First we note that for any computably enumerable probability distribution most long times are effectively rare, so in the limit they all have the same behavior irrespective of the choice of the distribution. Our second argument is based on the common properties of the times when programs may stop. Our proof consists of three parts: a) the exact time a program stops is algorithmically not too complicated; it is (algorithmically) nonrandom because most programs either stop 'quickly' or never halt, b) an N-bit program which has not stopped by time $2^{N+\text{constant}}$ cannot halt at a later random time, c) since nonrandom times are (effectively) rare, the density of times an N-bit program can stop vanishes.

We will start by examining a particular case in detail, the behavior of all programs of length 3 for a certain Turing machine. This case study will describe various possibilities of halting/non-halting programs, the difference between a program stopping exactly *at* a time and a program stopping *by* some time, as well as the corresponding probabilities for each such event.

Finally, we show some differences between the halting probabilities for different types of universal machines.

Some comments will be made about the "practicality" of the results presented in this chapter: can we use them to approximately solve any mathematical problems?

2.2 A case study

We consider all programs of length $N = 3$ for a simple Turing machine M whose domain is the finite set $\{000, 010, 011, 100, 110, 111\}$. The halting "history" of these programs, presented in Table 1, shows the times at which the programs in the domain of M halt. The program $M(p_1)$ halts at time $t = 1$, so it is indicated by an "h" on the row corresponding to p_1 and time $t = 1$; the program $M(p_4)$ has not halted at time $t = 5$, so on the row corresponding to p_4 and time $t = 4$ there is a blank. Finally, programs which have not stopped by time $t = 17$ never stop.

Program/time	$t = 1$	$t = 2$	$t = 5$	$t = 8$	...	$t = 14$	$t = 15$	$t = 16$	$t = 17$
$p_1 = 000$	h	h	h	h	h	h	h	h	h
$p_2 = 001$									
$p_3 = 010$							h	h	h
$p_4 = 011$				h	h	h	h	h	h
$p_5 = 100$						h	h	h	h
$p_6 = 101$									
$p_7 = 110$	h	h	h	h	h	h	h	h	h
$p_8 = 111$								h	h

Table 1: Halting "history" for 3-bit programs.

Here are a few miscellaneous facts we can derive from Table 1:

- the program $M(p_1)$ halts exactly at time $t = 1$,
- the set of 3-bit programs which halt exactly at time $t = 1$ consists of $\{p_1, p_7\}$, so the 'probability' that a randomly chosen 3-bit program halts at time $t = 1$ is

$$\frac{\#\{\text{3-bit programs halting at time 1}\}}{\#\{\text{3-bit programs}\}} = \frac{\#\{p1, p7\}}{8} = \frac{2}{8} = \frac{1}{4},$$

- the set of 3-bit programs which halt by time $t = 8$ consists of $\{p_1, p_4, p_7\}$, so the 'probability' that a randomly picked 3-bit program halts by time $t = 8$ is

$$\frac{\#\{\text{3-bit programs halting by time 8}\}}{\#\{\text{3-bit programs}\}} = \frac{\#\{p1, p4, p7\}}{8} = \frac{3}{8},$$

- the 'probability' that a random 3-bit program eventually stops is

$$\#\{\text{3-bit programs that halt}\}/\#\{\text{3-bit programs}\} = \frac{6}{8},$$

- the program $M(p_4)$ has not stopped by time $t = 5$, but stops at time $t = 8$,
- the 'probability' that a 3-bit program does not stop by time $t = 5$ and that it eventually halts is

$$\frac{\#\{\text{3-bit programs that eventually halt that have not stopped by time } t = 5\}}{\#\{\text{3-bit programs}\}}$$
$$= \frac{\#\{p3, p4, p5, p8\}}{8} = \frac{4}{8} = \frac{1}{2},$$

- the 'probability' that a 3-bit program eventually stops given that it has not halted by time $t = 5$ is

$$\frac{\#\{\text{3-bit programs that eventually halt that have not stopped by time } t = 5\}}{\#\{\text{3-bit programs that have not halted by time } t = 5\}} = \frac{4}{8-2} = \frac{2}{3},$$

- the 'probability' that a 3-bit program halts at time $t = 8$ given that it has not halted by time $t = 5$ is

$$\frac{\#\{\text{3-bit programs that halt by time } t = 8 \text{ but not by time } t = 5\}}{\#\{\text{3-bit programs that have not halted by time } t = 5\}} = \frac{1}{6}.$$

We can express the above facts in a bit more formal way as follows. We fix a universal Turing machine U (see section 3 for a definition) and a pair (N, T), where N represents the length of the program and T is the "time-window", i.e. the interval $\{1, 2, \ldots, T\}$, where the computational time is observed. The probability space is thus

$$\text{Space}_{N,T} = \{0, 1\}^N \times \{1, 2, \ldots, T\}.$$

Both programs and times are assumed to be *uniformly distributed*. For $A \subseteq \text{Space}_{N,T}$ we define $\text{Prob}_{N,T}(A)$ to be $\#(A) \cdot 2^{-N} \cdot T^{-1}$.

Define

$$A_{N,T} = \{(p, t) \in \text{Space}_{N,T} \mid U(p) \text{ stops exactly at time } t\},$$

and

$$B_{N,T} = \{(p, t) \in \text{Space}_{N,T} \mid U(p) \text{ stops by time } t\}.$$

Fact 1 *We have:* $\text{Prob}_{N,T}(A_{N,T}) \leq \frac{1}{T}$ *and* $\text{Prob}_{N,T}(B_{N,T}) \leq 1$.

Proof. It is easy to see that $\#(A) \leq 2^N$, consequently,

$$\text{Prob}_{N,T}(A_{N,T}) = \frac{\#(A_{N,T})}{2^N \cdot T} \leq \frac{2^N}{2^N \cdot T} = \frac{1}{T}, \ \text{Prob}_{N,T}(B_{N,T}) \leq \frac{\#(B_{N,T})}{2^N \cdot T} \leq \frac{2^N \cdot T}{2^N \cdot T} = 1.$$

□

Comment. The inequality $\text{Prob}_{N,T}(B_{N,T}) \leq 1$ does not seem to be very informative. However, for all N, one can construct a universal Turing machine U_N such that $\text{Prob}_{N,T}(B) = 1$; U_N cannot be prefix-free (see, for a definition, section 4). There is no universal Turing machine U such that $\text{Prob}_{N,T}(B) = 1$, for all N, so can we do better than stated in Fact 1?

More precisely, we are interested in the following problem:

> *We are given a universal Turing machine U and a randomly chosen program p of length N that we know not to stop by time t. Can we effectively evaluate the 'probability' that $U(p)$ eventually stops?*

An obvious way to proceed is the following. Simply, run in parallel all programs of length N till the time $T_N = \max\{t_p \mid |p| = N, U(p) \text{ halts}\} = \min\{t \mid \text{for all } |p| = N, t_p \leq t\}$, where t_p is the exact time $U(p)$ halts (if indeed it stops). In other words, get the analogue of the Table 1 for U and N, and then calculate directly all probabilities. This method, as simple as it may look, is not very useful, since the time T_N is *not computable* because of the undecidability of the Halting Problem.

Can we overcome this serious difficulty?

2.3 Notation

All strings are binary and the set of strings is denoted by $\{0,1\}^*$. The length of the string x is denoted by $|x|$. The logarithms are binary too. Let $\mathbb{N} = \{1, 2, \ldots\}$ and let $\text{bin} : \mathbb{N} \to \{0,1\}^*$ be the computable bijection which associates to every $n \geq 1$ its binary expansion without the leading 1,

n	n_2	bin(n)	|bin(n)|
1	1	λ	0
2	10	0	1
3	11	1	1
4	100	00	2
$\vdots$	$\vdots$	$\vdots$	$\vdots$

We will work with Turing machines M which process strings into strings. The domain of M, dom(M), is the set of strings on which M halts (is defined). The *natural complexity* [52] of the string $x \in \{0,1\}^*$ (with respect to M) is $\nabla_M(x) = \min\{n \geq 1 \mid M(\text{bin}(n)) = x\}$. The Invariance Theorem [43] has the following form: we can effectively construct a machine U (called *universal*) such that for every machine M, there is a constant $\varepsilon > 0$ (depending on U and M) such that $\nabla_U(x) \leq \varepsilon \cdot \nabla_M(x)$, for all strings x. For example, if $U(0^i 1x) = M_i(x)$ (where (M_i) is an effective enumeration of all Turing machines), then $\nabla_U(x) \leq (2^{i+1} + 1) \cdot \nabla_{M_i}(x)$, because $0^i 1\text{bin}(m) = \text{bin}(2^{i+1+\lfloor \log(m) \rfloor} + m)$, for all $m \geq 1$. In what follows we will fix a universal Turing machine U and we will write ∇ instead of ∇_U. There are some advantages in working with the complexity ∇ instead of the classical complexity K (see [43]); for example, for every $N > 0$, the inequality $\#\{x \in \{0,1\}^* : \nabla(x) < N\} \leq N$ is obvious; a better example appears in [50] where ∇ is a more natural measure to investigate the relation between incompleteness and uncertainty.

2.4 Halting according to a computably enumerable time distribution

We postulate *an a priori computably enumerable probability distribution on all possible runtimes*. Consequently, the probability space is the product of the space of programs—either taken to be all programs of a fixed length, where programs are uniformly distributed, or to be all programs of all possible lengths, where the distribution depends on the length—and the time space, which is discrete and has an *a priori* computably enumerable probability distribution.

In what follows we randomly choose a time i according to a probability distribution $\rho(i)$ which effectively converges to 1; that is, there exists a computable function B such that for every $n \geq B(k)$,

$$|1 - \sum_{i=1}^{n} \rho(i)| < 2^{-k}.$$

How long does it take for an N-bit program p to run without halting on U to conclude that the probability that $U(p)$ eventually stops is less than 2^{-k}?

It is not difficult to see that the probability that an N-bit program which has not stopped on U by time t_k (which can be effectively computed) will eventually halt is not larger than $\sum_{i \geq t_k} \rho(i)$, which effectively converges to 0, that is, there is a computable function $b(k)$ such that for $n \geq b(k)$, $\sum_{i \geq n} \rho(i) < 2^{-k}$.

The probability distribution $\rho(i)$ may or may not be related to the computational runtime of a program for U. Here is an example of a probability distribution which effectively converges to 1 and relates the observer time to the computational runtime. This probability distribution is reminiscent of Chaitin's halting probability [43].

The idea is to define the distribution at moment i to be 2^{-i} divided by the exact time it takes $U(\text{bin}(i))$ to halt, or to be 0 if $U(\text{bin}(i))$ does not halt. Recall that t_p is the exact time $U(p)$ halts (or $t_p = \infty$ when $U(p)$ does not halt).

First we define the number

$$\Upsilon_U = \sum_{i \geq 1} 2^{-i} / t_{\text{bin}(i)}.$$

It is clear that $0 < \Upsilon_U < 1$. Moreover, Υ_U is computable. Indeed, we construct an algorithm computing, for every positive integer n, the nth digit of Υ_U. The proof is simple: only the terms $2^{-i}/t_{\text{bin}(i)}$ for which $U(\text{bin}(i))$ does not halt, i.e. $t_{\text{bin}(i)} = \infty$, produce 'false' information because at every finite step of the computation they appear to be non-zero when, in fact, they are zero! The solution is to run all non-stopping programs $U(\text{bin}(i))$ for enough time such that their cumulative contribution is too small to affect the nth digit of Υ_U: indeed, if $n > 2$, and $t_{\text{bin}(i)} = 1$, for $i \geq n$, then $\sum_{i=n}^{\infty} 2_{-i}/t_{\text{bin}(i)} < 2^{-n}$.

So, Υ_U induces a natural probability distribution on the runtime: to i we associate[†]

$$\rho(i) = \frac{2^{-i}}{t_{\text{bin}(i)} \cdot \Upsilon_U}. \tag{2.1}$$

The probability space is

$$\text{Space}_{N,\{\rho(i)\}} = \{0,1\}^N \times \{1,2,\ldots\},$$

where N-bit programs are assumed to be uniformly distributed, and we choose at random a runtime from distribution (2.1).

Theorem 2 *Assume that $U(p)$ has not stopped by time $T > k - \lfloor \log \Upsilon_U \rfloor$. Then, the probability (according to the distribution (2.1)) that $U(p)$ eventually halts is smaller than 2^{-k}.*

Proof. It is seen that

$$\frac{1}{\Upsilon_U} \sum_{i=T}^{\infty} \frac{2^{-i}}{t_{\text{bin}(i)}} \leq \frac{1}{\Upsilon_U \cdot 2^{T-1}} < 2^{-k},$$

for $T > k - \lfloor \log \Upsilon_U \rfloor$. The bound is computable because Υ_U is computable.

□

[†]Of course, instead of $2^{-i}/t_{\text{bin}(i)}$ we can take $r_i/t_{\text{bin}(i)}$, where $\sum_{i \geq 1} r_i < \infty$, effectively.

We now consider the probability space to be

$$\text{Space}_{\{\rho(i)\}} = \{0,1\}^* \times \{1,2,\ldots\},$$

where N-bit programs are assumed to be uniformly distributed, and the runtime is chosen at random from the computably enumerable probability distribution $\{\rho(i)\}$.

Theorem 3 *Assume that U and $\text{Space}_{\{\rho(i)\}}$ have been fixed. For every integer $k > 0$, the set of halting programs for U can be written as a disjoint union of a computable set and a set of probability effectively smaller than 2^{-k}.*

Proof. Let b be a computable function such that for $n \geq b(k)$ we have $\sum_{i \geq n} \rho(i) < 2^{-k}$. The set of halting programs for U can be written as a disjoint union of the computable set $\{(p, t_p) \mid t_p < 2^{b(k+|p|+2)}\}$ and the set $\{(p, t_p) \mid 2^{b(k+|p|+2)} \leq t_p < \infty\}$. The last set has probability effectively less than

$$\sum_{N=1}^{\infty} \sum_{n=b(k+N+2)}^{\infty} \rho_n \leq \sum_{N=1}^{\infty} 2^{-N-k-2} = 2^{-k-1}.$$

□

Comment. A stronger (in the sense that the computable set is even polynomially decidable), but machine-dependent, decomposition theorem for the set of halting programs was proved in [92].

2.5 How long does it take for a halting program to stop?

The common wisdom says that it is possible to write short programs which stop after a very long time. However, it is less obvious that there are only a few such programs; these programs are "exceptions".

Working with prefix-free Turing machines, Chaitin [58] has given the following estimation of the complexity[‡] of the runtime of a program which eventually halts: there is a constant c such that if $U(\text{bin}(i))$ first halts in time t[§], then

$$\nabla(\text{bin}(t)) \leq 2^{|\text{bin}(i)|} \cdot c \leq i \cdot c. \tag{2.2}$$

The above relation puts a limit on the *complexity* of the time t a program $\text{bin}(i)$, that eventually halts on U, has to run before it stops; this translates into a limit on the time t because only finitely many strings have complexity bounded by a constant. In view of (2.2), *the bound depends only upon the length of the program*; the program itself (i.e. $\text{bin}(i)$) does not matter.

Because $\lim_{t \to \infty} \nabla(\text{bin}(t)) = \infty$, there are only finitely many integers t satisfying the inequality (2.2). That is, there exists a critical value $T_{critical}$ (depending upon U and $|\text{bin}(i)|$) such that if for each $t < T_{critical}$, $U(\text{bin}(i))$ does not stop in time t, then $U(\text{bin}(i))$ never halts. In other words,

[‡]Chaitin used program-size complexity.

[§]Of course, if $U(\text{bin}(i))$ halts in time t, it stops also in time $t' > t$, but only finitely many t' satisfy the inequality (2.2). For the reader more familiar with the program-size complexity H—with repsect to a universal prefix-free Turing machine [43]—the inequality (2.2) corresponds to $H(\text{bin}(t)) \leq |\text{bin}(i)| + c$.

if $U(\mathrm{bin}(i))$ does not stop in time $T_{critical}$, then $U(\mathrm{bin}(i))$ never halts.

So, what prevents us from running the computation $U(\mathrm{bin}(i))$ for the time $T_{critical}$ and deciding whether it halts or not? Obviously, the uncomputability of $T_{critical}$. Neither the natural complexity ∇ nor any other size complexity, like K or H, is computable (see [43]). Obviously, there are large integers t with small complexity $\nabla(\mathrm{bin}(t))$, but they cannot be effectively "separated" because we cannot effectively compute a bound $b(k)$ such that $\nabla(\mathrm{bin}(t)) > k$ whenever $t > b(k)$.

The above analysis suggests that *a program that has not stopped after running for a long time has smaller and smaller chances to eventually stop*. The bound (2.2) is not computable. Still, can we "extract information" from the inequality (2.2) to derive a computable probabilistic description of this phenomenon?

Without loss of generality, we assume that the universal Turing machine U has a built-in counting instruction. Based on this, there is an effective transformation which for each program p produces a new program $time(p)$ such that there is a constant $c > 0$ (depending upon U) for which the following three conditions are satisfied:

1. $U(p)$ halts iff $U(time(p))$ halts,
2. $|time(p)| \leq |p| + c$,
3. if $U(p)$ halts, then it halts at the step $t_p = \mathrm{bin}^{-1}(U(time(p)))$.

Intuitively, $time(p)$ either calculates the number of steps t_p till $U(p)$ halts and prints $\mathrm{bin}(t_p)$, or, if $U(p)$ is not defined, never halts. The constant c can be taken to be less than or equal to 2, as the counting instruction is used only once, and we need one more instruction to print its value; however, we do not need to print the value $U(p)$.

We continue with a proof of the inequality (2.2) for an arbitrary universal Turing machine.

Theorem 4 *Assume that $U(p)$ stops at time t_p, exactly. Then,*

$$\nabla(bin(t_p)) \leq 2^{|p|+c+1}. \tag{2.3}$$

Proof. First we note that for every program p of length at most N, $\mathrm{bin}^{-1}(p) < 2^{N+1}$. Indeed, $|p| = |\mathrm{bin}(\mathrm{bin}^{-1}(p))| \leq N$ implies

$$2^{|p|} \leq \mathrm{bin}^{-1}(p) < 2^{|p|+1} \leq 2^{N+1}. \tag{2.4}$$

Since $U(p) = U(\mathrm{bin}(\mathrm{bin}^{-1}(p)))$ we have:

$$\nabla(U(p)) = \min\{i \geq 1 \ : \ U(\mathrm{bin}(i)) = U(p)\} \leq \mathrm{bin}^{-1}(p),$$

hence

$$\nabla(\mathrm{bin}(t_p)) = \nabla(U(time(p))) \leq \mathrm{bin}^{-1}(time(p)) < 2^{|p|+c+1},$$

because $|time(p)| \leq |p| + c$ and (2.4). □

2.6 Can a program stop at an algorithmically random time?

In this section we prove that no program of length $N \geq 2$ which has not stopped by time $2^{2N+2c+1}$ will stop at an algorithmically random time. Consequently, since algorithmically nonrandom times are (effectively) rare, there are only a few times an N-bit program can stop in a suitably large range. As a consequence, the set of times at which an N-bit program can stop after the time $2^{N+\text{constant}}$ has effectively zero density.

A binary string x is "algorithmically random" if $\nabla(x) \geq 2^{|x|}/|x|$. Most binary strings of a given length n are algorithmically random because they have high density: $\#\{x \in \{0,1\}^* : |x| = n, \nabla(x) \geq 2^n/n\} \cdot 2^{-n} \geq 1 - 1/n$ which tends to 1 when $n \to \infty$.[¶]

A time t will be called "algorithmically random" if $\text{bin}(t)$ is algorithmically random.

Theorem 5 *Assume that an N-bit program p has not stopped on U by time $2^{2N+2c+1}$, where $N \geq 2$ and c comes from Theorem 4. Then, $U(p)$ cannot exactly stop at any algorithmically random time $t \geq 2^{2N+2c+1}$.*

Proof. First we prove that for every $n \geq 4$ and $t \geq 2^{2n-1}$, we have:

$$2^{|\text{bin}(t)|} > 2^n \cdot |\text{bin}(t)|. \tag{2.5}$$

Indeed, the real function $f(x) = 2^x/x$ is strictly increasing for $x \geq 2$ and tends to infinity when $x \to \infty$. Let $m = |\text{bin}(t)|$. As $2^{2n-1} \leq t < 2^{m+1}$, it follows that $m \geq 2n - 1$, hence $2^m/m \geq 2^{2n-1}/(2n-1) \geq 2^n$. The inequality is true for every $|\text{bin}(t)| \geq 2n - 1$, that is, for every $t \geq 2^{2n-1}$.

Next we take $n = N + c + 1$ in (2.5) and we prove that every algorithmically random time $t \geq 2^{2N+2c+1}$, $N \geq 2$, does not satisfy the inequality (2.3). Consequently, no program of length N which has not stopped by time $2^{2N+2c+1}$ will stop at an algorithmically random time.

□

A time t is called "exponential stopping time" if there is a program p which stops on U exactly at $t = t_p > 2^{2|p|+2c+1}$. How large is the set of exponential stopping times? To answer this question we first need a technical result.

Lemma 6 *Let $m \geq 3$, $s \geq 1$. Then*

$$\frac{1}{2^s - 1} \cdot \sum_{i=0}^{s} \frac{2^i}{m+i} < \frac{5}{m+s-1}.$$

Proof. Let us denote by x_s^m the left-hand side of the inequality below. It is easy to see that

$$x_{s+1}^m = \frac{2^s - 1}{2^{s+1} - 1} \cdot x_s^m + \frac{2^{s+1}}{m+s+1} \leq \frac{x_s^m}{2} + \frac{2}{m+s+1}.$$

[¶]In the language of program-size complexity, x is "algorithmically random" if $H(x) \geq |x| - \log(|x|)$.

Next we prove by induction (on s) the inequality in the statement of the lemma. For $s = 1$ we have $x_1^m = 1/m + 2/(m+1) < 5/m$. Assume that $x_s^m < 5/(m+s-1)$. Then:

$$x_{s+1}^m \leq \frac{x_s^m}{2} + \frac{2}{m+s+1} < \frac{5}{2(m+s-1)} + \frac{2}{m+s+1} \leq \frac{5}{m+s}.$$

□

The density of times in the set $\{1, 2, \ldots, N\}$ satisfying the property P is the ratio $\#\{t \mid 1 \leq t \leq N, P(t)\}/N$. A property P of times has "effective zero density" if the density of times satisfying the property P effectively converges to zero, that is, there is a computable function $B(k)$ such that for every $N > B(k)$, the density of times satisfying the property P is smaller than 2^{-k}.

Theorem 7 *For every length N, we can effectively compute a threshold time θ_N (which depends on U and N) such that if a program of length N runs for time θ_N without halting, then the density of times greater than θ_N at which the program can stop has effective zero density. More precisely, if an N-bit program runs for time $T > \max\{\theta_N, 2^{2+5\cdot 2^k}\}$, then the density of times at which the program can stop is less than 2^{-k}.*

Proof. We choose the bound $\theta_N = 2^{2N+2c+1} + 1$, where c comes from (2.3). Let $T > \theta_N$ and put $m = 2N + 2c + 1$, and $s = \lfloor \log(T+1) \rfloor - m$. Then, using Theorem 5 and Lemma 6, we have:

$$\begin{aligned}
&\frac{1}{T - 2^m + 1} \cdot \#\left\{2^m \leq t \leq T \mid \nabla(\mathrm{bin}(t)) \geq \frac{2^{|\mathrm{bin}(t)|}}{|\mathrm{bin}(t)|}\right\} \\
&\quad \geq \frac{1}{T - 2^m + 1} \cdot \sum_{i=0}^{s} \#\left\{2^{m+i} \leq t \leq 2^{m+i+1} - 1 \mid \nabla(\mathrm{bin}(t)) \geq \frac{2^{|\mathrm{bin}(t)|}}{|\mathrm{bin}(t)|}\right\} \\
&\quad = \frac{1}{T - 2^m + 1} \cdot \sum_{i=0}^{s} \#\left\{2^{m+i} \leq t \leq 2^{m+i+1} - 1 \mid \nabla(\mathrm{bin}(t)) \geq \frac{2^{m+i}}{m+i}\right\} \\
&\quad \geq \frac{1}{T - 2^m + 1} \cdot \sum_{i=0}^{s} 2^{m+i}\left(1 - \frac{1}{m+i}\right) \\
&\quad \geq 1 - \frac{1}{T - 2^m + 1} \cdot \sum_{i=0}^{s} \frac{2^{m+i}}{m+i} \\
&\quad \geq 1 - \frac{1}{(2^s - 1)} \cdot \sum_{i=0}^{s} \frac{2^i}{m+i} \\
&\quad > 1 - \frac{5}{m+s-1},
\end{aligned}$$

consequently, the density of algorithmically random times effectively converges to 1:

$$\lim_{T\to\infty} \frac{\#\{t \mid t > \theta_N, t \le T, t \ne t_p, \text{ for all } p \text{ with } |p| = N\}}{T - 2^m + 1}$$

$$\geq \lim_{T\to\infty} \frac{1}{T - 2^m + 1} \cdot \#\left\{2^m \le t \le T \mid \nabla(\text{bin}(t)) \geq \frac{2^{|\text{bin}(t)|}}{|\text{bin}(t)|}\right\} = 1,$$

so the density of times greater than θ_N at which an N-bit program can stop effectively converges to zero. □

The next result states that "almost any" time is not an exponential stopping time.

Corollary 8 *The set of exponential stopping times has effective zero density.*

Proof. It is seen that

$$\begin{aligned}
&\{t \mid t = t_p, \text{ for some } p \text{ with } t > 2^{2|p|+2c+1}\} \\
&\quad\subseteq \bigcup_{N\geq 1}\left\{t \mid t > 2^{2|p|+2c-1}, |p| = N, \nabla(\text{bin}(t)) < \frac{2^{|\text{bin}(t)|}}{|\text{bin}(t)|}\right\} \\
&\quad\subseteq \left\{t \mid t > 2^{2c+1}, \nabla(\text{bin}(t)) < \frac{2^{|\text{bin}(t)|}}{|\text{bin}(t)|}\right\},
\end{aligned}$$

which has effectively zero density in view of Theorem 7.

□

2.7 Halting probabilities for different universal machines

In this section we show a significant difference between the halting probability of a program of a given length for a universal Turing machine and for a universal prefix-free Turing machine: in the first case the probability is always positive, while in the second case the probability tends to zero when the length tends to infinity.

The probability that an arbitrary string of length N belongs to $A \subset \{0,1\}^*$ is $\text{Prob}_N(A) = \#(A \cap \{0,1\}^N) \cdot 2^{-N}$, where $\{0,1\}^N$ is the set of N-bit strings.

If U is a universal Turing machine, then the probability that an N-bit program p halts on U is $\text{Prob}_N(\text{dom}(U))$.

Fact 9 *Let U be a universal Turing machine. Then,* $\lim_{N\to\infty} \text{Prob}_N(\text{dom}(U)) > 0$.

Proof. We consider the universal Turing machine $U(0^i x) = T_i(x)$, described in section 3. If N is sufficiently large, then there exists an $i < N$ such that T_i is a total function, i.e. T_i is defined on each input, so $\#(\mathrm{dom}(U) \cap \{0,1\}^N) \geq 2^{N-i-1}$. For all such N's, $\mathrm{Prob}_N(\mathrm{dom}(U)) \geq 2^{-i-1} > 0$. The result extends to any universal Turing machine because of universality.

□

A convergent machine V is a Turing machine such that its ζ number is finite:

$$\zeta_V = \sum_{\mathrm{bin}(n) \in \mathrm{dom}(V)} 1/n < \infty,$$

see [52]. The Ω number of V is $\Omega_V = \sum_{N=1}^{\infty} \mathrm{Prob}_N(\mathrm{dom}(V))$. Because $\zeta_V < \infty$ if and only if $\Omega_V < \infty$, see [52], we get:

Fact 10 *Let V be a convergent machine. Then,* $\lim_{N\to\infty} \mathrm{Prob}_N(\mathrm{dom}(V)) = 0$.

Recall that a prefix-free Turing machine V is a machine with a prefix-free domain. For such a machine, $\Omega_V < 1$, hence we have:

Corollary 11 *Let V be a universal prefix-free Turing machine. Then* $\lim_{N\to\infty} \mathrm{Prob}_N(\mathrm{dom}(V)) = 0$.

The probability that an N-bit program never stops on a convergent Turing machine tends to one when N tends to infinity; this is not the case for a universal Turing machine.

2.8 Final comments

We studied the halting probability using a new approach, namely we considered the probability space extend over both space and time, and the probability that a random N-bit program will halt by a random later time given that it has not stopped by some threshhold time. We postulated *an a priori computably enumerable probability distribution on all possible runtimes*. Consequently, the probability space is the product of the space of programs—either taken to be all programs of a fixed length, where programs are uniformly distributed, or to be all programs of all possible lengths, where the distribution depends on the length—and the time space, which is discrete and has an a priori computable probability distribution. We proved that given an integer $k > 0$, we can effectively compute a time bound T such that the probability that an N-bit program will eventually halt, given that it has not stopped by time T, is smaller than 2^{-k}.

We also proved that the set of halting programs (which is computably enumerable, but not computable) can be written as a disjoint union of a computable set and a set of probability effectively smaller than any fixed bound.

Finally we showed that runtimes much longer than the lengths of their respective halting programs are (effectively) rare. More formally, the set of times at which an N-bit program can stop after the time $2^{N+\text{constant}}$ has effectively zero density.

Can we use this type of analysis for developing a probabilistic approach for proving theorems?

The class of problems which can be treated in this way are the "finitely refutable conjectures". A conjecture is finitely refutable if verifying a finite number of instances suffices to disprove it [47]. The method seems simple: we choose a natural universal Turing machine U and to each such conjecture C we can effectively associate a program Π_C such that C is true iff $U(\Pi_C)$ never halts. Running $U(\Pi_C)$ for a time longer than the threshold will produce a good evidence of the likelihood validity of the conjecture. For example, it has been shown [44] that for a natural U, the length of the program validating the Riemann Hypothesis is 7,780 bits, while for the Goldbach's Conjecture the length of the program is 3,484 bits.

Of course, the choice of the probability distribution on the runtime is paramount. Further, there are at least two types of problems with this approach.

First, the choice of the universal machine is essential. Pick a universal U and let p be a program such that $U(p)$ never stops if and only if a fixed finitely refutable conjecture (say, the Riemann Hypothesis) is true. Define W such that $W(1) = U(p)$ (tests the conjecture), and $W(0x) = U(x)$. The Turing machine W is clearly universal, but working with W "artificially" makes the threshold $\underline{\theta}$ very small. Going in the opposite direction, we can write our simulator program in such a way that it takes a huge number of steps to simulate the machine—say Ackermann's function of the runtime given by the original machine. Then the new runtime will be very long, while the program is very short. Or we could choose very powerful instructions so that even a ridiculously long program on the original machine would have a very short runtime on the new one.

The moral is that if we want to have some real idea about the probability that a conjecture has a counter-example, we should choose a simulator and program that are "honest": they should not overcharge or undercharge for each time-step advancing the computation. This phenomenon is very similar to the fact that the complexity of a single string cannot be independent of the universal machine; here, the probability of halting cannot be independent of the machine whose steps we are counting.

Second, the threshold T will increase exponentially with the length of the program Π_C (of course, the length depends upon the chosen U). For most interesting conjectures the length is greater than 100, so it is hopeless to imagine that these computations can be effectively carried out (see [148] for an analysis of the maximum speed of dynamical evolution). It is an open question whether another type of computation (possibly, quantum) can be used to speed-up the initial run of the program.

2.9 Wick rotation

Part I of this thesis has dealt with sums over programs as being analogous to sums over physical microstates in thermal systems. It is well-known that Wick rotation turns equations describing

thermal systems into equations describing quantum systems. By replacing the energy scale kT in a classical partition function by the imaginary energy $i\hbar/t$, we get a quantum partition function. Consider a large collection of harmonic oscillators at temperature T. The relative probability of finding any given oscillator with energy E is $\exp(-E/k_BT)$, where k_B is Boltzmann's constant. The average value of an observable Q is, up to a normalizing constant,

$$\sum_j Q_j e^{-E_j/(k_BT)}.$$

Now consider a single quantum harmonic oscillator in a superposition of basis states, evolving for a time t under a Hamiltonian H. The relative phase change of the basis state with energy E is $\exp(-Eit/\hbar)$, where $\hbar$ is Planck's constant. The probability amplitude that a uniform superposition of states $|\psi\rangle = \sum_j |j\rangle$ evolves to an arbitrary superposition $|Q\rangle = \sum_j Q_j|j\rangle$ is, up to a normalizing constant,

$$\begin{aligned}
&\langle Q|e^{-iHt/\hbar}|\psi\rangle \\
&= \sum_j Q_j e^{-E_j it/\hbar}\langle j|j\rangle \\
&= \sum_j Q_j e^{-E_j it/\hbar}.
\end{aligned}$$

Feynman's path integral formulation of quantum mechanics considers a sum over paths γ rather than a sum over states, each weighted by a phase $e^{-iS(\gamma)/\hbar}$, where $S(\gamma)$ is the classical action of the path. When we move from quantum mechanics to quantum field theory, the partition function sums over diagrams rather than paths. Feynman diagrams form a category: there is a trivial diagram for any set of particles where they do not interact at all, and we can compose any two diagrams where the output particles of one diagram match the input particles of the next. This category is equipped with certain structure that also appears in programming languages, linear logic, and topology.

Part II of this thesis will explore this shared structure in detail. Chapter II.1 is an expository chapter that describes the gradual discovery of this structure over the course of a century, beginning with the Curry-Howard isomorphism and extending to Feynman diagrams and topological quantum field theory. Chapter II.2 "categorifies" this structure and concludes with a proof that a particular bicategory of spans is an example.

Part II

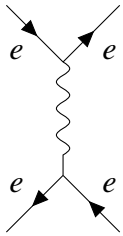
Symmetric monoidal closed categories and bicategories

Chapter 1

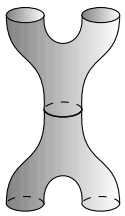
The Rosetta Stone

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 [111] 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 [160]. 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 [68, 101]. 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 [157]. With this, people began to think of quantum processes as forms of information processing and to apply ideas from computer science. It was then realized that the loose analogy 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 chapter 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 do not 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 1.1 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 1.2 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' [83]. In Section 1.3 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 1.4, 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.

1.1 The Analogy Between Physics and Topology

1.1.1 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 worldview 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 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.

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

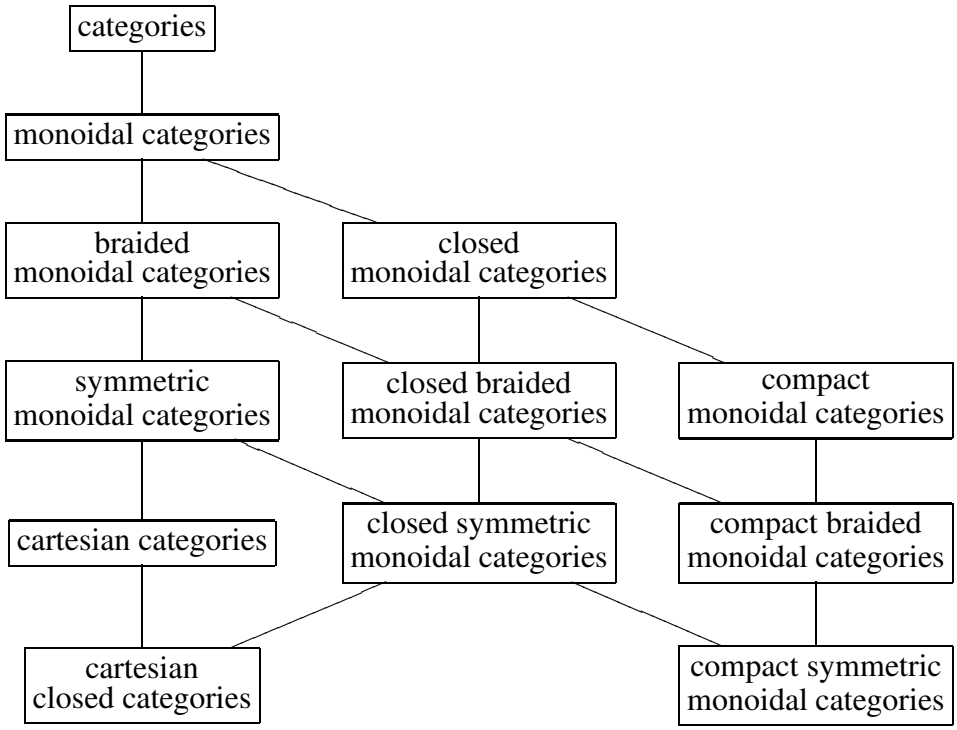
Elsewhere we have argued at great length that this is important for better understanding quantum gravity [16] and even the foundations of quantum theory [17]. The idea is that if Hilb is more like nCob than Set, maybe we should stop thinking of a quantum process as a function from one set of states to 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' [13]. Topological quantum field theory uses higher-dimensional cobordisms [18]. 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.

1.1.2 Categories

Category theory was born around 1945, with Eilenberg and Mac Lane [76] defining 'categories', 'functors' between categories, and 'natural transformations' between functors. By now there are many introductions to the subject [67, 143, 151], including some available for free online [30, 86]. Nonetheless, we begin at the beginning:

Definition 12 *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:

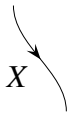
- *for every object X there is an* **identity morphism** $1_X \colon X \to X$;
- *morphisms are composable: given* $f \colon X \to Y$ *and* $g \colon Y \to Z$*, there is a* **composite morphism** $gf \colon X \to Z$*; sometimes also written* $g \circ f$*.*

- *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*

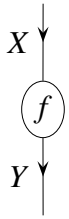
- *composition is* **associative***:* $(hg)f = h(gf)$ *whenever either side is well-defined.*

Definition 13 *We say a morphism* $f\colon X \to Y$ *is an* ***isomorphism*** *if it has an inverse— that is, there exists another morphism* $g\colon Y \to X$ *such that* $gf = 1_X$ *and* $fg = 1_Y$*.*

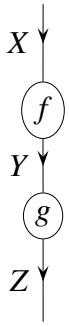
A category is the simplest framework where we can talk about systems (objects) and processes (morphisms). To visualize these, we can use 'Feynman diagrams' of a very primitive sort. In applications to linear algebra, these diagrams are often called 'spin networks', but category theorists call them 'string 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':



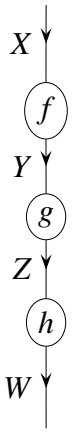
and morphisms $f\colon X \to Y$ label 'black boxes' with an input wire of type X and an output wire of type Y:



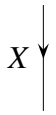
We compose two morphisms by connecting the output of one black box to the input of the next. So, the composite of $f\colon X \to Y$ and $g\colon Y \to Z$ looks like this:



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.

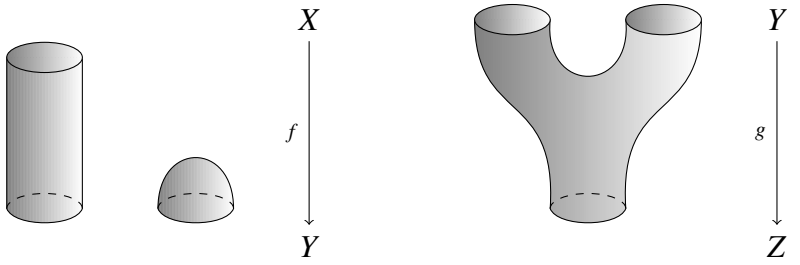
However, in quantum physics we do something subtly different: we use categories where objects are *Hilbert spaces* and morphisms are *bounded linear operators*. We specify a system by giving a Hilbert space, but this Hilbert space is not really the set of states of the system: a state is actually a ray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

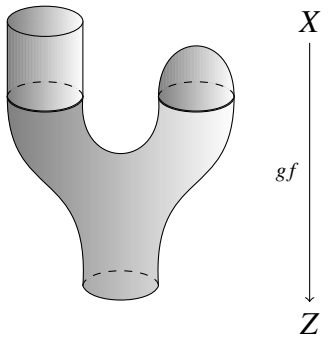
own terms. As we shall see, this sheds new light on the quandaries that have always plagued our understanding of the quantum realm [17].

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 purposes; 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 [169], 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:

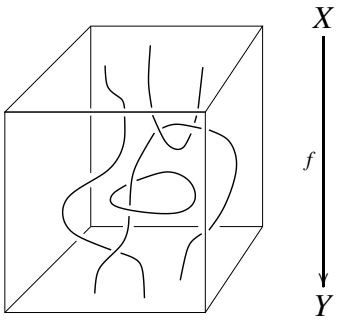


Another kind of category important in physics has objects representing *collections of particles*, and morphisms representing their *worldlines and interactions*. Feynman diagrams are the classic example, but in these diagrams the 'edges' are not taken literally as particle trajectories. An example with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

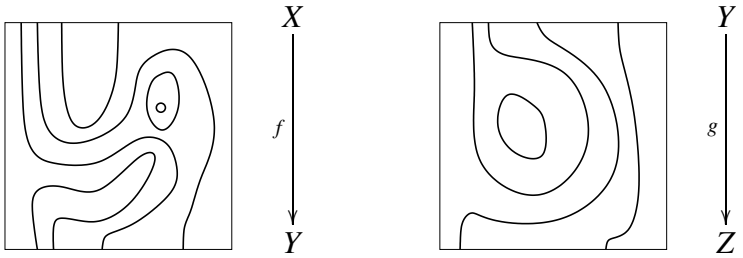
cube only at its top and bottom, and only at their endpoints. A bit more precisely, tangles are 'isotopy classes' of such embedded arcs and circles: this equivalence relation means that only the topology of the tangle matters, not its geometry. We compose tangles by attaching one cube to another top to bottom.

More precise definitions can be found in many sources, at least for $k = 2$, which gives tangles in a 3-dimensional cube [81, 114, 169, 181, 200, 205]. But since a picture is worth a thousand words, here is a picture of a morphism in Tang_2:

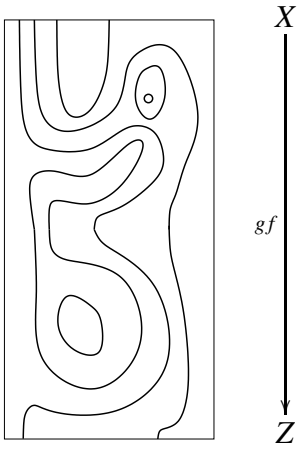


Note that we can think of a morphism in Tang_k as a 1-dimensional cobordism *embedded in a k-dimensional cube*. This is why Tang_k and nCob behave similarly in some respects.

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 [18, 181], 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 14 *A* **functor** $F\colon C \to D$ *from a category C to a category D is a 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 [136], 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' [175] and topological quantum field theories [12] can be seen as functors of this sort.

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

Definition 15 *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 16 *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.

1.1.3 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 17 *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:* $1_{(X,X')} = (1_X, 1_{X'})$.

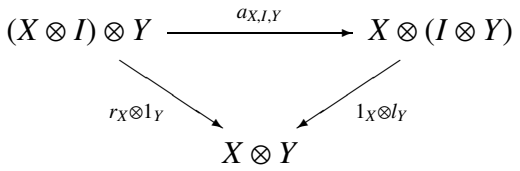
Mac Lane [144] defined monoidal categories in 1963. The subtlety of the definition lies in the fact that $(X \otimes Y) \otimes Z$ and $X \otimes (Y \otimes Z)$ are not usually equal. Instead, we should specify an isomorphism between them, called the 'associator'. Similarly, while a monoidal category 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 18 *A* **monoidal category** *consists of:*

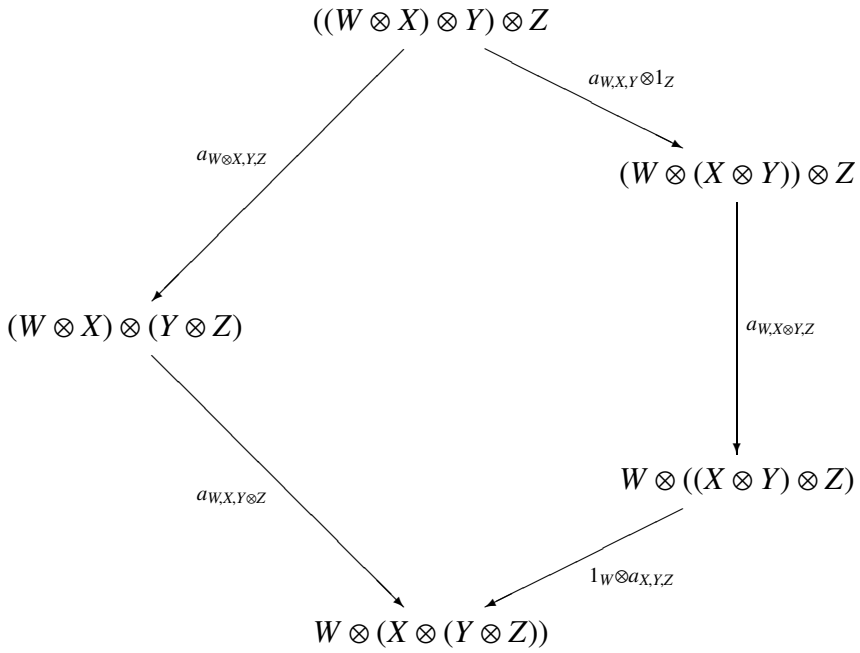
- *a category* C,
- *a* **tensor product** *functor* $\otimes \colon C \times C \to C$,
- *a* **unit object** $I \in C$,
- *a natural isomorphism called the* **associator***, assigning to each triple of objects* $X, Y, Z \in C$ *an isomorphism*
$$a_{X,Y,Z} : (X \otimes Y) \otimes Z \xrightarrow{\sim}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

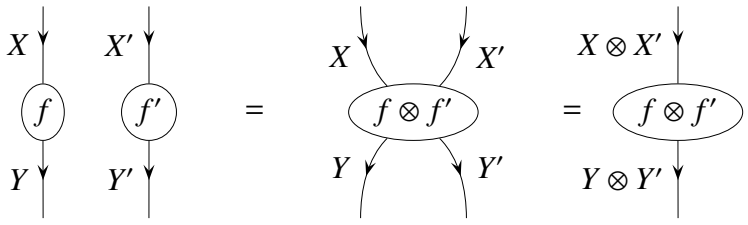


- *for all $W, X, Y, Z \in C$, the* **pentagon equation** *holds:*



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.

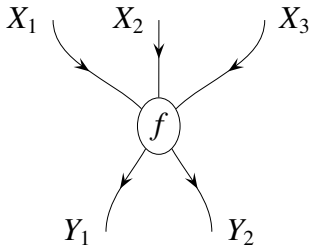
In a monoidal category we can do processes in 'parallel' as well as in 'series'. Doing processes in series is just composition of morphisms, which works in any category. But in a monoidal category we can also tensor morphisms $f\colon X \to Y$ and $f'\colon X' \to Y'$ and obtain a 'parallel process'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



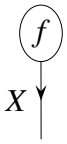
More generally, we can draw any morphism

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

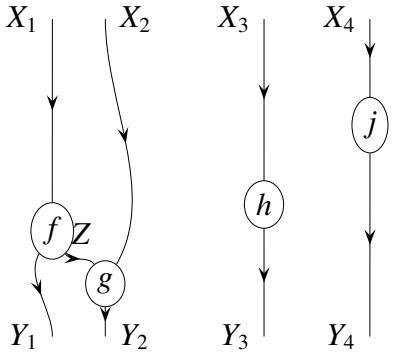
as a black box with n input wires and m output wires:



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

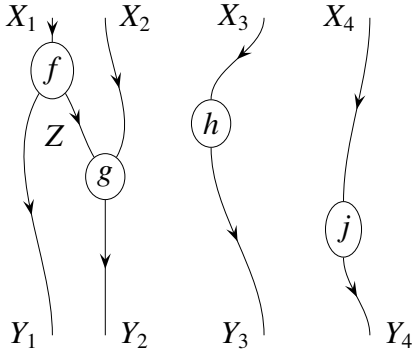


By composing and tensoring morphisms, we can build up elaborate pictures resembling Feynman diagrams:



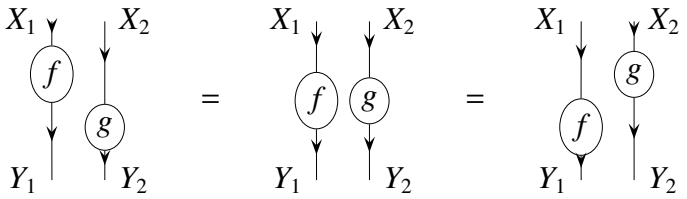
The laws governing a monoidal category allow us to neglect associators and unitors when drawing such pictures, without getting in trouble. The reason is that Mac Lane's coherence theorem says any monoidal category is 'equivalent', in a suitable sense, to one where all associators and unitors are identity morphisms [144].

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 [108] and the book by Yetter [205].

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 is 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

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

  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

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

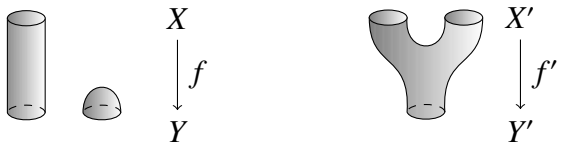
  However, in what follows, when we speak of Set *as a monoidal category, we always use the cartesian product!*

- 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 a 1-dimensional Hilbert space, for example $\mathbb{C}$.

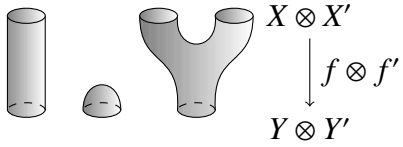
  There is also a way to make Hilb into a monoidal category where the tensor product is the direct sum: $\mathbb{C}^n \oplus \mathbb{C}^m \cong \mathbb{C}^{n+m}$. In this case the unit object is the zero-dimensional Hilbert space, $\{0\}$.

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

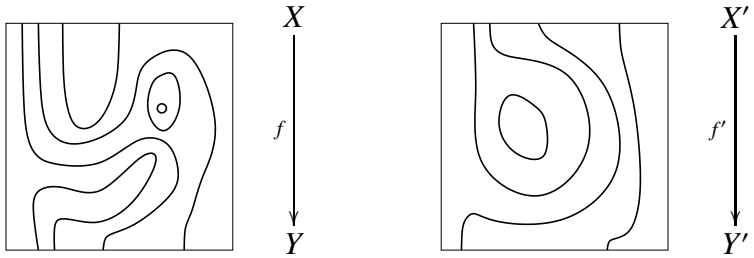
- The tensor product of objects and morphisms in nCob is given by disjoint union. For example, the tensor product of these two morphisms:



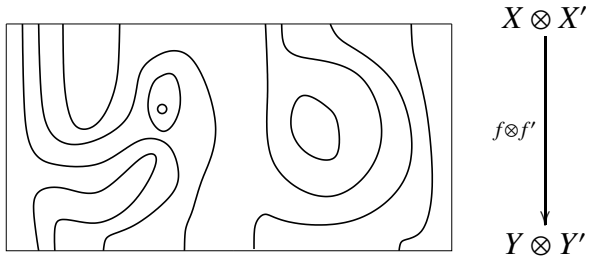
  is this:



- 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 one-element set:

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

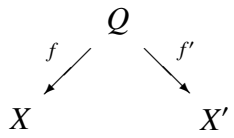
Again, our other main examples lack this feature, though Hilb made into a monoidal category using $\oplus$ has it. A fascinating feature of quantum mechanics is that we make Hilb into a monoidal category using $\otimes$ instead of $\oplus$, even though the latter approach would lead to a category more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

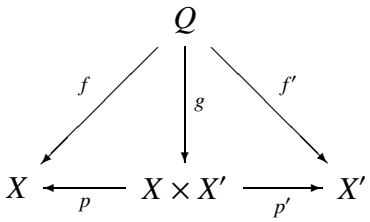
Definition 19 *Given objects X and X' in some category, we say an object $X \times X'$ equipped with morphisms*

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

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:



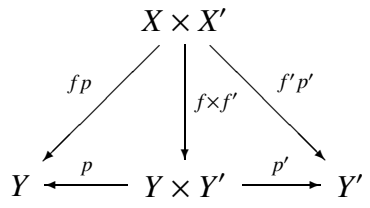
(That is, $f = pg$ and $f' = p'g$.) We say a category has **binary products** *if every pair of objects has a product.*

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 20 *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:

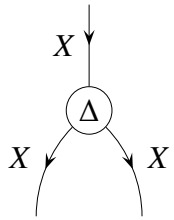
Definition 21 *A category has* **finite products** *if it has binary products and a terminal object.*

A category with finite products can always be made into a monoidal category by choosing a specific product $X \times Y$ to be 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:

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

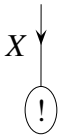
In the category Set one can check that this maps any element $x \in X$ to the pair (x, x). In general, we can draw the diagonal as follows:



Similarly, we call the unique map to the terminal object

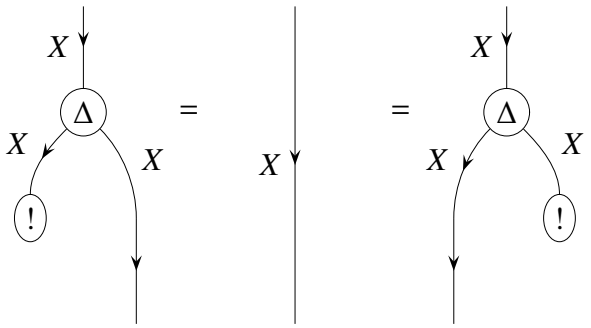
$$!_X \colon X \to 1$$

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



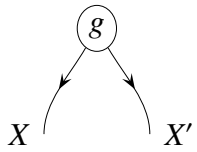
Note that we draw the unit object as an empty space.

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:

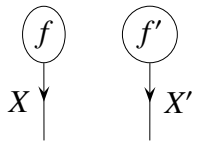


We leave the proof as an exercise for the reader.

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

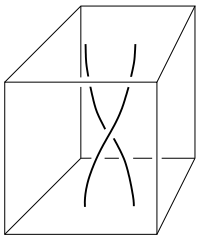


In the case of Set, this says that every point of the set $X \times X'$ comes from a point of X and a point of X'. In physics, this would say that every state g of the combined system $X \otimes X'$ is built by combining states of the systems X and X'. Bell's theorem [29] says that is *not* true in quantum 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 [204] 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.

1.1.4 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 22 *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*

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

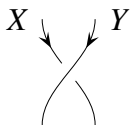
such that the **hexagon equations** *hold:*

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

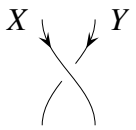
$$\begin{array}{ccccc}
(X \otimes Y) \otimes Z & \xrightarrow{a_{X,Y,Z}} & X \otimes (Y \otimes Z) & \xrightarrow{1_X \otimes b_{Y,Z}} & X \otimes (Z \otimes Y) \\
\Big\downarrow{\scriptstyle b_{X \otimes Y,Z}} & & & & \Big\downarrow{\scriptstyle a^{-1}_{X,Z,Y}} \\
Z \otimes (X \otimes Y) & \xleftarrow[a_{Z,X,Y}]{}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

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.

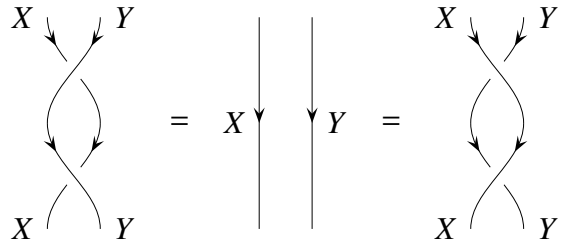
In string diagrams, we draw the braiding $b_{X,Y} \colon X \otimes Y \to Y \otimes X$ like this:



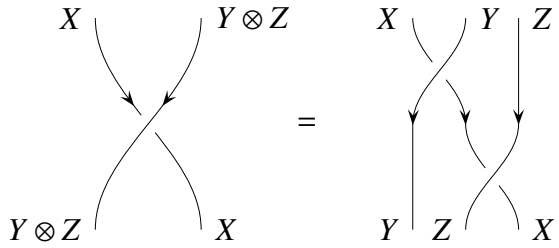
We draw its inverse $b_{X,Y}^{-1}$ like this:

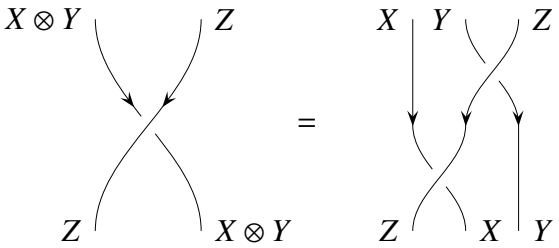


This is a nice notation, because it makes the equations saying that $b_{X,Y}$ and $b_{X,Y}^{-1}$ are inverses 'topologically true':

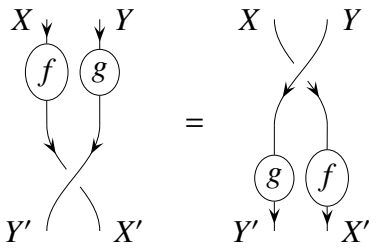


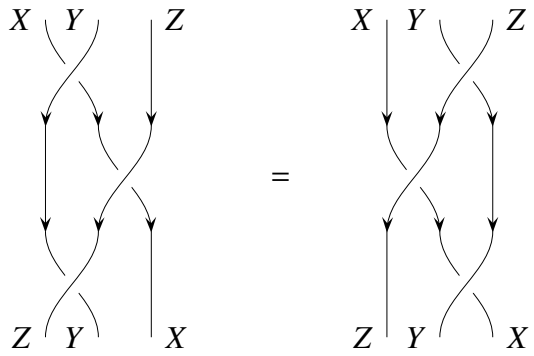
Here are the hexagon equations as string diagrams:





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 [109].

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:

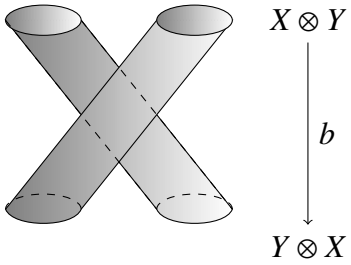
- Any cartesian category automatically becomes braided, and in Set with its cartesian product, this standard braiding is given by:

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

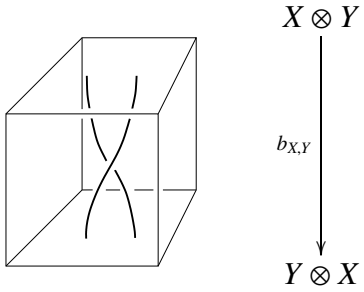
- In Hilb with its usual tensor product, the standard braiding is given by:

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

- 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:



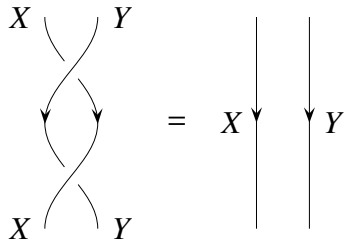
The example of Tang_k illustrates an important pattern. Tang_0 is just a category, because in 0-dimensional space we can only do processes in 'series': that is, compose morphisms. Tang_1 is a monoidal category, because in 1-dimensional space we can also do processes in 'parallel': that is, tensor morphisms. 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!

1.1.5 Symmetric Monoidal Categories

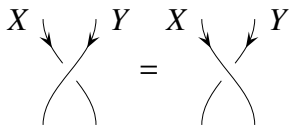
Sometimes switching two objects and switching them again is the same as doing nothing at all. Indeed, this situation is very familiar. So, the first braided monoidal categories to be discovered were 'symmetric' ones [144]:

Definition 23 *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 [18]. 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 [20], and in physics we can use them to describe not just particles but also strings and higher-dimensional membranes [18, 21]. 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. We examine the $n = 2$ column in Part III of this thesis.

	$n = \mathbf{0}$	$n = \mathbf{1}$	$n = \mathbf{2}$
$k = \mathbf{0}$	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 $j < k$.

1.1.6 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' [87]. 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 [75]. 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 X 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:

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

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 24 *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 25 *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

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

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

Definition 26 *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*

$$c_{X,Y,Z} \colon \quad \hom(X \otimes Y, Z) \quad \tilde{\to} \quad \hom(X, Y \multimap Z)$$

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

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

The term 'currying' is mainly used in computer science, after the work of Curry [68]. 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

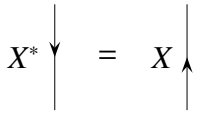
$$\begin{array}{rccc} e_X\colon & X \otimes X^* & \to & I \\ & x \otimes f & \mapsto & f(x) \end{array}$$

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:

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

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:



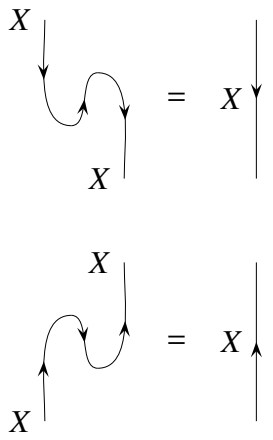
(Here we should admit that most physicists use the opposite convention, where time marches up the page. Since we read from top to bottom, we prefer to let time run down the page.)

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**:



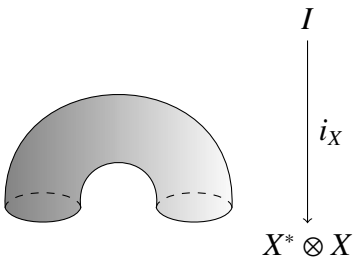
Verifying these is a fun exercise in linear algebra, which we leave to the reader. If we write these equations as commutative diagrams, we need to include some associators and unitors, and they become a bit intimidating:

$$\begin{array}{ccccc} X \otimes I & \xrightarrow{1_X \otimes i_X} & X \otimes (X^* \otimes X) & \xrightarrow{a^{-1}_{X,X^*,X}} & (X \otimes X^*) \otimes X \\ \downarrow{\scriptstyle r_X} & & & & \downarrow{\scriptstyle e_X \otimes 1_X} \\ X & & \xleftarrow[l_X]{} & & I \otimes X \end{array}$$

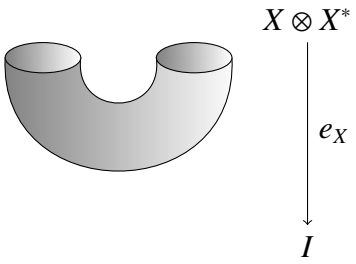
$$\begin{array}{ccccc} I \otimes X^* & \xrightarrow{i_X \otimes 1_X} & (X^* \otimes X) \otimes X^* & \xrightarrow{a_{X^*,X,X^*}} & X^* \otimes (X \otimes X^*) \\ \downarrow{\scriptstyle l_X} & & & & \downarrow{\scriptstyle 1_{X^*} \otimes e_X} \\ X^* & & \xleftarrow[r_{X^*}]{} & & X^* \otimes I \end{array}$$

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 27 *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 28 *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 [81]. To add to the confusion, compact symmetric monoidal categories are often called simply 'compact closed categories'.

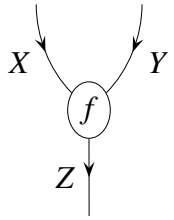
A partial explanation for the last piece of terminology is that any compact monoidal category is automatically closed! For this, we define the internal hom on objects by

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

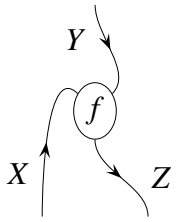
We must then show that the $*$ operation extends naturally to a functor $* \colon C \to C$, so that $\multimap$ is actually a functor. Finally, we must check that there is a natural isomorphism

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

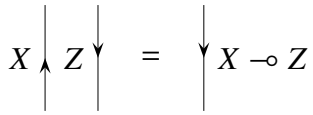
In terms of string diagrams, this isomorphism takes any morphism



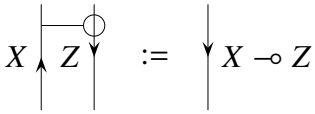
and bends back the input wire labelled X to make it an output:



Now, in a compact monoidal category, we have:

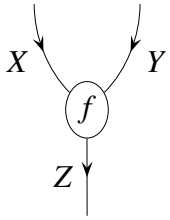


But in general, closed monoidal categories do not allow arrows pointing up! So for these, drawing the internal hom is more of a challenge. We can use the same style of notation as long as we add a decoration — a **clasp** — that binds two strings together:

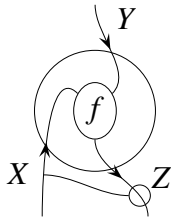


Only when our closed monoidal category happens to be compact can we eliminate the clasp.

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

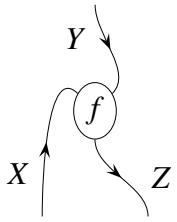


we can draw its curried version $\tilde{f}\colon Y \to X \multimap Z$ by bending down the input wire labelled X to make it part of the output:



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

Closed monoidal categories do not 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:



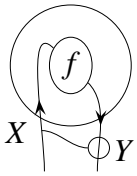
An important special case of currying gives the **name** of a morphism $f\colon X \to Y$,

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

This is obtained by currying the morphism

$$f r_x \colon I \otimes X \to Y.$$

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



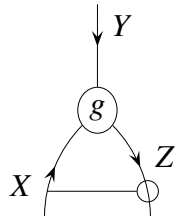
In the category Set, the unit object is the one-element set, 1. So, a morphism from this object to a set Q picks out a point of Q. In particular, the name $\ulcorner f \urcorner \colon 1 \to X \multimap Y$ picks out the element of $X \multimap Y$ corresponding to the function $f\colon X \to Y$. More generally, in any cartesian closed category the unit object is the terminal object 1, and 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 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' [87].

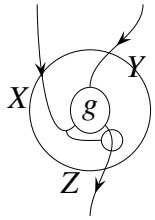
Currying is a bijection, so we can also **uncurry**:

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

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.

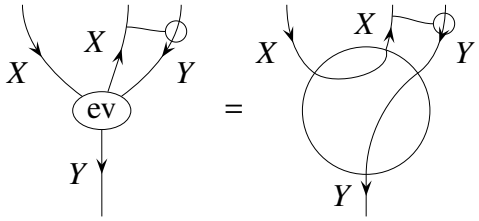
A good example of uncurrying is the **evaluation** morphism:

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

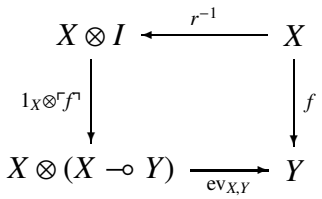
This is obtained by uncurrying the identity

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

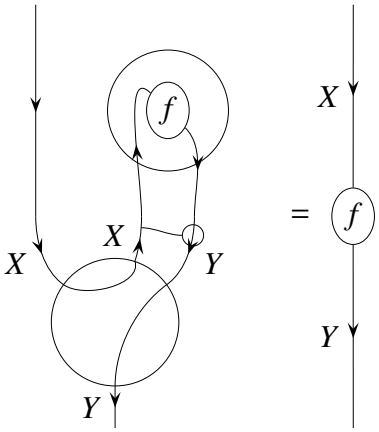
In Set, $\mathrm{ev}_{X,Y}$ takes any function from X to Y and evaluates it at any element of X to give an element of Y. In terms of string diagrams, the evaluation morphism looks like this:



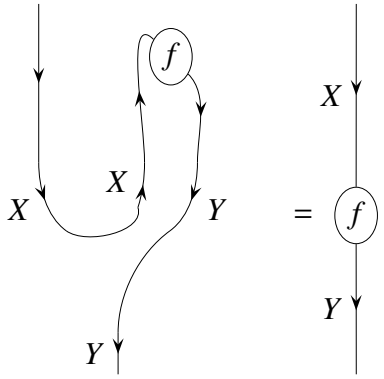
In any closed monoidal category, we can recover a morphism from its name using evaluation. More precisely, this diagram commutes:



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 [69] develops detailed string diagram descriptions of these representation categories for the classical Lie groups SU(n), SO(n), 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 a compact *braided* monoidal category of finite-dimensional representations. Kauffman's *Knots and Physics* [117] 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 [114].

- Kauffman and Lins [118] have written a beautiful string diagram treatment of the category of representations of the simplest quantum group, $\mathrm{SU}_q(2)$. 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' [182]. For generalizations to other quantum groups, see the more advanced texts by Turaev [200] and by Bakalov and Kirillov [24]. The key ingredient is a special class of compact braided monoidal categories called 'modular tensor categories'.

- Kock [125] 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, 62, 64, 65] have developed string diagrams as a tool for understanding quantum computation. The easiest introduction is Coecke's 'Kindergarten quantum mechanics' [63].

1.1.7 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.

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 [143].

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 29 *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:

- 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.
- 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'.

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, 177]. 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 [17] and also for topological quantum field theory [18], where dagger categories seem to be part of larger story involving 'n-categories with duals' [20]. However, this story is still poorly understood — there is much more work to be done.

1.2 Logic

1.2.1 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 [95] 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 [85] and Gentzen [193]. 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 $X \Rightarrow \bot$, 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 [193]. 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 [84]. 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 can be understood in terms of the duality between a category and its opposite.

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 above the line appears as part of some proposition in the sequent below 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' [83].

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 I, 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* [83]. 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 intuitionistic linear logic' [93, 170]. It turns out to be the system of logic suitable for closed symmetric monoidal categories — nothing more or less.

1.2.2 Proofs as Morphisms

In Section 1.1 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 1.2.3 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 (∘) come from the presence of identity morphisms and composition in any category. Rules (⊗), (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 ⊗ as 'and', the connective ⊸ 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:

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

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:

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

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 [83]. The word 'linear' should remind us of the category Hilb. As noted in Section 1.1.3, 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 ca not 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 ca not 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:

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

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

$$H_2 \nvdash H_2 \otimes H_2$$

and

$$H_2 \nvdash I$$

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 do not think about antimatter very often — but particle physicists do. They do not 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 an antineutrino together with a proton and an electron:

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

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

1.2.3 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 $X \nvdash Y$. 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 18):

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

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 [174]. 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 30 *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* (∘) *and* (i) *must obey equations describing associativity and the left and right unit laws. Rule* (⊗) *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* (⊗) *must also obey the triangle and pentagon equations.*

Definition 31 *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 32 *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 1.2.1.

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 [93, 170]. 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 Trimble's thesis [198] and the papers by Jay [105] and Soloviev [184]. Also see Kelly and Mac Lane's coherence theorem for closed symmetric monoidal categories [123], and the related theorem for compact symmetric monoidal categories [122].

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 [136] 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' [41]. This work initiated the categorical approach to logic which we have been sketching here. Crole's book [67] 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 [37] 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 [73] is a good place to start. For more, try the original paper by Girard [83] and the book by Troelstra [199]. Blute and Scott's review article [38] serves as a Rosetta Stone for linear logic and category theory, and so do the lectures notes by Schalk [170].
- Intuitionistic Logic (IL). Lambek and Scott's classic book [133] is still an excellent introduction to intuitionistic logic and cartesian closed categories. The online review article by Moschovakis [156] 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 25, 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 15, 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

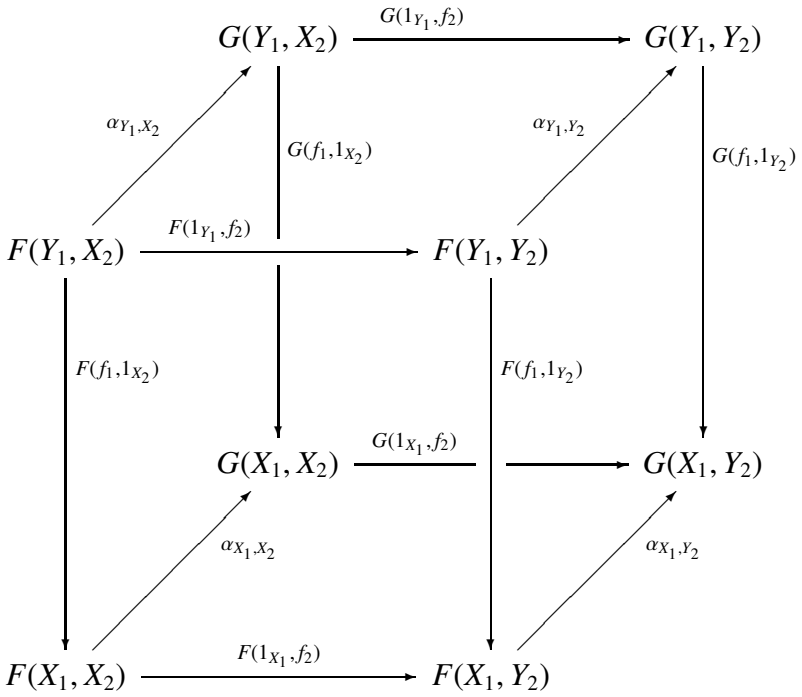
$$\mathrm{i}_X \colon 1 \;\to\; X \vdash X$$

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 [144].

As noted in Definition 15, 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, this cube commutes:



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 1.1, 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.

This motivates the following:

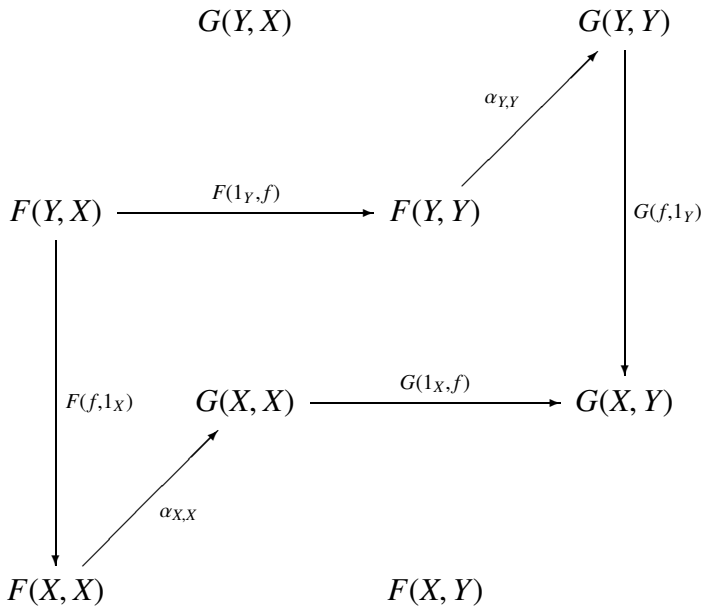


Figure 1.1: A natural transformation between functors $F, G\colon C^{\mathrm{op}} \times C \to D$ gives a commuting cube in D for any morphism $f\colon X \to Y$, and there are two paths around the cube that only involve α for repeated arguments.

Definition 33 *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 1.1 commutes.*

In the case of the identity rule, this commuting hexagon follows from the fact that the identity morphism is a left and right unit for composition: see Figure 1.2. For the cut rule, this commuting hexagon says that composition is associative: see Figure 1.3.

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.

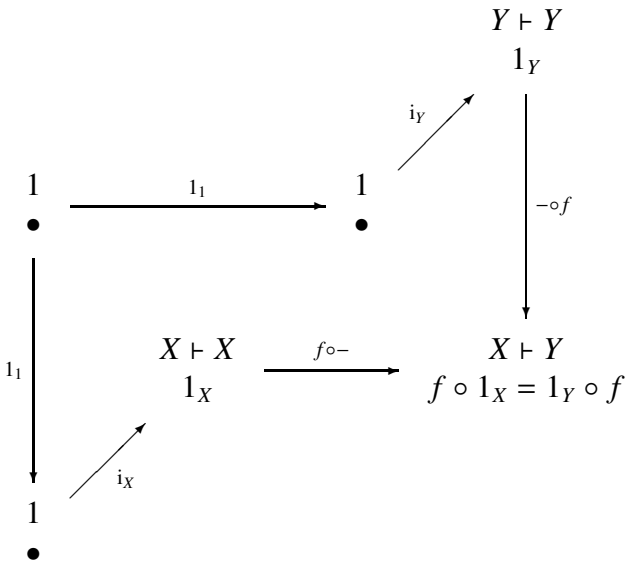


Figure 1.2: Dinaturality of the (i) rule, where $f\colon X \to Y$. Here $\bullet \in 1$ denotes the one element of the one-element set.

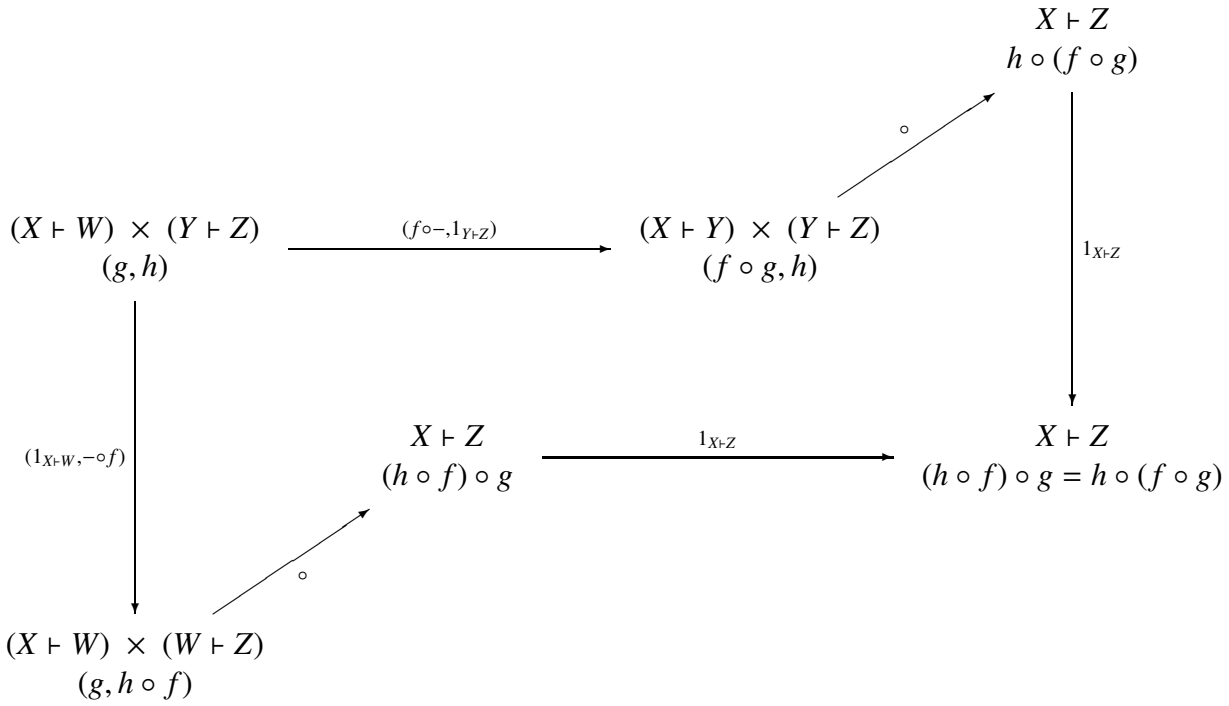


Figure 1.3: Dinaturality of the cut rule, where $f\colon W \to Y$, $g\colon X \to W$, $h\colon Y \to Z$.

1.3 Computation

1.3.1 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' [61, 124]. 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 there are many careful treatments of the lambda calculus in the literature, from Barendregt's magisterial tome [25] to the classic category-theoretic treatment of Lambek and Scott [133], to Hindley and Seldin's user-friendly introduction [97] and Selinger's elegant free online notes [176]. 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*. More prosaically, everything is a 'λ-term', or 'term' for short. These are defined inductively:

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

Let us explain the meaning of application and lambda-abstraction. Application is simple. Since 'programs are data', we can think of any term either as a program or a piece of data. Since we can apply programs to data and get new data, we can apply any term f to any other term t and get a new term $f(t)$.

Lambda-abstraction is more interesting. We think of $(\lambda x.t)$ as the program that, given x as input, returns t as output. For example, consider

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

This program takes any program x as input and returns $x(x)$ as output. In other words, it applies any program to itself. So, we have

$$(\lambda x.x(x))(s) = s(s)$$

for any term s.

More generally, if we apply $(\lambda x.t)$ to any term s, we should get back t, but with s substituted for each free occurrence of the variable x. This fact is codified in a rule called **beta reduction**:

$$(\lambda x.t)(s) = t[s/x]$$

where $t[s/x]$ is the term we get by taking t and substituting s for each free occurrence of x. 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. Starting with a term and repeatedly applying rewrite rules is how we take a program and let it run!

There are two other rewrite rules in the lambda calculus. If x is a variable and t is a term, the term

$$(\lambda x.t(x))$$

stands for the program that, given x as input, returns $t(x)$ as output. But this is just a fancy way of talking about the program t. So, the lambda calculus has a rewrite rule called **eta reduction**, saying

$$(\lambda x.t(x)) = t.$$

The third rewrite rule is **alpha conversion**. This allows us to replace a bound variable in a term by another variable. For example:

$$(\lambda x.x(x)) = (\lambda y.y(y))$$

since x is 'bound' in the left-hand expression by its appearance in 'λx'. In other words, x is just a dummy variable; its name is irrelevant, so we can replace it with y. On the other hand,

$$(\lambda x.y(x)) \neq (\lambda x.z(x)).$$

We cannot replace the variable y by the variable z here, since this variable is 'free', not bound. Some care must be taken to make the notions of free and bound variables precise, but we shall gloss over this issue, referring the reader to the references above for details.

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

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

and so on. Note that f is a variable above. Thus, the Church numeral $\overline{n}$ is the program that 'takes any program to the nth power': if you give it any program f as input, it returns the program that applies f n times to whatever input x it receives.

To get a feeling for how we can define arithmetic operations on Church numerals, consider

$$\lambda g.\overline{3}(\overline{2}(g)).$$

This program takes any program g, squares it, and then cubes the result. So, it raises g to the sixth power. This suggests that

$$\lambda g.\overline{3}(\overline{2}(g)) = \overline{6}.$$

Indeed this is true. If we treat the definitions of Church numerals as reversible rewrite rules, then we can start with the left side of the above equation and grind away using rewrite rules until we

reach the right side:

$$\begin{aligned}
(\lambda g.\overline{3}(\overline{2}(g)) &= (\lambda g.\overline{3}((\lambda f.(\lambda x.f(f(x)))))(g)) && \text{def. of } \overline{2} \\
&= (\lambda g.\overline{3}(\lambda x.g(g(x)))) && \text{beta} \\
&= (\lambda g.(\lambda f.(\lambda x.f(f(f(x)))))(\lambda x.g(g(x)))) && \text{def. of } \overline{3} \\
&= (\lambda g.(\lambda x.(\lambda x.g(g(x)))((\lambda x.g(g(x)))((\lambda x.g(g(x)))(x))))) && \text{beta} \\
&= (\lambda g.(\lambda x.(\lambda x.g(g(x)))((\lambda g.g(g(x)))(g(g(x)))))) && \text{beta} \\
&= (\lambda g.(\lambda x.(\lambda x.g(g(x)))(g(g(g(g(x))))))) && \text{beta} \\
&= (\lambda g.(\lambda x.g(g(g(g(g(g(g(x)))))))) && \text{beta} \\
&= \overline{6} && \text{def. of } \overline{6}
\end{aligned}$$

If this calculation seems mind-numbing, that is precisely the point: it resembles the inner workings of a computer. We see here how the lambda calculus can serve as a programming language, with each step of computation corresponding to a rewrite rule.

Of course, we got the answer $\overline{6}$ because $3 \times 2 = 6$. Generalizing from this example, we can define a program called 'times' that multiplies Church numerals:

$$\text{times} = (\lambda a.(\lambda b.(\lambda x.a(b(x))))).$$

For example,

$$\text{times}(\overline{3})(\overline{2}) = \overline{6}.$$

The enterprising reader can dream up similar programs for the other basic operations of arithmetic. 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'.

After this burst of theoretical work, it took a few decades for programmable computers to actually be built. It took even longer for computer scientists to profit from Church and Kleene's insights. This began around 1958, when McCarthy invented the programming language Lisp, based on the lambda calculus [149]. In 1965, an influential paper by Landin [134] pointed out a powerful analogy between the lambda calculus and the language ALGOL. These developments led to a renewed interest in 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!

We should admit that 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 is a series of instructions that tell the computer what to do. By constrast, in functional programming, a program simply describes a function. To run the program, we apply it to an input. So, as in the lambda calculus, 'application' is a fundamental operation in functional programming. If we combine application with lambda abstraction, we obtain a language powerful enough to compute any computable function.

However, most functional programming languages are more regimented than the original lambda calculus. As we have seen, in the lambda calculus as originally developed by Church and Kleene, any term can be applied to any other. In real life, programming involves many kinds of data. For example, suppose we are writing a program that involves days of the week. It would not make sense to write

$$\text{times}(\overline{3})(\text{Tuesday})$$

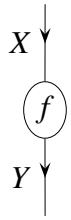
because Tuesday is not a number. We might choose to represent Tuesday by a number in some program, but doubling that number does not have a good interpretation: is the first day of the week Sunday or Monday? Is the week indexed from zero or one? These are arbitrary choices that affect the result. We could let the programmer make the choices, but the resulting 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. Thus, whenever we introduce a variable in a program, we should make a 'type declaration' saying what type it is. For example, we might write:

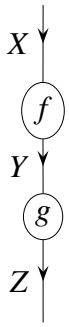
$$\text{Tuesday} : \text{day}$$

This notation is used in Ada, Pascal and some other languages. Other notations are also in widespread use. Then, our system should have a 'type checker' (usually part of the compiler) that complains if we try to apply a program to a piece of data of the wrong type.

Mathematically, this idea is formalized by a more sophisticated version of the lambda calculus: the 'typed' lambda calculus, where every term has a type. This idea is also fundamental to 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:



Indeed, there is a deep relation between the typed lambda calculus and cartesian closed categories. This was discovered by Lambek in 1980 [132]. Quite roughly speaking, a 'typed lambda-theory' is a very simple functional programming language with a specified collection of basic data types from which other more complicated types can be built, and a specified collection of basic terms from which more complicated terms can be built. The data types of this language are *objects* in a cartesian closed category, while the programs — that is, terms — give *morphisms!*

Here we are being a bit sloppy. Recall from Section 1.2.3 that in logic we can build closed monoidal categories where the morphisms are equivalence classes of proofs. We need to take equivalence classes for the axioms of a closed monoidal category to hold. Similarly, to get closed monoidal categories from computer science, we need the morphisms to be equivalence classes of terms. Two terms count as equivalent if they differ by rewrite rules such as beta reduction, eta reduction and alpha conversion. As we have seen, these rewrites represent the steps whereby a program carries out its computation. For example, in the original 'untyped' lambda calculus, the terms times$(\overline{3})(\overline{2})$ and $\overline{6}$ differ by rewrite rules, but they give the same morphism. So, when we construct a cartesian closed category from a typed lambda-theory, we *neglect the actual process of computation*. To remedy this we should work with a cartesian closed 2-category which has:

- types as objects,
- terms as morphisms,
- equivalence classes of rewrites as 2-morphisms.

For details, see the work of Seely [174], Hilken [96], and Melliés [152]. Someday this work will be part of the larger n-categorical Rosetta Stone mentioned at the end of Section 1.1.5.

In any event, Lambek showed that every typed lambda-theory gives a cartesian closed category — and conversely, every cartesian closed category gives a typed lambda-theory. This discovery led to a rich 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 [67] and the online review article by Scott [173].

In what follows, our goal is more limited. First, in Section 1.3.2, we explain how every 'typed lambda-theory' gives a cartesian closed category, and conversely. We follow the treatment of Lambek and Scott [133], in a somewhat simplified form. Then, in Section 1.3.3, we describe how every 'linear type theory' gives a closed symmetric monoidal category, and conversely.

The idea here is roughly that a 'linear type theory' is a programming language suitable for *both classical and quantum computation*. This language differs from the typed lambda calculus in that it 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' and 'deletion' morphisms:

$$\Delta_X \colon X \to X \otimes X, \qquad !_X \colon X \to 1,$$

a symmetric monoidal category typically lacks these. As we saw in Section 1.1.3, a great example is the category Hilb with its usual tensor product. So, a programming language suitable for quantum computation should not assume we can duplicate all types of data [157, 204].

Various versions of 'quantum' or 'linear' lambda calculus have already been studied, for example by Benton, Bierman de Paiva and Hyland [35], Dorca and van Tonder [201], and Selinger and Valiron [178]. Abramsky and Tzevelekos sketch a version in their paper in this volume [6]. We instead explain the 'linear type theories' developed by Simon Ambler in his 1991 thesis [8].

1.3.2 The Typed Lambda Calculus

Like the original 'untyped' lambda calculus explained above, the typed lambda calculus uses terms to represent both programs and data. However, now every term has a specific type. A program that inputs data of type X and outputs data of type Y is said to be of type $X \multimap Y$. So, we can only apply a term s to a term t of type X if s is of type $X \multimap Y$ for some Y. In this case $s(t)$ is a well-defined term of type Y. We call $X \multimap Y$ a **function type**.

Whenever we introduce a variable, we must declare its type. We write $t : X$ to mean that t is a term of type X. So, in lambda abstraction, we no longer simply write expressions like $(\lambda x \,.\, t)$. Instead, if x is a variable of type X, we write

$$(\lambda x : X \,.\, t).$$

For example, here is a simple program that takes a program of type $X \multimap X$ and 'squares' it:

$$(\lambda f : X \multimap X \,.\, (\lambda x : X \,.\, f(f(x)))).$$

In the original lambda calculus, all programs take a single piece of data as input. In other words, they compute unary functions. This is no real limitation, since we can handle functions that take more than one argument using a trick called 'currying', discussed in Section 1.1.6 This turns a function of several arguments into a function that takes the first argument and returns a function of the remaining arguments. We saw an example in the last section: the program 'times'. For example, $\mathrm{times}(\overline{3})$ is a program that multiplies by 3, so $\mathrm{times}(\overline{3})(\overline{2}) = \overline{6}$.

While making all programs compute unary functions is economical, it is not very kind to the programmer. So, in the typed lambda calculus we also introduce products: given types X and Y, there is a type $X \times Y$ called a **product type**. We can think of a datum of type $X \times Y$ as a pair consisting of a datum of type X and a datum of type Y. To make this intuition explicit, we insist that given terms $s : X$ and $t : Y$ there is a term $(s, t) : X \times Y$. We also insist that given a term $u : X \times Y$

there are terms $p(u) : X$ and $p'(u) : Y$, which we think of as the first and second components of the pair t. We also include rewrite rules saying:

$$\begin{array}{rcll} (p(u), p'(u)) & = & u & \text{for all } u : X \times Y, \\ p(s,t) & = & s & \text{for all } s : X \text{ and } t : Y, \\ p'(s,t) & = & t & \text{for all } s : X \text{ and } t : Y. \end{array}$$

Product types allow us to write programs that take more than one input. Even more importantly, they let us deal with programs that produce more than one output. For example, we might have a type called 'integer'. Then we might want a program that takes an integer and duplicates it:

$$\text{duplicate} : \text{integer} \multimap (\text{integer} \times \text{integer})$$

Such a program is easy to write:

$$\text{duplicate} = (\lambda x : \text{integer} \,.\, (x, x)).$$

Of course this a program we should *not* be allowed to write when duplicating information is forbidden, but in this section our considerations are all 'classical', i.e., suited to cartesian closed categories.

The typed lambda calculus also has a special type called the 'unit type', which we denote as 1. There is a single term of this type, which we denote as (). From the viewpoint of category theory, the need for this type is clear: a category with finite products must have not only binary products but also a terminal object (see Definition 21). For example, in the category Set, the terminal object can be taken as any one-element set, and () is the unique element of this set. It may be less clear why this type is useful in programming. One reason is that it lets us think of a constant of type X as a function of type $1 \multimap X$ — that is, a 'nullary' function, one that takes no arguments. There are some other reasons, but they go beyond the scope of this discussion. Suffice it to say that Haskell, Lisp and even widely used imperative languages such as C, C++ and Java include the unit type.

Having introduced the main ingredients of the typed lambda calculus, let us give a more formal treatment. As we shall see, a 'typed lambda-theory' consists of types, terms and rewrite rules. From a typed lambda-theory we can get a cartesian closed category. The types will give objects, the terms will give morphisms, and the rewrite rules will give equations between morphisms.

First, the **types** are given inductively as follows:

- **Basic types:** There is an arbitarily chosen set of types called **basic types**.
- **Product types:** Given types X and Y, there is a type $X \times Y$.
- **Function types:** Given types X and Y, there is a type $X \multimap Y$.
- **Unit type:** There is a type 1.

Next, we may quotient out by some notion of type equality; for example we may have a type X satisfying $X \times X = X$. However, we demand that:

- If $X = X'$ and $Y = Y'$ then $X \times Y = X' \times Y'$.
- If $X = X'$ and $Y = Y'$ then $X \multimap Y = X' \multimap Y'$.

Next we define **terms**. Each term has a specific type, and if t is a term of type X we write $t : X$. The rules for building terms are as follows:

- **Basic terms:** For each type X there is a set of **basic terms** of type X.
- **Variables:** For each type X there is a countably infinite collection of terms of type X called **variables** of type X.
- **Application:** If $f : X \multimap Y$ and $t : X$ then there is a term $f(t)$ of type Y.
- **Lambda abstraction:** If x is a variable of type X and $t : Y$ then there is a term $(\lambda x : X \,.\, t)$ of type $X \multimap Y$.
- **Pairing:** If $s : X$ and $t : Y$ then there is a term (s, t) of type $X \times Y$.
- **Projection:** If $t : X \times X'$ then there is a term $p(t)$ of X and a term $p'(t)$ of type X'.
- **Unit term:** There is a term $()$ of type 1.

Finally there are **rewrite rules** going between terms of the same type. Given any fixed set of variables $\mathcal{S}$, there will be rewrite rules between terms of the same type, all of whose free variables lie in the set $\mathcal{S}$. For our present purposes, we only need these rewrite rules to decide when two terms determine the same morphism in the cartesian closed category we shall build. So, what matters is not really the rewrite rules themselves, but the equivalence relation they generate. We write this equivalence relation as $s \sim_{\mathcal{S}} t$.

The relation $\sim_{\mathcal{S}}$ can be any equivalence relation satisfying the following list of rules. In what follows, $t[s/x]$ denotes the result of taking a term t and replacing every free occurence of the variable x by the term s. Also, when when we say 'term' without further qualification, we mean 'term all of whose free variables lie in the set $\mathcal{S}$'.

- **Type preservation:** If $t \sim_{\mathcal{S}} t'$ then t and t' must be terms of the same type, all of whose free variables lie in the set $\mathcal{S}$.
- **Beta reduction:** Suppose x is a variable of type X, s is a term of type X, and t is any term. If no free occurrence of a variable in s becomes bound in $t[s/x]$, then:

$$(\lambda x : X \,.\, t)(s) \;\sim_{\mathcal{S}}\; t[s/x].$$

- **Eta reduction:** Suppose the variable x does not appear in the term f. Then:

$$(\lambda x : X \,.\, f(x)) \sim_{\mathcal{S}} f.$$

- **Alpha conversion:** Suppose x and y are variables of type X, and no free occurrence of any variable in t becomes bound in $t[x/y]$. Then:

$$(\lambda x{:}X\ .\ t)\ \sim_{\mathcal{S}}\ (\lambda y{:}X\ .\ t[x/y]).$$

- **Application:** Suppose t and t' are terms of type X with $t \sim_{\mathcal{S}} t'$, and suppose that $f : X \multimap Y$. Then:

$$f(t)\ \sim_{\mathcal{S}}\ f(t').$$

- **Lambda abstraction:** Suppose t and t' are terms of type Y, all of whose free variables lie in the set $\mathcal{S} \cup \{x\}$. Suppose that $t \sim_{\mathcal{S}\cup\{x\}} t'$. Then:

$$(\lambda x{:}X\ .\ t)\ \sim_{\mathcal{S}}\ (\lambda x{:}X\ .\ t')$$

- **Pairing:** If u is a term of type $X \times Y$ then:

$$(p(u), p'(u))\ \sim_{\mathcal{S}}\ u.$$

- **Projection:** if s is a term of type X and t is a term of type Y then:

$$\begin{aligned} p(s,t)\ &\sim_{\mathcal{S}}\ s \\ p'(s,t)\ &\sim_{\mathcal{S}}\ t. \end{aligned}$$

- **Unit term:** If t is a term of type 1 then:

$$t\ \sim_{\mathcal{S}}\ ().$$

Now we can describe Lambek's classic result relating typed lambda-theories to cartesian closed categories. From a typed lambda-theory we get a cartesian closed category C for which:

- The objects of C are the types.
- The morphisms $f\colon X \to Y$ of C are equivalence classes of pairs (x, t) consisting of a variable $x : X$ and a term $t : Y$ with no free variables except perhaps x. Here (x, t) is equivalent to (x', t') if and only if:

$$t\ \sim_{\{x\}}\ t'[x/x'].$$

- Given a morphism $f\colon X \to Y$ coming from a pair (x, t) and a morphism $g\colon Y \to Z$ coming from a pair (y, u) as above, the composite $gf\colon X \to Y$ comes from the pair $(x, u[t/y])$.

We can also reverse this process and get a typed lambda-theory from a cartesian closed category. In fact, Lambek and Scott nicely explain how to construct a category of cartesian closed categories and a category of typed-lambda theories. They construct functors going back and forth between these categories and show these functors are inverses up to natural isomorphism. We thus say these categories are 'equivalent' [133].

1.3.3 Linear Type Theories

In his thesis [8], Ambler described how to generalize Lambek's classic result from cartesian closed categories to closed symmetric monoidal categories. To do this, he replaced typed lambda-theories with 'linear type theories'. A linear type theory can be seen as a programming language suitable for both classical and quantum computation. As we have seen, in a noncartesian category like Hilb, we cannot freely duplicate or delete information. So linear type theories must prevent duplication or deletion of data *except when it is expressly allowed*.

To achieve this, linear type theories must not allow us to write a program like this:

$$(\lambda x{:}X \,.\, (x, x)).$$

Even a program that 'squares' another program, like this:

$$(\lambda f{:}X \multimap X \,.\, (\lambda x : X \,.\, f(f(x)))),$$

is not allowed, since it 'reuses' the variable f. On the other hand, a program that composes two programs is allowed!

To impose these restrictions, linear type theories treat variables very differently than the typed lambda calculus. In fact, in a linear type theory, any term will contain a given variable at most *once*. But linear type theories depart even more dramatically from the typed lambda calculus in another way. They make no use of lambda abstraction! Instead, they use 'combinators'.

The idea of a combinator is very old: in fact, it predates the lambda calculus. Combinatory logic was born in a 1924 paper by Schönfinkel [172], and was rediscovered and extensively developed by Curry [68] starting in 1927. In retrospect, we can see their work as a stripped-down version of the untyped 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 f to any term t and get a term $f(t)$.

To build a Turing-complete programming language in such an impoverished setup, we need a sufficient stock of combinators. Remarkably, it suffices to use three. In fact it is possible to use just *one* cleverly chosen combinator — but this tour de force is not particularly enlightening, so we shall describe a commonly used set of three. The first, called I, acts like the identity, since it comes with the rewrite rule:

$$I(a) = a$$

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

$$K(a)(b) = a$$

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:

$$S\,(a)(b)(c) = a(c)(b(c)).$$

Later it was seen that the combinator calculus can be embedded in the untyped lambda calculus as follows:

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

The rewrite rules for these combinators then follow from rewrite rules in the lambda calculus. More surprisingly, any function computable using the lambda calculus can also be computed using just I, K and S ! While we do not need this fact to understand linear type theories, we cannot resist sketching the proof, since it is a classic example of using combinators to avoid explicit use of lambda abstraction.

Note that all the variables in the lambda calculus formulas for I, K, and S are bound variables. More generally, in the lambda calculus we define a **combinator** to be a term in which all variables are bound 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 $c(t)$ and $d(t)$ in a way that leads to the same term. There is a process called 'abstraction elimination' that takes any combinator in the lambda calculus and produces an extensionally equivalent one built from I, K, and S.

Abstraction elimination works by taking a term $t = (\lambda x.u)$ with a single lambda abstraction and rewriting it into the form $(\lambda x.f(x))$, where f has no instances of lambda abstraction. Then we can apply eta reduction, which says $(\lambda x.f(x)) = f$. This lets us rewrite t as a term f that does not involve lambda abstraction. We shall use the notation $[[u]]_x$ to mean 'any term f satisfying $f(x) = u$'.

There are three cases to consider; each case justifies the definition of one combinator:

1. $t = (\lambda x.x)$. We can rewrite this as $t = (\lambda x.I(x))$, so $t = [[x]]_x = I$.
2. $t = (\lambda x.u)$, where u does not depend on x. We can rewrite this as $t = (\lambda x.K(u)(x))$, so $t = [[u]]_x = K(u)$.
3. $t = (\lambda x.u(v))$, where u and v may depend on x. We can rewrite this as $t = (\lambda x.(([[u]]_x x)([[v]]_x x))$ or $t = (\lambda x.S([[u]]_x)([[v]]_x)(x))$, so $t = S([[u]]_x)([[v]]_x)$.

We can eliminate all use of lambda abstraction from any term by repeatedly using these three rules 'from the inside out'. To see how this works, consider the lambda term $t = (\lambda x.(\lambda y.y))$, which takes two inputs and returns the second. Using the rules above we have:

$$\begin{aligned} (\lambda x.(\lambda y.y)) &= (\lambda x.(\lambda y.[[y]]_y(y))) \\ &= (\lambda x.(\lambda y.I(y))) \\ &= (\lambda x.I) \\ &= (\lambda x.[[I]]_x(x)) \\ &= (\lambda x.K(I)(x) \\ &= K(I). \end{aligned}$$

We can check that it works as desired: $K(I)(x)(y) = I(y) = y$.

Now let us return to our main theme: linear type theories. Of the three combinators described above, only I is suitable for use in an arbitrary closed symmetric monoidal category. The reason is that K deletes data, while S duplicates it. We can see this directly from the rewrite rules they satisfy:

$$\begin{aligned} K(a)(b) &= a \\ S(a)(b)(c) &= a(c)(b(c)). \end{aligned}$$

Every linear type theory has a set of 'basic combinators', which neither duplicate nor delete data. Since linear type theories generalize *typed* lambda-theories, these basic combinators are typed. Ambler writes them using notation resembling the notation for morphisms in category theory.

For example, given two types X and Y in a linear type theory, there is a **tensor product type** $X \otimes Y$. This is analogous to a product type in the typed lambda calculus. In particular, given a term s of type X and a term t of type Y, we can combine them to form a term of type $X \otimes Y$, which we now denote as $(s \otimes t)$. We reparenthesize iterated tensor products using the following basic combinator:

$$\mathrm{assoc}_{X,Y,Z} \colon (X \otimes Y) \otimes Z \to X \otimes (Y \otimes Z).$$

This combinator comes with the following rewrite rule:

$$\mathrm{assoc}_{X,Y,Z}((s \otimes t) \otimes u) = (s \otimes (t \otimes u))$$

for all terms $s : X$, $t : Y$ and $u : Z$.

Of course, the basic combinator $\mathrm{assoc}_{X,Y,Z}$ is just a mildly disguised version of the associator, familiar from category theory. Indeed, all the basic combinators come from natural or dinatural transformations implicit in the definition of 'closed symmetric monoidal category'. In addition to these, any given linear type theory also has combinators called 'function symbols'. These come from the morphisms particular to a given category. For example, suppose in some category the tensor product $X \otimes X$ is actually the cartesian product. Then the corresponding linear type theory should have a function symbol

$$\Delta_X \colon X \to X \otimes X$$

which lets us duplicate data of type X, together with function symbols

$$p \colon X \otimes X \to X, \qquad p' \colon X \otimes X \to X$$

that project onto the first and second factors. To make sure these work as desired, we can include rewrite rules:

$$\begin{aligned} \Delta(s) &= (s \otimes s) \\ p(s \otimes t) &= s \\ p'(s \otimes t) &= t. \end{aligned}$$

So, while duplication and deletion of data is not a 'built-in feature' of linear type theories, we can include it when desired.

Using combinators, we could try to design a programming language suitable for closed symmetric monoidal categories that completely avoid the use of variables. Ambler follows a different path. He retains variables in his formalism, but they play a very different — and much *simpler* —- role

than they do in the lambda calculus. Their only role is to help decide which terms should count as equivalent. Furthermore, lambda abstraction plays no role in linear type theories, so the whole issue of free versus bound variables does not arise! In a sense, all variables are free. Moreover, every term contains any given variable at most once.

After these words of warning, we hope the reader is ready for a more formal treatment of linear type theories. A **linear type theory** has types, combinators, terms, and rewrite rules. The types will correspond to objects in a closed symmetric monoidal category, while equivalence classes of combinators will correspond to morphisms. Terms and rewrite rules are only used to define the equivalence relation.

First, the set of **types** is defined inductively as follows:

- **Basic types:** There is an arbitarily chosen set of types called **basic types**.
- **Product types:** Given types X and Y, there is a type $(X \otimes Y)$.
- **Function types:** Given types X and Y, there is a type $(X \multimap Y)$.
- **Trivial type:** There is a type I.

Next, we quotient out by some (perhaps empty) set of equations. We require that:

- If $X = X'$ and $Y = Y'$ then $X \otimes Y = X' \otimes Y'$.
- If $X = X'$ and $Y = Y'$ then $X \multimap Y = X' \multimap Y'$.

Second, a linear type theory has for each pair of types X and Y a set of **combinators** of the form $f \colon X \to Y$. These are defined by the following inductive rules:

- Given types X and Y there is an arbitrarily chosen set of combinators $f \colon X \to Y$ called **function symbols**.
- Given types X, Y, and Z we have the following combinators, called **basic combinators**:
  - $\mathrm{id}_X \colon X \to X$
  - $\mathrm{assoc}_{X,Y,Z} \colon (X \otimes Y) \otimes Z \to X \otimes (Y \otimes Z)$
  - $\mathrm{unassoc}_{X,Y,Z} \colon X \otimes (Y \otimes Z) \to (X \otimes Y) \otimes Z$
  - $\mathrm{braid}_{X,Y} \colon X \otimes Y \to Y \otimes X$
  - $\mathrm{left}_X \colon I \otimes X \to X$
  - $\mathrm{unleft}_X \colon X \to I \otimes X$
  - $\mathrm{right}_X \colon I \otimes X \to X$
  - $\mathrm{unright}_X \colon X \to I \otimes X$
  - $\mathrm{eval}_{X,Y} \colon X \otimes (X \multimap Y) \to Y$

- If $f\colon X \to Y$ and $g\colon Y \to Z$ are combinators, then $(g \circ f)\colon X \to Z$ is a combinator.
- If $f\colon X \to Y$ and $g\colon X' \to Y'$ are combinators, then $(f \otimes g)\colon X \otimes X' \to Y \otimes Y'$ is a combinator.
- If $f\colon X \otimes Y \to Z$ is a combinator, then we can **curry** f to obtain a combinator $\tilde{f}\colon Y \to (X \multimap Z)$.

It will generally cause no confusion if we leave out the subscripts on the basic combinators. For example, we may write simply 'assoc' instead of $\mathrm{assoc}_{X,Y,Z}$.

Third, a linear type theory has a set of **terms** of any given type. As usual, we write $t : X$ to say that t is a term of type X. Terms are defined inductively as follows:

- For each type X there is a countably infinite collection of **variables** of type X. If x is a variable of type X then $x : X$.
- There is a term 1 with $1 : I$.
- If $s : X$ and $t : Y$, then there is a term $(s \otimes t)$ with $(s \otimes t) : X \otimes Y$, as long as no variable appears in both s and t.
- If $f\colon X \to Y$ is a combinator and $t : X$ then there is a term $f(t)$ with $f(t) : X$.

Note that any given variable may appear at most once in a term.

Fourth and finally, a linear type theory has **rewrite rules** going between terms of the same type. As in our treatment of the typed lambda calculus, we only care here about the equivalence relation $\sim$ generated by these rewrite rules. This equivalence relation must have all the properties listed below. In what follows, we say a term is **basic** if it contains no combinators. Such a term is just an iterated tensor product of distinct variables, such as

$$(z \otimes ((x \otimes y) \otimes w)).$$

These are the properties that the equivalence relation $\sim$ must have:

- If $t \sim t'$ then t and t' must be terms of the same type, containing the same variables.
- The equivalence relation is **substitutive**:
  - Given terms $s \sim s'$, a variable x of type X, and terms $t \sim t'$ of type X whose variables appear in neither s nor s', then $s[t/x] \sim s'[t'/x]$.
  - Given a basic term t with the same type as a variable x, if none of the variables of t appear in the terms s or s', and $s[t/x] \sim s'[t/x]$, then $s \sim s'$.
- The equivalence relation is **extensional**: if $f : X \multimap Y$, $g : X \multimap Y$ and $\mathrm{eval}(t \otimes f) \sim \mathrm{eval}(t \otimes g)$ for all basic terms $t : X$, then $f \sim g$.

- We have:
  - $\mathrm{id}(s) \sim s$
  - $(g \circ f)(s) \sim g(f(s))$
  - $(f \otimes g)(s \otimes t) \sim (f(s) \otimes g(t))$
  - $\mathrm{assoc}((s \otimes t) \otimes u) \sim (s \otimes (t \otimes u))$
  - $\mathrm{unassoc}(s \otimes (t \otimes u)) \sim ((s \otimes t) \otimes u)$
  - $\mathrm{braid}(s \otimes t) \sim (t \otimes s)$
  - $\mathrm{left}(1 \otimes s) \sim s$
  - $\mathrm{unleft}(s) \sim (1 \otimes s)$
  - $\mathrm{right}(1 \otimes s) \sim s$
  - $\mathrm{unright}(s) \sim (1 \otimes s)$
  - $\mathrm{eval}(s \otimes \tilde{f}(t)) \sim f(s \otimes t)$

Note that terms can have variables appearing anywhere within them. For example, if x, y, z are variables of types X, Y and Z, and $f\colon Y \otimes Z \to W$ is a function symbol, then

$$\mathrm{braid}(x \otimes f(y \otimes z))$$

is a term of type $W \otimes X$. However, every term t is equivalent to a term of the form $\mathrm{cp}(t)(\mathrm{vp}(t))$, where $\mathrm{cp}(t)$ is the **combinator part** of t and $\mathrm{vp}(t)$ is a basic term called the **variable part** of t. For example, the above term is equivalent to

$$\mathrm{braid} \circ (\mathrm{id} \otimes (f \circ (\mathrm{id} \otimes \mathrm{id})))(x \otimes (y \otimes z)).$$

The combinator and variable parts can be computed inductively as follows:

- If x is a variable of type X, $\mathrm{cp}(x) = \mathrm{id}\colon X \to X$.
- $\mathrm{cp}(1) = \mathrm{id}\colon I \to I$.
- For any terms s and t, $\mathrm{cp}(s \otimes t) = \mathrm{cp}(s) \otimes \mathrm{cp}(t)$.
- For any term $s : X$ and any combinator $f : X \to Y$, $\mathrm{cp}(f(s)) = f \circ \mathrm{cp}(s)$.

- If x is a variable of type X, $\mathrm{vp}(x) = x$.
- $\mathrm{vp}(1) = 1$.
- For any terms s and t, $\mathrm{vp}(s \otimes t) = \mathrm{vp}(s) \otimes \mathrm{vp}(t)$.
- For any term $s : X$ and any combinator $f\colon X \to Y$, $\mathrm{vp}(f(s)) = \mathrm{vp}(s)$.

Now, suppose that we have a linear type theory. Ambler's first main result is this: there is a symmetric monoidal category where objects are types and morphisms are equivalence classes of combinators. The equivalence relation on combinators is defined as follows: two combinators $f, g \colon X \to Y$ are equivalent if and only if

$$f(t) \sim g(t)$$

for some basic term t of type X. In fact, Ambler shows that $f(t) \sim g(t)$ for *some* basic term $t : X$ if and only if $f(t) \sim g(t)$ for *all* such basic terms.

Ambler's second main result describes how we can build a linear type theory from any closed symmetric monoidal category, say C. Suppose C has composition $\Box$, tensor product $\bullet$, internal hom $\multimap_\bullet$, and unit object ι. We let the basic types of our linear type theory be the objects of C. We take as equations between types those generated by:

- $\iota = I$
- $A \bullet B = A \otimes B$
- $A \multimap_\bullet B = A \multimap B$

We let the function symbols be all the morphisms of C. We take as our equivalence relation on terms the smallest allowed equivalence relation such that:

- $1_A(x) \sim A$
- $(g \,\Box\, f)(x) \sim g(f(x))$
- $(f \bullet g)(x \otimes y) \sim (f(x) \otimes g(y))$
- $a_{A,B,C}((x \otimes y) \otimes z) \sim (x \otimes (y \otimes z))$
- $b_{A,B}(x \otimes y) \sim (y \otimes x)$
- $l_A(1 \otimes x) \sim x$
- $r_A(x \otimes 1) \sim x$
- $\mathrm{ev}_{A,B}(x \otimes \tilde{f}(y)) \sim f(x \otimes y)$

Then we define

- $\mathrm{id} = 1$
- $\mathrm{assoc} = a$
- $\mathrm{unassoc} = a^{-1}$
- $\mathrm{braid} = b$

- left = l
- unleft = l^{-1}
- right = r
- unright = r^{-1}
- eval = ev
- $g \circ f = g \,\square\, f$

and we're done!

Ambler also shows that this procedure is the 'inverse' of his procedure for turning linear type theories into closed symmetric monoidal categories. More precisely, he describes a category of closed symmetric monoidal categories (which is well-known), and also a category of linear type theories. He constructs functors going back and forth between these, based on the procedures we have sketched, and shows that these functors are inverses up to natural isomorphism. So, these categories are 'equivalent'.

In this section we have focused on closed symmetric monoidal categories. What about closed categories that are just braided monoidal, or merely monoidal? While we have not checked the details, we suspect that programming languages suited to these kinds of categories can be obtained from Ambler's formalism by removing various features. To get the braided monoidal case, the obvious guess is to remove Ambler's rewrite rule for the 'braid' combinator and add two rewrite rules corresponding to the hexagon equations (see Section 1.1.4 for these). To get the monoidal case, the obvious guess is to completely remove the combinator 'braid' and all rewrite rules involving it. In fact, Jay [104] gave a language suitable for closed monoidal categories in 1989; Ambler's work is based on this.

1.4 Conclusions

In this chapter 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.

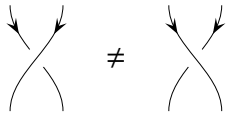
However, we only scratched the surface! There is much more to say about categories equipped with extra structure, and how we can use them to strengthen the ties between physics, topology, logic and computation—not to mention what happens when we go from categories to n-categories, as we do in the next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

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 $f\colon X \to Y$
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: $X \otimes Y$
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: $f \otimes g$
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: $X \multimap Y$

Table 1.4: The Rosetta Stone (larger version)

In the late 1980s, Witten [203] realized that string theory was deeply connected to a 3d topological quantum field theory and thus the theory of knots and tangles [128]. 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' [24, 200].

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 [187, 188]. 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:

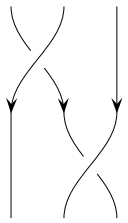


So, collections of anyons are described by objects in a braided monoidal category! The details depend on things like the strength of the magnetic field; the range of possibilities can be worked out with the help of modular tensor categories [154, 168].

So far this is all about physics and topology. Computation entered the game around 2000, when Freedman, Kitaev, Larsen and Wang [80] 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 implica-

tions 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.

One failing of the approach in this chapter is the mismatch around the treatment of *time*. A computer is a physical device, and we could, at least in principle, use Feynman diagrams to describe it. However, the analogy described in this chapter associates a Feynman diagram with an equivalence class of programs that *mods out by the process of computation*: the equivalence class explicitly ignores changes that happen over time!

If we want a categorical semantics for computation that treats time in the same way that physics does, we need to go up one dimension from categories to "bicategories". The next chapter defines compact closed bicategories and proves that a particular bicategory of spans is compact closed as preparation for future work outlined in the conclusion.

Chapter 2

Compact Closed Bicategories

2.1 Introduction

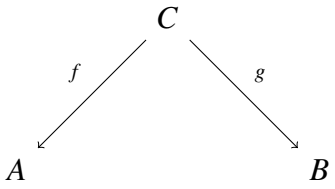
When moving from set theory to category theory and higher to n-category theory, we typically weaken equations at one level to natural isomorphisms at the next. These isomorphisms are then subject to new coherence equations. For example, multiplication in a monoid is associative, but the tensor product in a monoidal category is, in general, only associative up to a natural isomorphism. This "associator" natural isomorphism has to satisfy some extra equations that are trivial in the case of the monoid. In a similar way, when we move from compact closed categories to compact closed bicategories, the "zig-zag" equations governing the dual get weakened to natural isomorphisms and we need to introduce some new coherence laws.

In Section 2.2, we will give several examples of important mathematical structures and how they arise in relation to compact closed bicategories. Following the examples, in Section 2.3 we give the history of the concept and related work. Next, in Section 2.4 we give the complete definition, which to our knowledge has not appeared elsewhere; we try to motivate each piece of the definition so that the reader could afterwards reconstruct the definition without the aid of this thesis. In Section 2.5, we prove that a construction by Hoffnung is an instance of a compact closed bicategory and a few others as corollaries.

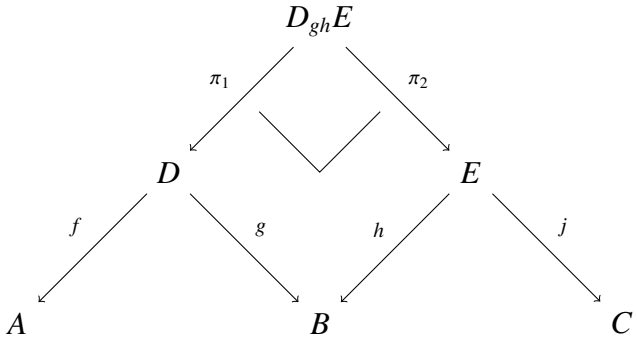
2.2 Examples

In order to get across some of the flavor of these different compact closed bicategories, we will describe the bicategories as well as some weak monoids and monads in them. That these bicategories are compact closed is mostly folklore; we prove that a few of them are compact closed as corollaries of the main theorem at the end of this chapter. The weak monoids and monads play no role in the rest of the chapter, but we have found that comparing and contrasting them helps when trying to develop intuition about the bicategories. The monoids are variations on the notion of an associative algebra, while the monads are variations on the notion of a category.

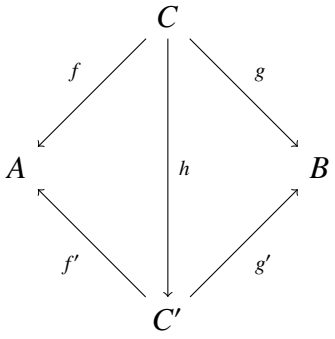
- A **span** from A to B in a category T is an object C in T together with an ordered pair of morphisms $(f\colon C \to A, g\colon C \to B)$.



If T is a category with pullbacks, we can compose spans:



A **map of spans** between two spans $A \xleftarrow{f} C \xrightarrow{g} B$ and $A \xleftarrow{f'} C' \xrightarrow{g'} B$ is a morphism $h : C \to C'$ making the following diagram commute:



Since the pullback is associative only up to a natural isomorphism, the same is true of the composition of two spans, so this construction does not give a 2-category; however, we do get a bicategory Span(T) of objects of T, spans in T, and maps of spans.

If T is a category with finite products as well as pullbacks, then the bicategory Span(T) is a compact closed bicategory where the tensor product is given by the product in T. A weak monoid object in Span(T) is a categorification of the notion of an associative algebra. For example, one weak monoid in Span(Set) is equivalent to the category of polynomial functors from Set to itself; such functors can be "added" using disjoint union, "multiplied" using the cartesian product, and "scaled" by sets [82]. A monad in Span(T) is a category internal to T [32].

- Sets, relations, and implications form the compact closed bicategory Rel, where the tensor product is given by the product in Set. A weak monoid object M in Rel is a quantale on the powerset of M [180], while a monad in Rel is a preorder.

- A **2-rig** is a cocomplete monoidal category where the tensor product distributes over the colimits [19], though for the purpose of constructing a compact closed bicategory we only need the tensor product to distribute over finite coproducts. Given a symmetric 2-rig R, Mat(R) is the compact closed bicategory of finitely-generated free R-modules, where the tensor product is the usual tensor product for matrices. A weak monoid object in Mat(R) is a categorified finite-dimensional associative algebra over R. A monad in Mat(R) is a finite R-enriched category. A **finite 2-rig** S is only finitely cocomplete, but Mat(S) is still compact closed. Kapranov and Voevodsky [112] described a bicategory equivalent to Mat(FinVect) and called its objects "2-vector spaces".

- For a category C, let $\widehat{C} = \mathrm{Set}^{C^{\mathrm{op}}}$ be the category of presheaves on C. The 2-category Cocont has

  - small categories as objects;
  - cocontinuous functors $f : \widehat{C} \to \widehat{D}$ between the categories of presheaves on the source and target as morphisms;
  - natural transformations as 2-morphisms.

  We can think of cocontinuous functors as being "Set-linear transformations", since they preserve sums. Day and Street [72] proved that Cocont is a compact closed 2-category.

- Recall that a profunctor $F : C \nrightarrow R$ is a functor $F : C^{\mathrm{op}} \times R \to \mathrm{Set}$; we can think of profunctors as being rather like matrices, where the set $F(c, r)$ is the "matrix element" at row r and column c. Composition of profunctors is given by taking the coend of the inner coordinates, just as matrix multiplication done by summing over the inner index. Small categories, profunctors, and natural transformations form the compact closed bicategory Prof, where the tensor product is the product in Cat. A weak monoid object in Prof is a promonoidal category [71, 100]. A symmetric monoidal monad in Prof is a Freyd category, also known as an "Arrow" in the functional programming community [10, 103].

  Cattani and Winskel [55] showed that Cocont and Prof are equivalent as bicategories. Though they do not explicitly state it, the equivalence they construct is symmetric monoidal; since symmetric monoidal equivalences preserve the dual, Cocont and Prof are equivalent as compact closed bicategories.

- So far the examples have been rather algebraic in flavor, but there are topological examples, too. The category nCob is the compact closed category whose

  - objects are $(n-1)$-dimensional manifolds and
  - morphisms are diffeomorphism classes of collared n-dimensional cobordisms between them,

where the tensor product is disjoint union. Atiyah [12] introduced the category informally in his paper defining topological quantum field theories.

Morton [155] defined the bicategory $n\mathrm{Cob}_2$ whose

- objects are $(n-2)$-dimensional manifolds,
- morphisms are collared $(n-1)$-dimensional cobordisms, or "manifolds with boundary", and
- 2-morphisms are diffeomorphism classes of collared n-dimensional maps of cobordisms, or "manifolds with corners".

The collars are necessary to preserve the smoothness when composing 1- and 2-morphisms. Schommer-Pries proved a purely algebraic characterization of $2\mathrm{Cob}_2$, essentially proving the "Baez-Dolan cobordism hypothesis" for the $n = 2$ case [18].

- In a letter to the author, John Baez defined two interesting compact closed bicategories.

  A **directed multigraph** is a finite set E of edges and a finite set V of vertices equipped with functions $s, t : E \to V$ mapping each edge to its source and target. A **resistor network** is a directed multigraph equipped with a function r assigning a resistance in $(0, \infty)$ to each edge:

$$(0,\infty) \xleftarrow[r]{} E \underset{t}{\overset{s}{\rightrightarrows}} V$$

  There are various choices one could make for a morphism of such networks; Baez defined a morphism of resistor networks to be a pair of functions ϵ, υ making the following diagrams commute:

$$\begin{array}{ccccc} & & E & & \\ & \overset{r}{\swarrow} & & \searrow^{\epsilon} & \\ (0,\infty) & & & & E' \\ & \underset{r'}{\nwarrow} & & \nearrow & \end{array} \qquad \begin{array}{ccc} E & \xrightarrow{s} & V \\ \downarrow{\scriptstyle \epsilon} & & \downarrow{\scriptstyle \upsilon} \\ E' & \xrightarrow[s]{} & V' \end{array} \qquad \begin{array}{ccc} E & \xrightarrow{t} & V \\ \downarrow{\scriptstyle \epsilon} & & \downarrow{\scriptstyle \upsilon} \\ E' & \xrightarrow[t]{} & V' \end{array}$$

  Resistor networks and morphisms between them form a category ResNet; this category has finite limits and colimits.

  There is a compact closed bicategory Cospan(ResNet) with an important compact closed subbicategory Circ consisting of cospans whose feet are resistor networks with no edges. A morphism in Circ is a **circuit,** a resistor network with chosen sets of input and output vertices across which one can measure a voltage drop.

2.3 Previous work

Compact closed categories were first defined by Kelly [120], and later studied in depth by Kelly and Laplaza [122].

Bénabou [32] defined bicategories and showed that small categories, distributors, and natural transformations form a bicategory Dist. Distributors later became more widely known as "profunctors", so we will call that bicategory "Prof" instead. Later, Bénabou defined closed bicategories and showed that Prof is closed [33]. He defined V-enriched profunctors when V is a cocomplete monoidal or symmetric monoidal closed category, defined V-Prof and proved that any V-enriched functor, regarded as a V-profunctor, has a right adjoint. More applications and details are in his lecture notes [31].

Kapranov and Voevodsky [112] defined braided semistrict monoidal 2-categories, but their definition left out some necessary axioms. Baez and Neuchl [22] gave an improved definition, but it was still missing a clause; Crans [66] gave the complete definition. See Baez and Langford [20] and Shulman [179] for details.

Gordon, Power, and Street [89] defined fully weak tricategories; a monoidal bicategory is a one-object tricategory.

Another name for semistrict monoidal 2-categories is "Gray monoids", *i.e.* monoid objects in the 2-category Gray [88]. Day and Street [72] defined compact closed Gray monoids, and appealed to the coherence theorem of Gordon, Power, and Street to extend compact closedness to arbitrary bicategories. The semistrict approach is somewhat artificial when dealing with most "naturally occurring" bicategories, since the associator for composition of 1-morphisms is rarely the identity. Cocont is a notable exception.

Katis, Sabadini and Walters [115] gave a precise account of the double-entry bookkeeping method *partita doppia* in terms of the compact closed bicategory Span(RGraph); in a later paper [116], they cite a handwritten note by McCrudden for the "swallowtail" coherence law we use in this thesis.

Preller and Lambek [163] generalized compact monoidal categories in a different direction. They considered a compact monoidal category to be a one-object bicategory satisfying some criteria, and then extend that definition to multiple objects. The resulting concept of "compact bicategory" is *not* what is being studied in this thesis.

McCrudden [150] gave the first fully general definitions of braided, sylleptic, and symmetric monoidal bicategories. Schommer-Pries [171] gave the correct notion of a monoidal transformation between monoidal functors between monoidal bicategories.

Carboni and Walters [54] proved that V-Prof is a cartesian bicategory. Later, they showed [55] that Prof is equivalent to Cocont as a bicategory. Together with Kelly and Wood [53], they proved that any cartesian bicategory is symmetric monoidal in the sense of McCrudden.

2.4 Compact closed bicategories

In this section, we lay out the definition of a compact closed bicategory. First we give the definition of a bicategory, then start adding structure to it: we introduce the tensor product and monoidal unit; then we look at the different ways to move objects around each other, giving braided, sylleptic and symmetric monoidal bicategories. Next, we define closed monoidal bicategories by introducing a right pseudoadjoint to tensoring with an object; and finally we introduce duals for objects in a bicategory.

Definition 1 *A* **bicategory** $\mathcal{K}$ *consists of*

1. *a collection of* **objects**

2. *for each pair of objects* A, B *in* $\mathcal{K}$*, a category* $\mathcal{K}(A, B)$*; the objects of* $\mathcal{K}(A, B)$ *are called* **1-morphisms***, while the morphisms of* $\mathcal{K}(A, B)$ *are called* **2-morphisms**.

3. *for each triple of objects* A, B, C *in* $\mathcal{K}$*, a* **composition** *functor*

$$\circ_{A,B,C} \colon \mathcal{K}(B, C) \times \mathcal{K}(A, B) \to \mathcal{K}(A, C).$$

   We will leave off the indices and write it as an infix operator.

4. *for each object* A *in* $\mathcal{K}$*, an object* 1_A *in* $\mathcal{K}(A, A)$ *called the* **identity 1-morphism on** A*. We will often write this simply as* A*.*

5. *for each quadruple of objects* A, B, C, D*, a natural isomorphism called the* **associator for composition***; if* (f, g, h) *is an object of* $\mathcal{K}(C, D) \times \mathcal{K}(B, C) \times \mathcal{K}(A, B)$*, then*

$$\mathring{a}_{f,g,h} \colon (f \circ g) \circ h \to f \circ (g \circ h).$$

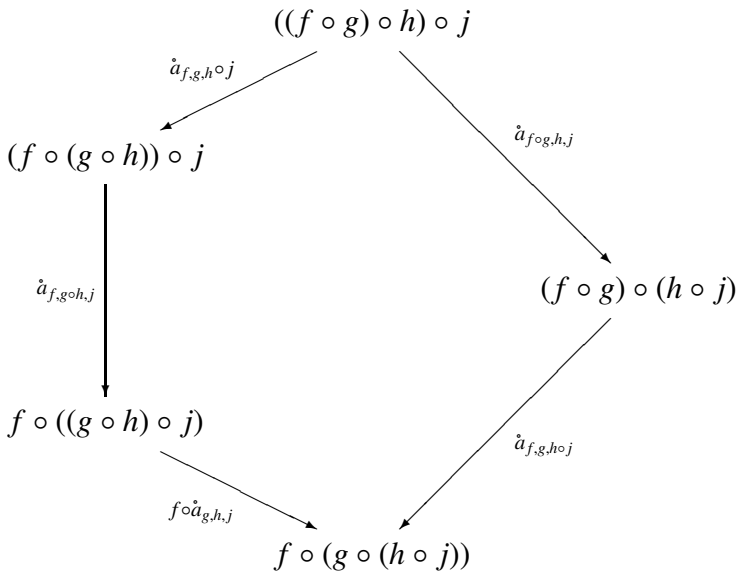
6. *for each pair of objects* A, B *in* $\mathcal{K}$*, natural isomorphisms called* **left and right unitors for composition.** *If* f *is an object of* $\mathcal{K}(A, B)$*, then*

$$\mathring{l}_f \colon B \circ f \xrightarrow{\sim} f$$
$$\mathring{r}_f \colon f \circ A \xrightarrow{\sim} f$$

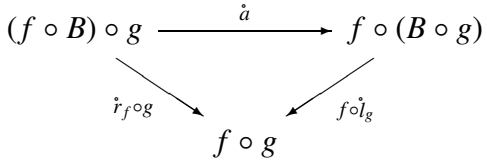
such that $\mathring{a}, \mathring{l}$*, and* $\mathring{r}$ *satisfy the following coherence laws:*

1. *for all* (f, g, h, j) *in* $\mathcal{K}(D, E) \times \mathcal{K}(C, D) \times \mathcal{K}(B, C) \times \mathcal{K}(A, B)$*, the following diagram, called*

the **pentagon equation**, *commutes:*



2. *for all* (f, g) *in* $\mathcal{K}(B, C) \times \mathcal{K}(A, B)$ *the following diagram, called the* **triangle equation**, *commutes:*



The associator $\mathring{a}$ and unitors $\mathring{r}, \mathring{l}$ for composition of 1-morphisms are necessary, but when we are drawing commutative diagrams of 1-morphisms they are very hard to show; fortunately, by the coherence theorem for bicategories [138], any consistent choice is equivalent to any other, so we leave them out.

We refer the reader to Tom Leinster's excellent "Basic bicategories" [138] for definitions of

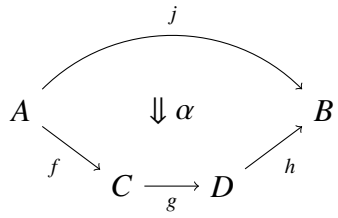
- morphisms of bicategories, which we call functors,
- transformations between functors, which we call pseudonatural transformations, and
- modifications between transformations.

Definition 2 *An* **equivalence** *of objects* A, B *in a bicategory is a pair of morphisms* $f\colon A \to B, g\colon B \to A$ *together with invertible 2-morphisms* $e : g \circ f \xRightarrow{\sim} 1_A$ *and* $i : f \circ g \xRightarrow{\sim} 1_B$.

Definition 3 *An* **adjoint equivalence** *is one in which the 2-morphisms* e *and* i^{-1} *exhibit that* g *is left adjoint to* f.

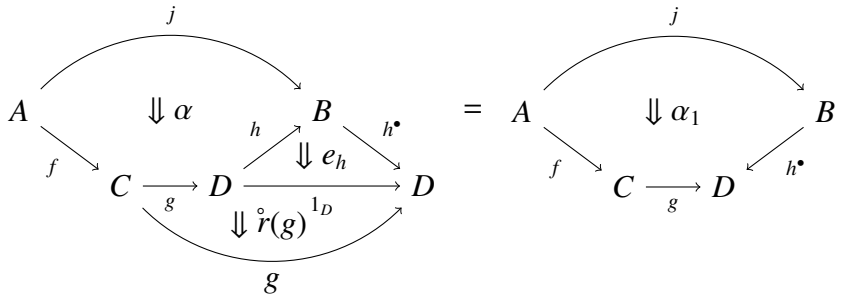
For a given morphism f, any two choices of data (g, e, i) making f an adjoint equivalence are canonically isomorphic, so any choice is as good as any other. When f, g form an adjoint equivalence, we write $g = f^\bullet$. Any equivalence can be improved to an adjoint equivalence.

We can often take a 2-morphism and "reverse" one of its edges. Given objects A, B, C, D, morphisms $f : A \to C$, $g : C \to D$, $h : D \to B$, $j : A \to B$ such that h is an adjoint equivalence, and a 2-morphism



we can get a new 2-morphism

$$(\mathring{r}(g) \circ f)(e_h \circ g \circ f)(h^\bullet \circ \alpha) : h^\bullet \circ j \Rightarrow g \circ f,$$



where $e_h : h^\bullet \circ h \Rightarrow 1$ is the 2-morphism from the equivalence. We denote such variations of a 2-morphism by adding numeric subscripts; the number simply records the order in which we introduce them, not any information about the particular variation.

In the following definitions, I have given some plausible combinatorical reasoning justifying many of the parts of the definition, but except where noted, this is not part of the definition; its intent is merely to help organize the rather long and dry content. I am not aware of any work on the combinatorics of cells in higher categories beyond that mentioned below by Stasheff, Kapranov and Voevodsky.

Also, some of the illustrations of 2-morphisms and coherence laws below are quite large. In order to preserve legibility, I use expressions like $(AB)C$ as a shorthand for functors like

$$\otimes \circ (\otimes \times 1) : \mathcal{K}^3 \to \mathcal{K},$$

since parentheses suffice to show where the tensor product should be.

Definition 4 *A* **monoidal bicategory** $\mathcal{K}$ *is a bicategory in which we can "multiply" objects. It consists of the following:*

- *A bicategory $\mathcal{K}$.*

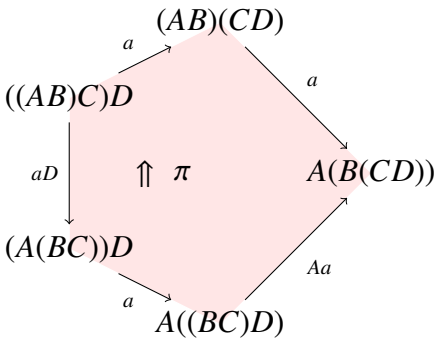
- *A* **tensor product** *functor $\otimes : \mathcal{K} \times \mathcal{K} \to \mathcal{K}$. This functor involves an invertible "tensorator" 2-morphism $(f \otimes g) \circ (f' \otimes g') \Rightarrow (f \circ f') \otimes (g \circ g')$ which we elide in most of the coherence equations below. The coherence theorem for monoidal bicategories implies that any 2-morphism involving the tensorator is the same no matter how it is inserted [90, Remark 3.1.6], so like the associator for composition of 1-morphisms, we leave it out.*

  The **Stasheff polytopes** *[185] are a series of geometric figures whose vertices enumerate the ways to parenthesize the tensor product of n objects, so the number of vertices is given by the Catalan numbers; for each polytope, we have a corresponding $(n-2)$-morphism of the same shape with directed edges and faces:*

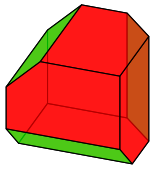
  1. *The tensor product of one object A is the one object A itself.*
  2. *The tensor product of two objects A and B is the one object (AB).*
  3. *There are two ways to parenthesize the product of three objects, so we have an* **associator** *adjoint equivalence pseudonatural in A, B, C*

     $$a \colon (AB)C \to A(BC)$$

     for moving parentheses from the left pair to the right pair.
  4. *There are five ways to parenthesize the product of four objects, so we have a* **pentagonator** *invertible modification π relating the two different ways of moving parentheses from being clustered at the left to being clustered at the right. (Mnemonic: Pink Pentagonator.)*

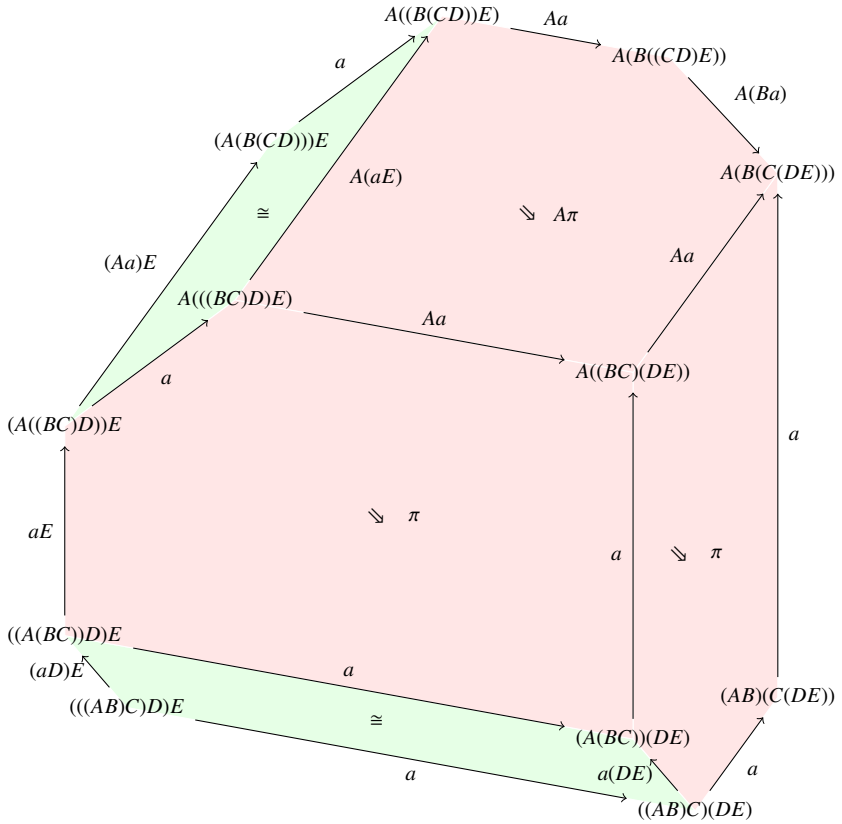
     

  5. *There are fourteen ways to parenthesize the product of five objects, so we have an* **associahedron** *equation of modifications with fourteen vertices relating the various ways of getting from the parentheses clustered at the left to clustered at the right.*

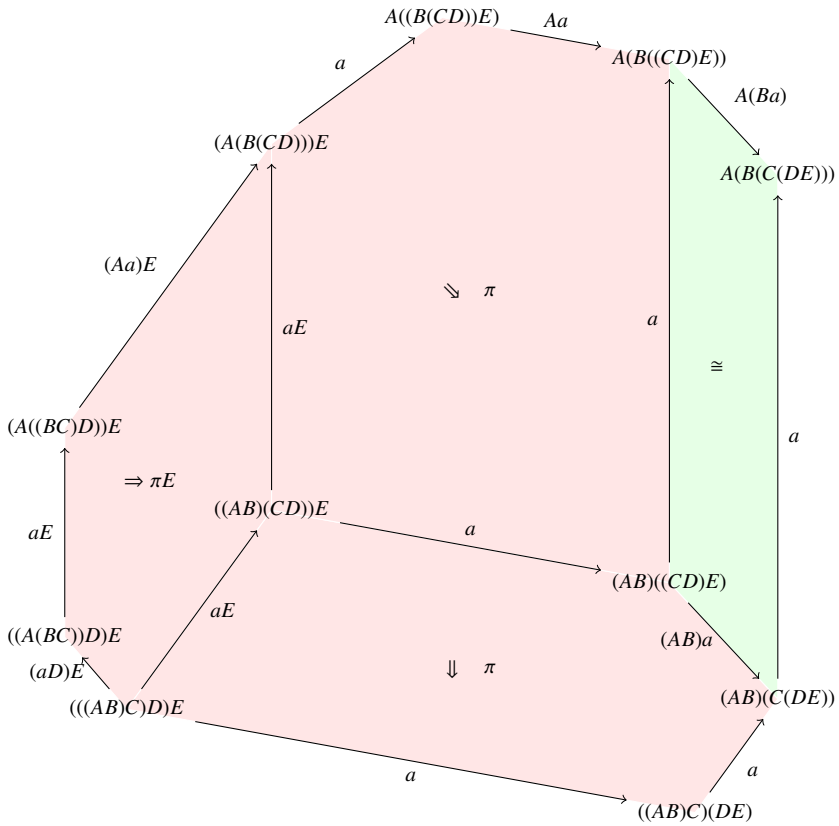
     

     The associahedron is a cube with three of its edges bevelled. It holds in the bicategory $\mathcal{K}$, where the unmarked 2-morphisms are instances of the pseudonaturality invertible

modification for the associator. (Mnemonic for the rectangular invertible modifications: GReen conGRuences.)



=



- *Just as in any monoid there is an identity element 1, in every monoidal bicategory there is a* **monoidal unit** *object I. Associated to the monoidal unit are a series of morphisms—one of each dimension—that express how to "cancel" the unit in a product. Each morphism of dimension $n > 0$ has two Stasheff polytopes of dimension $n-1$ as "subcells", one for parenthesizing $n+1$ objects and the other for parenthesizing the n objects left over after cancellation. There are $n+1$ ways to insert I into n objects, so there are $n+1$ morphisms of dimension n.*

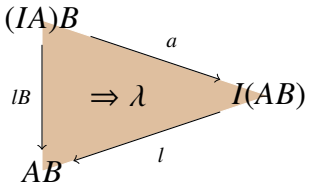
  1. *There is one monoidal unit object I.*

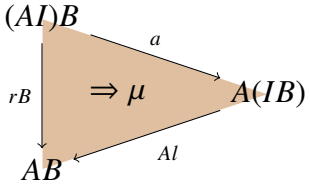
  2. *There are two* **unitor** *adjoint equivalences l and r that are pseudonatural in A. The Stasheff polytopes for two objects and for one object are both points, so the unitors are line segments joining them.*

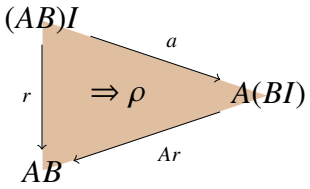
$$l\colon IA \to A$$

$$r\colon AI \to A.$$

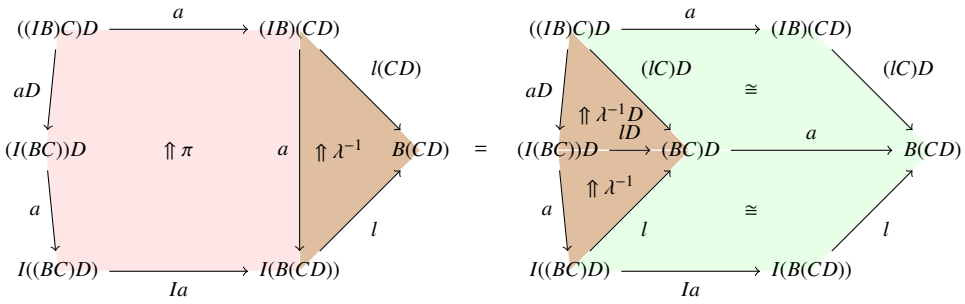
  3. *There are three* **2-unitor** *invertible modifications λ, μ, and ρ. The Stasheff polytope for three objects is a line segment and the Stasheff polytope for two objects is a point, so these modifications are triangles. (Mnemonic: Umber Unitor.)*

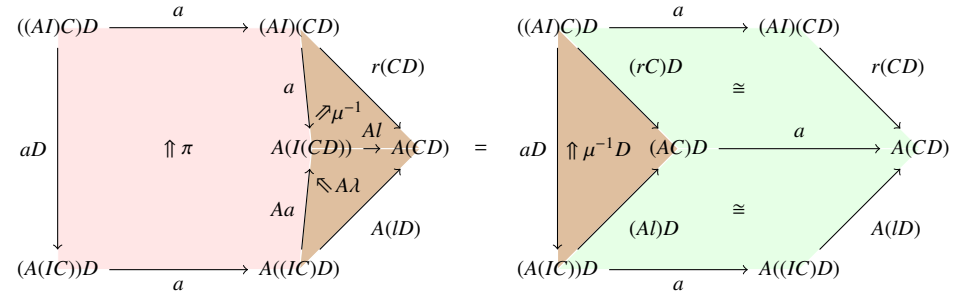


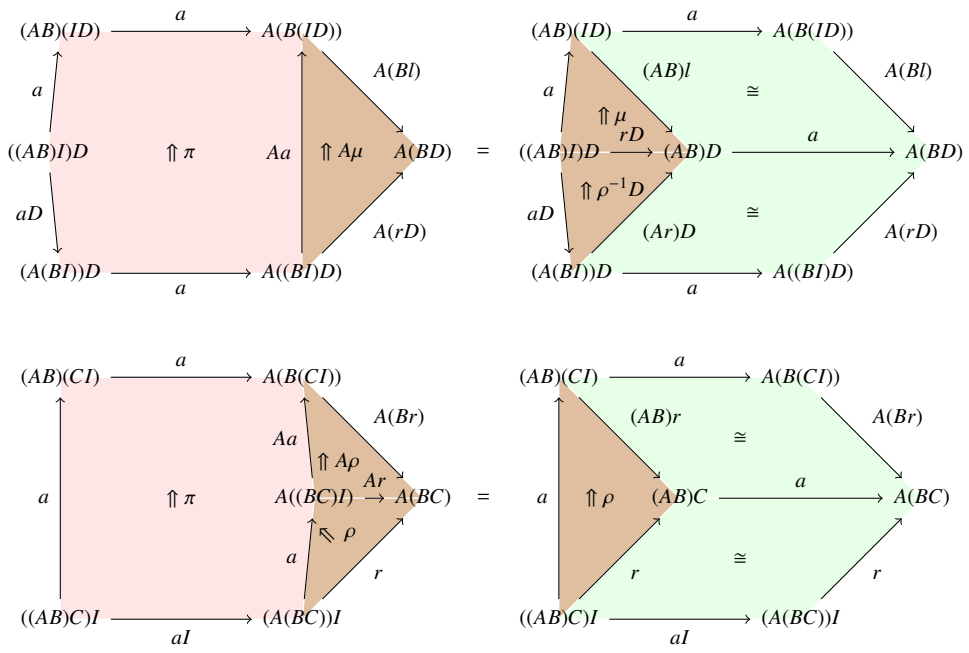




4. *There are four equations of modifications. The Stasheff polytope for four objects is a pentagon and the Stasheff polytope for three objects is a line segment, so these equations are irregular prisms with seven vertices.*







Definition 5 *A* **braided** *monoidal bicategory $\mathcal{K}$ is a monoidal bicategory in which objects can be moved past each other. A braided monoidal bicategory consists of the following:*

- *A monoidal bicategory $\mathcal{K}$;*
- *A series of morphisms for "shuffling".*

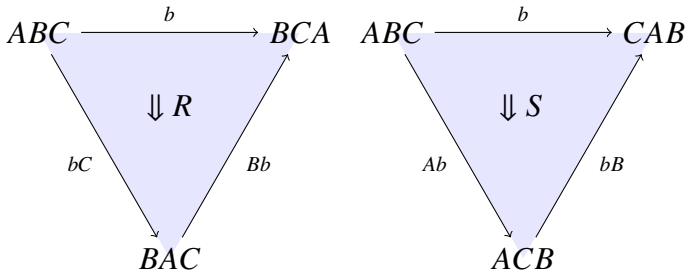
Definition 6 *A* **shuffle** *of a list $\mathcal{A} = (A_1, \ldots, A_n)$ into a list $\mathcal{B} = (B_1, \ldots, B_k)$ inserts each element of $\mathcal{A}$ into $\mathcal{B}$ such that if $0 < i < j < n + 1$ then A_i appears to the left of A_j.*

An "(n, k)-shuffle polytope" is an n-dimensional polytope whose vertices are all the different shuffles of an n-element list into a k-element list; there are $\binom{n+k}{k}$ ways to do this. General shuffle polytopes were defined by Kapronov and Voevodsky [112]. As with the Stasheff polytopes, we have morphisms of the same shape as (n, k)-shuffle polytopes with directed edges and faces.

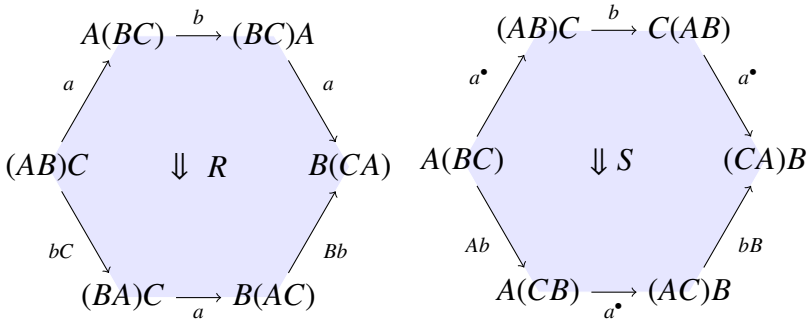
- *$(n = 1, k = 1)$: $\binom{1+1}{1} = 2$, so this polytope has two vertices, (A, B) and (B, A). It has a single edge, which we call a "braiding", which encodes how A moves past B. It is an adjoint equivalence pseudonatural in A, B.*

$$b : AB \to BA$$

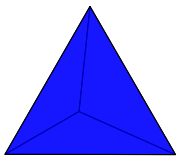
- *$(n = 1, k = 2)$ and $(n = 2, k = 1)$: $\binom{1+2}{1} = \binom{2+1}{1} = 3$, so whenever the associator is the identity*—e.g. *in a braided strictly monoidal bicategory—these polytopes are triangles, invertible modifications whose edges are the directed (1,1) polytope, the braiding.*



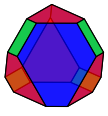
When the associator is not the identity, the triangles' vertices get replaced with associators, effectively truncating them, and we are left with hexagon invertible modifications. (Mnemonic: Blue Braiding.)



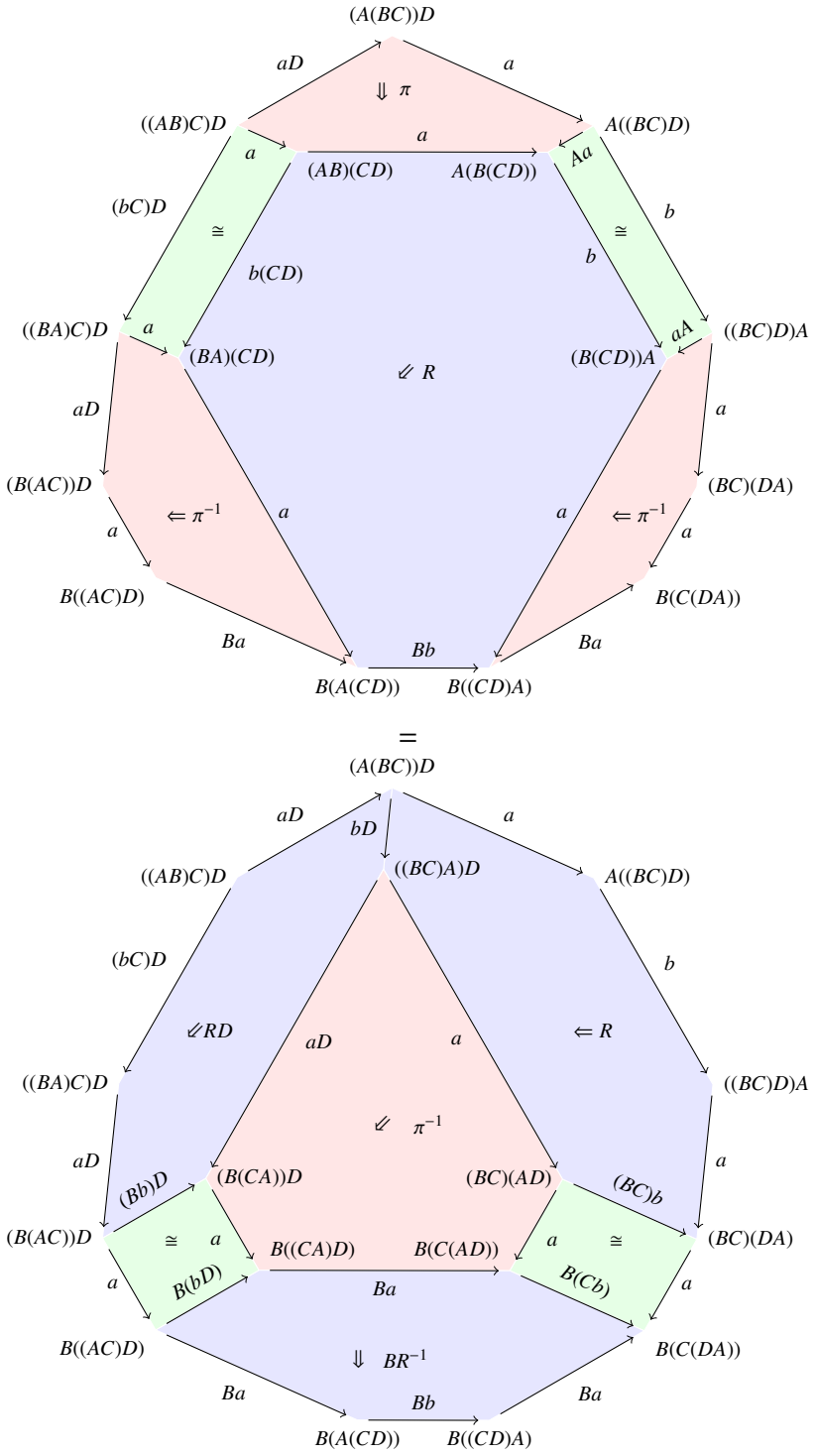
– $(n = 3, k = 1)$ *and* $(n = 1, k = 3)$*:* $\binom{3+1}{1} = \binom{1+3}{1} = 4$*, so in a braided strictly monoidal bicategory, these polytopes are tetrahedra whose faces are the* $(2, 1)$ *polytope.*



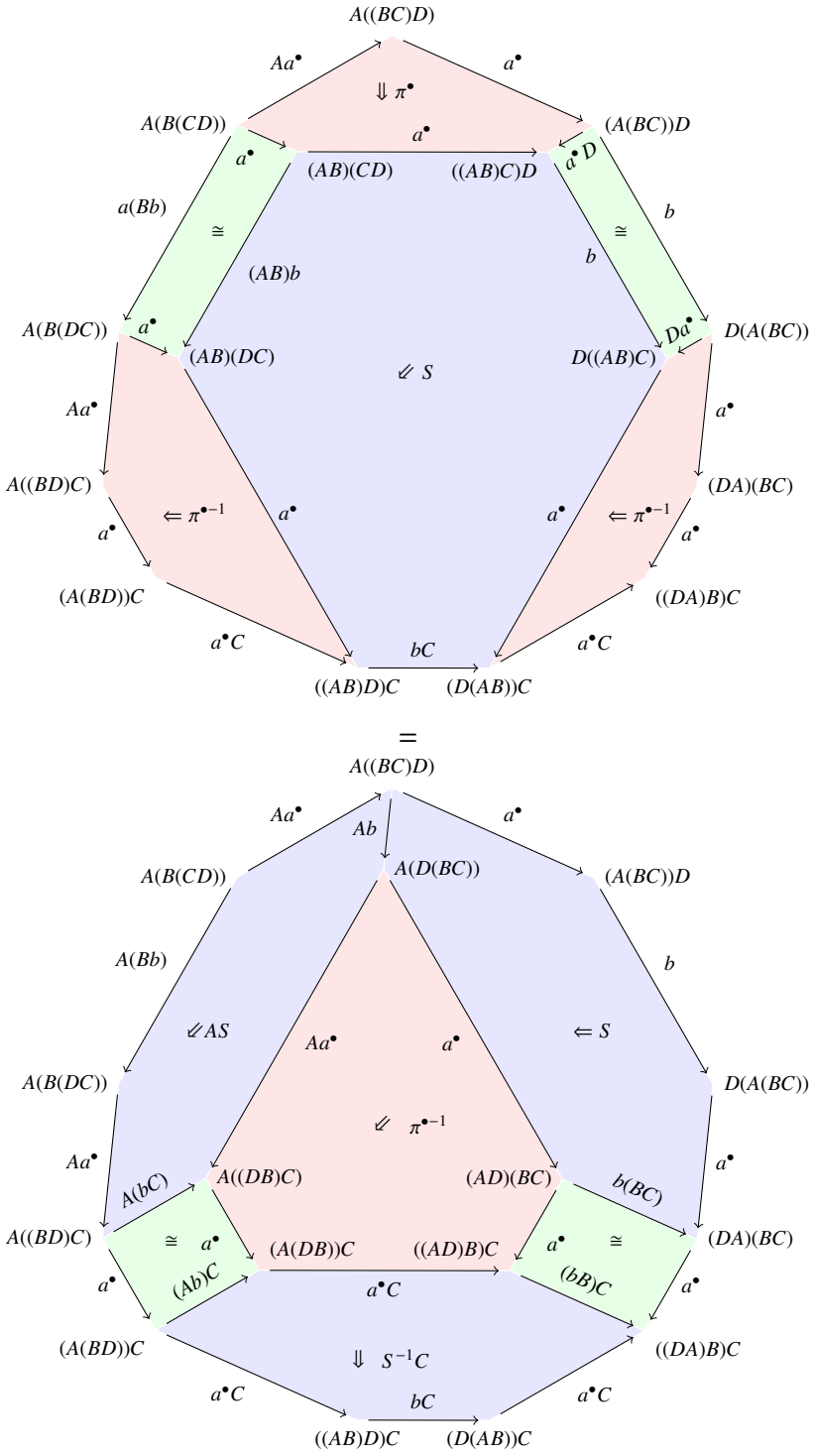
Again, when the associator is not the identity, the vertices get truncated, this time being replaced by pentagonators; as a side-effect, four of the six edges are also beveled.



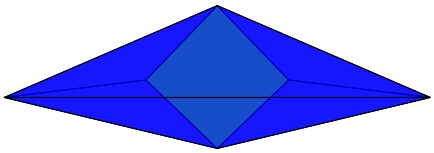
This equation governs shuffling one object A into three objects B, C, D*:*



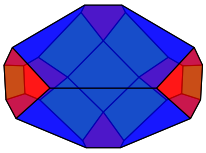
This equation governs shuffling one object D into three objects A, B, C:



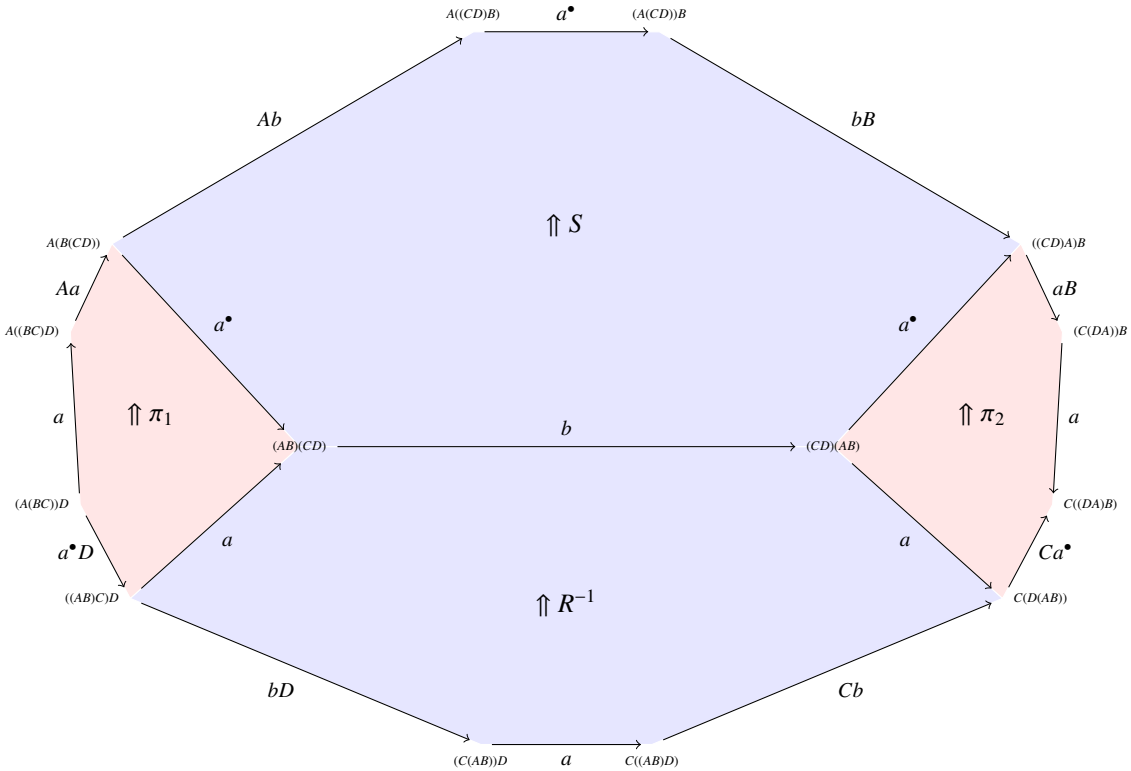
– $(n = 2, k = 2)$: $\binom{2+2}{2} = 6$; *in a braided strictly monoidal bicategory, this polytope is composed mostly of (2,1) triangles, but there is a pair of braidings that commute, so one face is a square.*



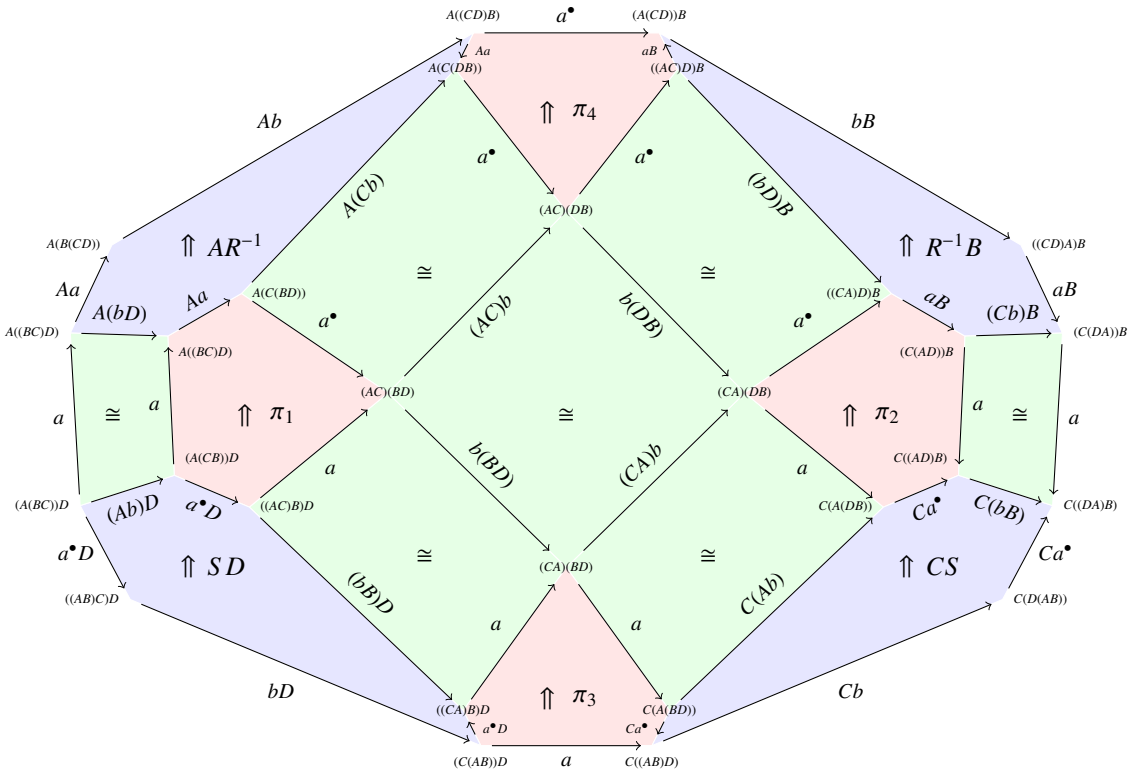
When the associator is not the identity, the six vertices get truncated and six of the edges get beveled.



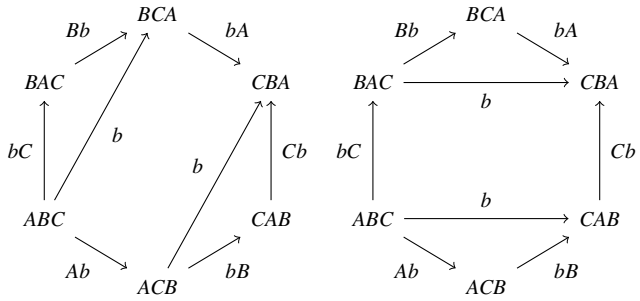
This equation governs shuffling two objects A, B into two objects C, D:



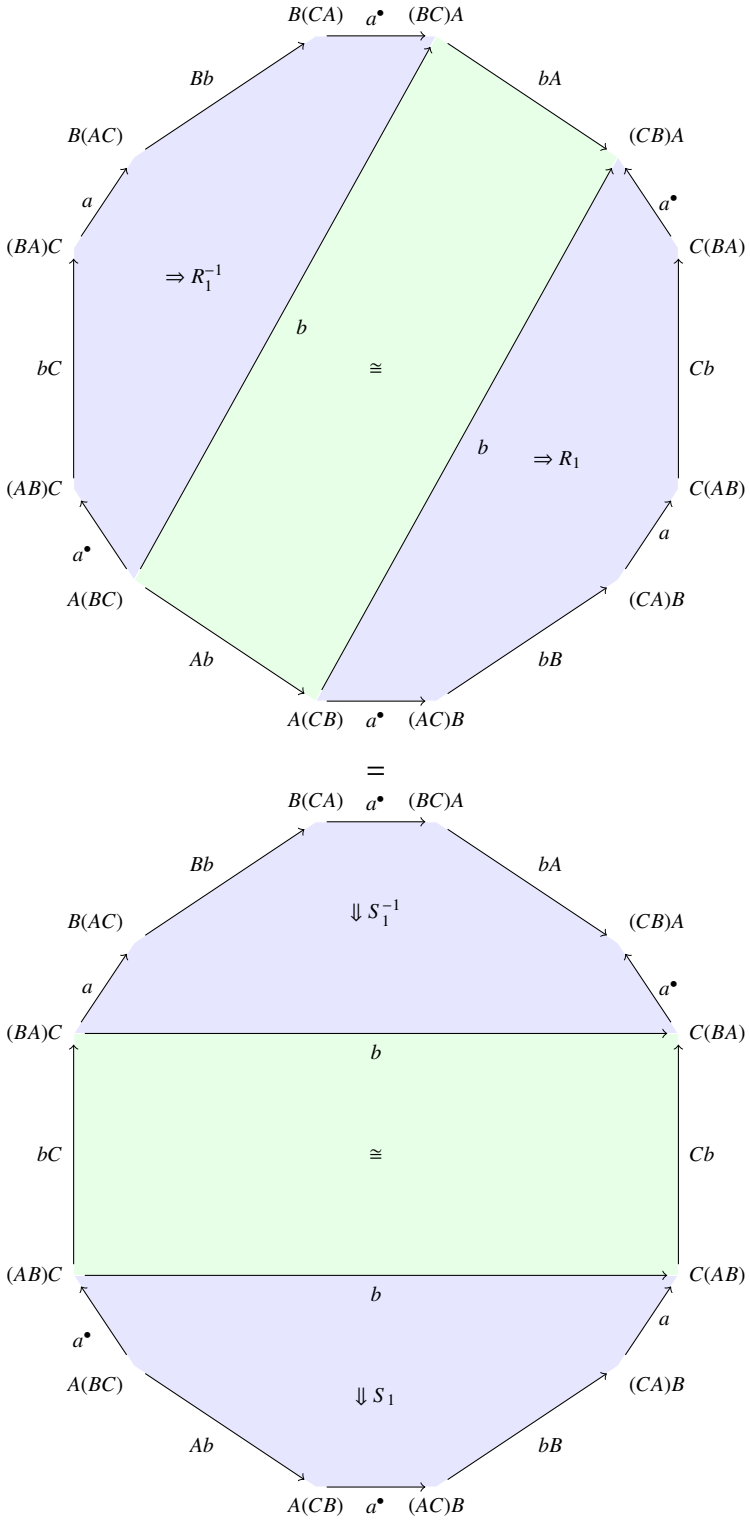
=



- *The Breen polytope. In a braided monoidal category, the Yang-Baxter equations hold; there are two fundamentally distinct proofs of this fact.*

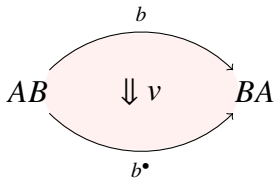


In a braided strictly monoidal bicategory, the two proofs become the front and back face of another coherence law governing the interaction of the (2,1)-shuffle polytopes; when the associator is nontrivial, the vertices get truncated. That the coherence law is necessary was something of a surprise: Kapranov and Voevodsky did not include it in their definition of braided semistrict monoidal 2-categories; Breen [40] corrected the definition. We therefore call the following coherence law the "Breen polytope". In retrospect, we can see that this is the start of a more subtle collection of polytopes relevant to braided monoidal n-categories, which can be systematically obtained using Batanin's approach to weak n-categories [28].



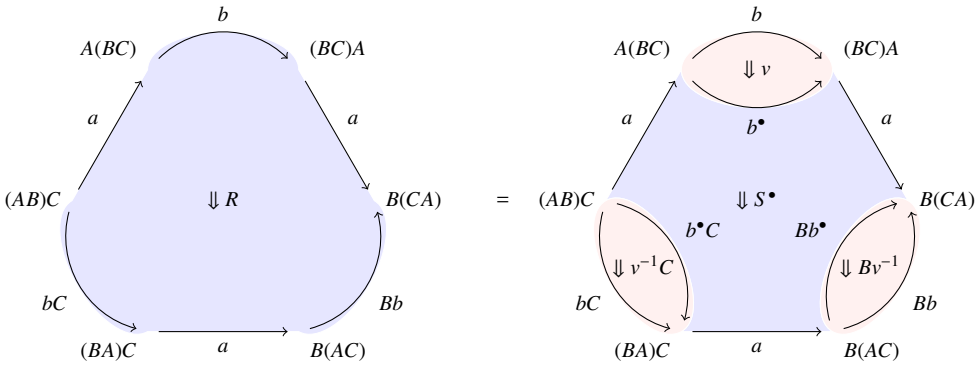
Definition 7 *A* **sylleptic** *monoidal bicategory $\mathcal{K}$ is a braided monoidal bicategory equipped with*

- *an invertible modification called the syllepsis, (Mnemonic: Salmon Syllepsis)*

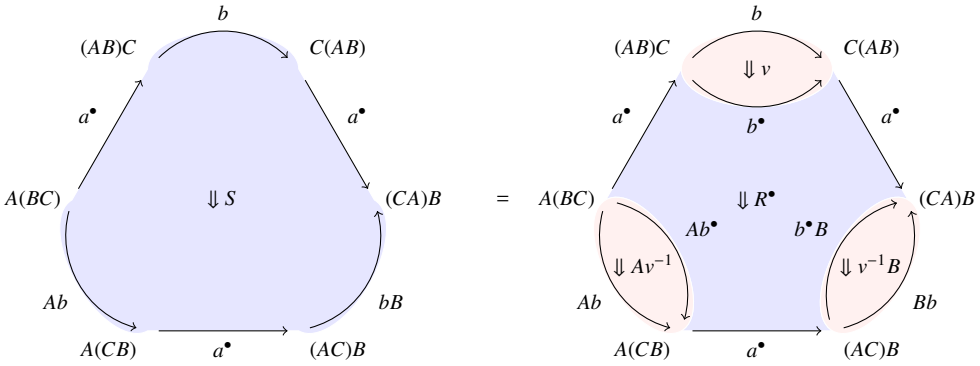


subject to the following axioms.

- *This equation governs the interaction of the syllepsis with the* $(n = 1, k = 2)$ *braiding:*

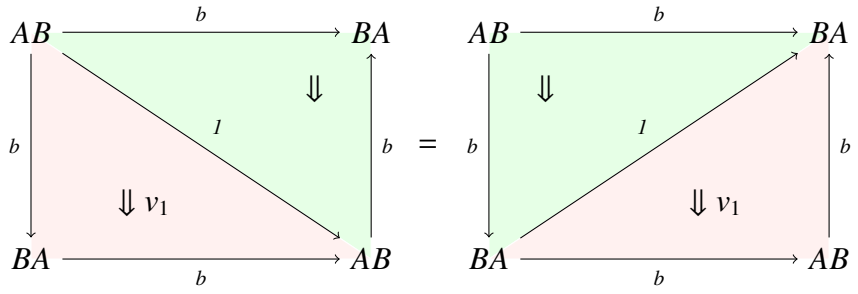


- *This equation governs the interaction of the syllepsis with the* $(n = 2, k = 1)$ *braiding:*



Definition 8 *A* **symmetric** *monoidal bicategory is a sylleptic monoidal bicategory subject to the following axiom, where the unlabeled green cells are identities:*

- *for all objects A and B of* $\mathcal{K}$*, the following equation holds:*



Definition 9 *Given two bicategories $\mathcal{J}, \mathcal{K}$, two functors $L\colon \mathcal{J} \to \mathcal{K}$ and $R\colon \mathcal{K} \to \mathcal{J}$ are* **pseudoadjoint** *if for all $A \in \mathcal{J}, B \in \mathcal{K}$ the categories $Hom_{\mathcal{K}}(LA, B)$ and $Hom_{\mathcal{J}}(A, RB)$ are adjoint equivalent pseudonaturally in A and B.*

Monoidal closed bicategories satisfy the obvious weakening of the definition of monoidal closed categories:

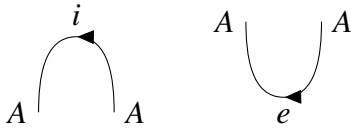
Definition 10 *A monoidal* **closed** *bicategory is one in which for every object A, the functor $- \otimes A$ has a right pseudoadjoint $A \multimap -$.*

Similarly, compact closed bicategories weaken the notion of duality from compact closed categories. In the following definition, we abstract the notion of pseudoadjointness from functors between bicategories to arbitrary objects of a bicategory.

Definition 11 *A* **compact closed** *bicategory is a symmetric monoidal bicategory in which every object has a pseudoadjoint.*

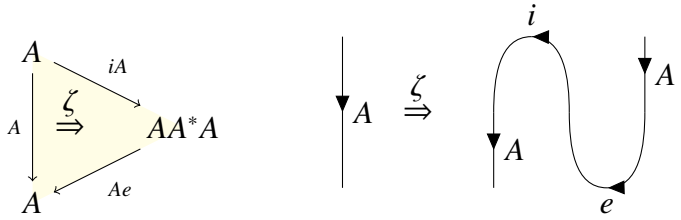
This means that every object A is equipped with a (weak) **dual**, an object A^* equipped with two 1-morphisms

$$i_A : I \to AA^* \qquad e_A : A^*A \to I$$

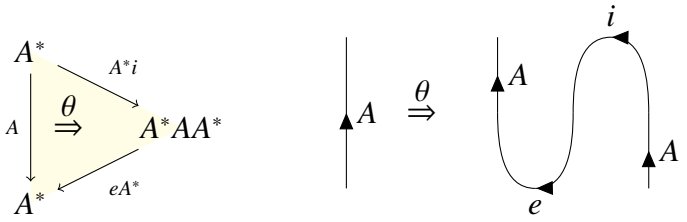


and two "zig-zag" 2-isomorphisms (Mnemonic: Yellow Yanking or Xanthic Zig-zag)

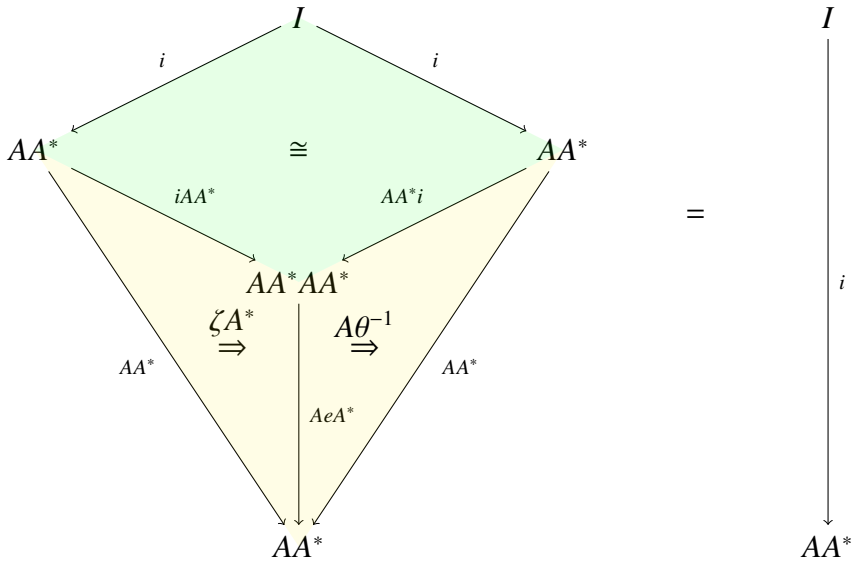
$$\zeta_A : A \Rightarrow (Ae_A) \circ (i_A A)$$

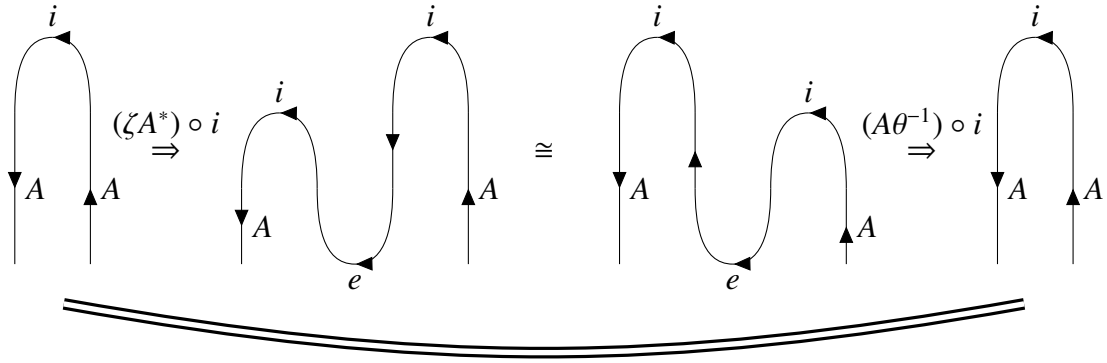


$$\theta_A : A^* \Rightarrow (e_A A^*) \circ (A^* i_A)$$

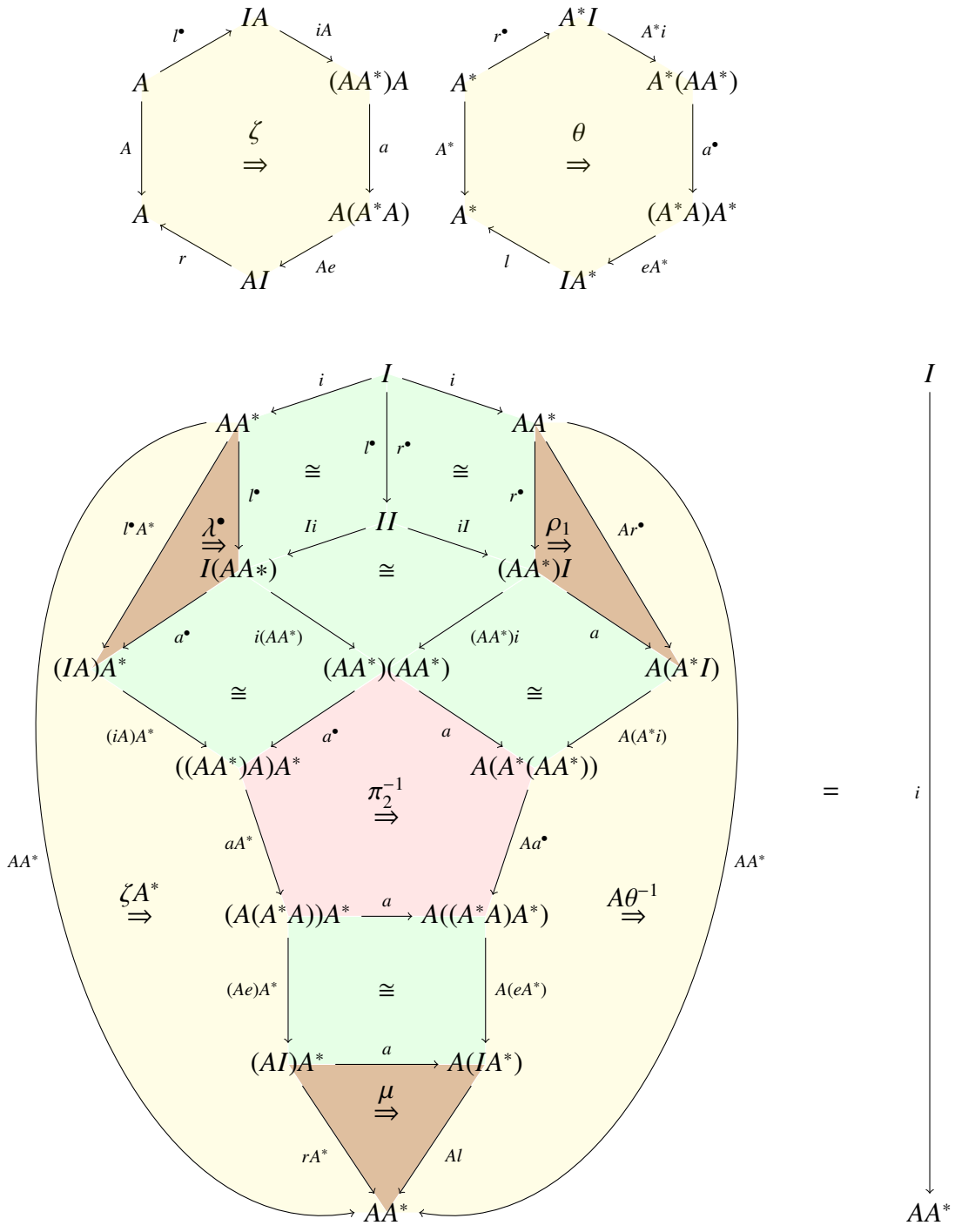


such that the following "swallowtail equation" holds:





We have drawn the diagrams in a strictly monoidal compact closed bicategory for clarity; when the associator is not the identity, we truncate some corners:



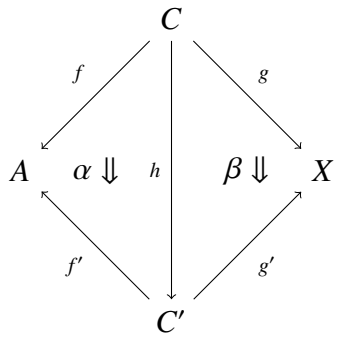
2.5 Bicategories of spans

At the start of this chapter, we stated that spans of sets form a compact closed bicategory. Street [189] suggested weakening the notion of a map of spans to hold only up to 2-isomorphism, al-

lowing to define spans in bicategories rather than mere categories; Hoffnung [99] worked out the details.

A **span** from A to B in a bicategory T is a pair of morphisms with the same source: $A \xleftarrow{f} C \xrightarrow{g} B$.

A **map of spans** h between two spans $A \xleftarrow{f} C \xrightarrow{g} B$ and $A \xleftarrow{f'} C' \xrightarrow{g'} B$ is a triple $(h\colon C \to C', \alpha\colon f \Rightarrow f'h, \beta\colon g \Rightarrow g'h)$ such that α and β are invertible.



A **map of maps of spans** is a 2-morphism $\gamma\colon h \Rightarrow h'$ such that $\alpha' = (f'\gamma) \cdot \alpha$ and $\beta' = (g'\gamma) \cdot \beta$. Maps and maps of maps compose in the obvious ways.

Hoffnung showed that any 2-category T with finite products and strict iso-comma objects (hereafter called "weak pullbacks") gives rise to a monoidal tricategory Span(T) whose

- objects are objects of T,
- morphisms are spans in T,
- 2-morphisms are maps of spans, and
- 3-morphisms are maps of maps of spans;

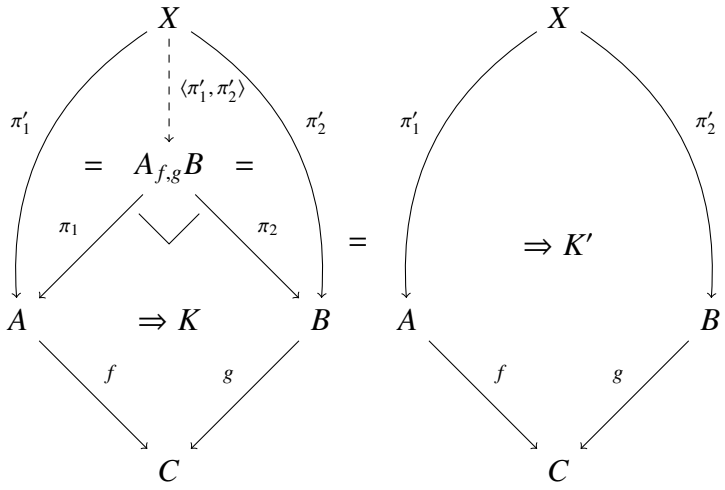
The tensor product of two spans $A \xleftarrow{f} C \xrightarrow{g} B$ and $A' \xleftarrow{f'} C' \xrightarrow{g'} B'$ is the span

$$A \times A' \xleftarrow{f \times f'} C \times C' \xrightarrow{g \times g'} B \times B'.$$

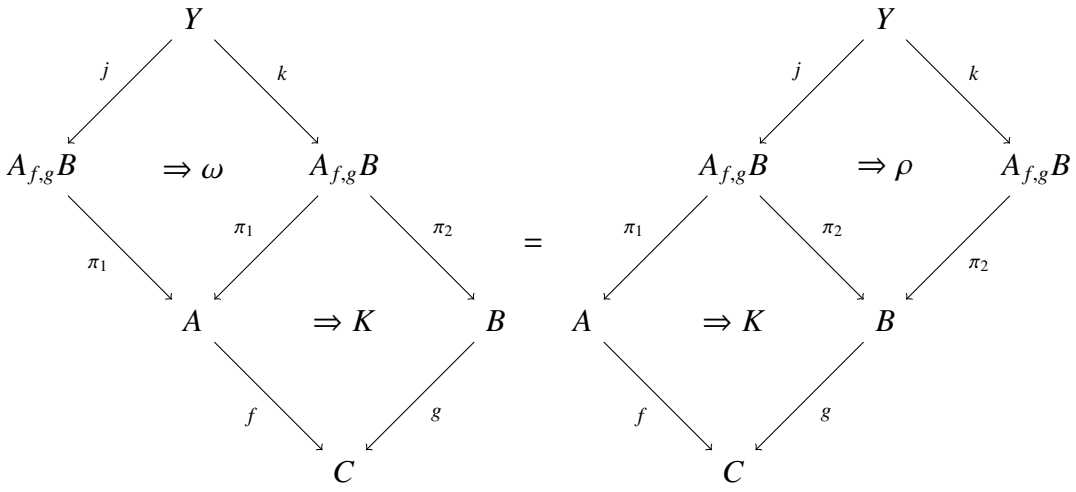
In this section, we will use A to mean the object A, the identity 1-morphism on A, or the identity 2-morphism on the identity 1-morphism on A, depending on the context. Similarly, we will use f to mean either the 1-morphism f or the identity 2-morphism on f, depending on the context. We will use juxtaposition to mean horizontal composition, *i.e.* composition of 1-morphisms: given $f\colon A \to B$ and $g\colon B \to C$, we get $gf\colon A \to C$. We also use juxtaposition to denote "whiskering": given a 2-morphism K, we denote by fK the the horizontal composition of K and the identity 2-morphism on f. We denote vertical composition of 2-morphisms $K\colon f \Rightarrow g$ and $L\colon g \Rightarrow h$ by $L \cdot K\colon f \Rightarrow h$. We use π_n to mean a projection out of a weak pullback, *not* a variant of the pentagonator 2-morphism.

We define composition of spans using the weak pullback in T. The weak pullback of a cospan $A \xrightarrow{f} C \xleftarrow{g} B$ consists of an object $A_{f,g}B$, 1-morphisms $\pi_1 \colon A_{f,g}B \to A$ and $\pi_2 \colon A_{f,g}B \to B$, and an invertible 2-morphism $K \colon f\pi_1 \Rightarrow g\pi_2$.

The weak pullback satisfies two universal properties. First, given any competitor $(X, \pi'_1 \colon X \to A, \pi'_2 \colon X \to B, K' \colon f\pi'_1 \Rightarrow g\pi'_2)$ where K' is invertible, there exists a unique 1-morphism $\langle \pi'_1, \pi'_2 \rangle \colon X \Rightarrow A_{f,g}B$ such that $\pi_1\langle \pi'_1, \pi'_2 \rangle = \pi'_1, \pi_2\langle \pi'_1, \pi'_2 \rangle = \pi'_2$, and $K\langle \pi'_1, \pi'_2 \rangle = K'$.



Second, given any object Y, 1-morphisms $j, k \colon Y \to A_{f,g}B$, and invertible 2-morphisms $\omega \colon \pi_1 j \Rightarrow \pi_1 k$ and $\rho \colon \pi_2 j \Rightarrow \pi_2 k$ such that



there is a unique 2-morphism $\gamma \colon j \Rightarrow k$ such that $\omega = \pi_1\gamma$ and $\rho = \pi_2\gamma$.

Here we show that the bicategory $\mathrm{Span}_2(T)$ whose

- objects are objects of T,

- morphisms are spans in T, and

- 2-morphisms are 3-isomorphism classes of maps of spans.

forms a compact closed bicategory.

Weak pullbacks are unique up to isomorphism [158]. The construction of Span(T) requires choosing specific weak pullbacks for each cospan [99, 3.2.1]; in our proof below, we choose especially nice pullbacks for the kinds of cospan that appear in the definition of a compact closed bicategory.

This raises the question of whether some choices are fundamentally different than others; however, Gurski and Osorno [91] proved that every symmetric monoidal bicategory is equivalent to a semistrict one. Schommer-Pries [171] strengthened their result by proving that every symmetric monoidal bicategory is equivalent to a quasistrict symmetric monoidal bicategory, and Bartlett [27] used Schommer-Pries' results to give semantics to a graphical calculus.

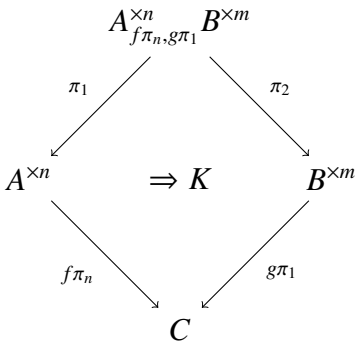
Every weak pullback of a cospan comes equipped with two projections out of it. Now suppose that we compose four identity spans on A, starting at the left; the resulting weak pullback is $((A_{A,A}A)_{\pi_2,A}A)_{\pi_2',A}A$, where $\pi_2 \colon A_{A,A}A \to A$ and $\pi_2' \colon (A_{A,A}A)_{\pi_2,A}A \to A$ are projections the weak pullbacks come equipped with. This notation clearly becomes very cumbersome very quickly—particularly when dealing with the composition of many spans, as we will below.

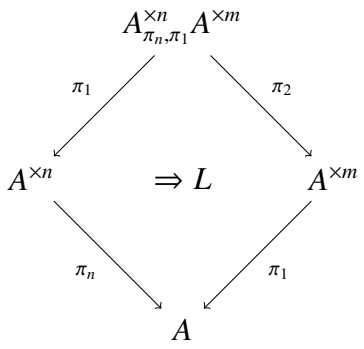
We introduce a new notation $A^{\times n}$ to mean the weak pullback in the composition of n identity spans on A, beginning at the left; that is, $A^{\times 1} = A, A^{\times 2} = A_{A,A}A$, and $A^{\times n} = A^{\times(n-1)}_{\pi_2,A}A$, where $\pi_2 \colon A^{\times(n-1)} \to A$ is the second projection that $A^{\times(n-1)}$ is equipped with. In Cat, for example, $A^{\times n}$ is a category whose objects are n-element lists of objects of A equipped with an isomorphism between every pair of consecutive elements.

These n-element lists are built out of pairs; in order to project onto the first element in the list using the notation developed so far, it would require us to choose the first element of each pair $(n-1)$ times, which is very cumbersome; effectively, we are forced to use unary to index the list. Instead, we will write π_1 through π_n for the n projections out of these pairs that result in an object of A. The construction $(-)^{\times n}$ is functorial; given a morphism $f \colon A \to B$, we get the pointwise application $f^{\times n} \colon A^{\times n} \to B^{\times n}$ of f to each object in the list.

Lemma 1 *Given isomorphisms $f \colon A \to C$ and $g \colon B \to C$, the weak pullback of the cospan $A^{\times n} \overset{f\pi_n}{\to} C \overset{g\pi_1}{\leftarrow} B^{\times m}$ is isomorphic to the weak pullback of the cospan $A^{\times n} \overset{\pi_n}{\to} A \overset{\pi_1}{\leftarrow} A^{\times m}$.*

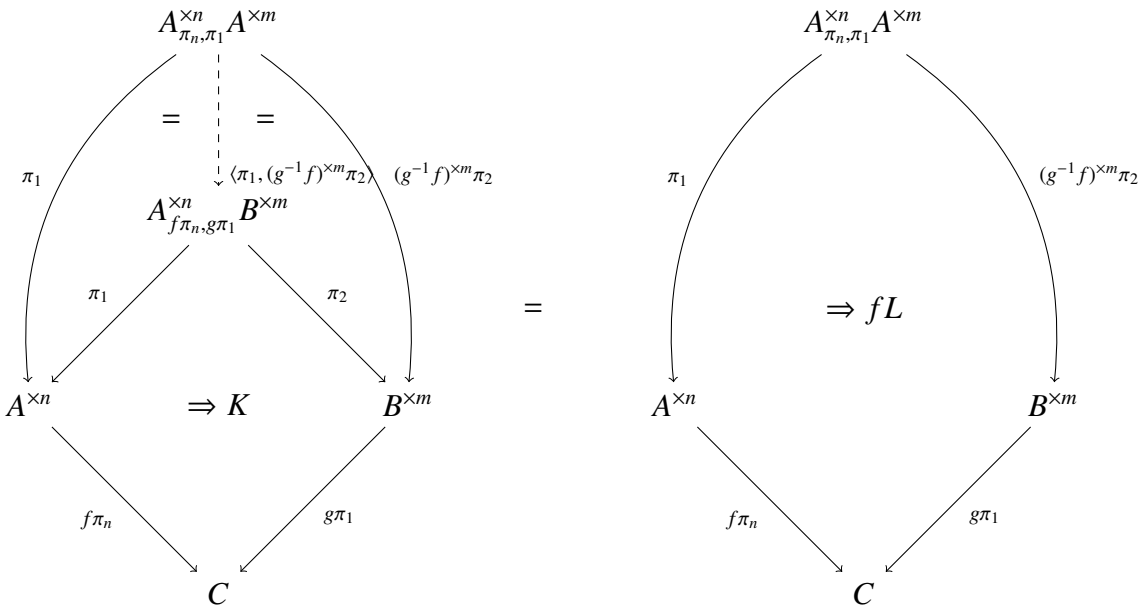
Proof. The weak pullbacks of the two cospans are

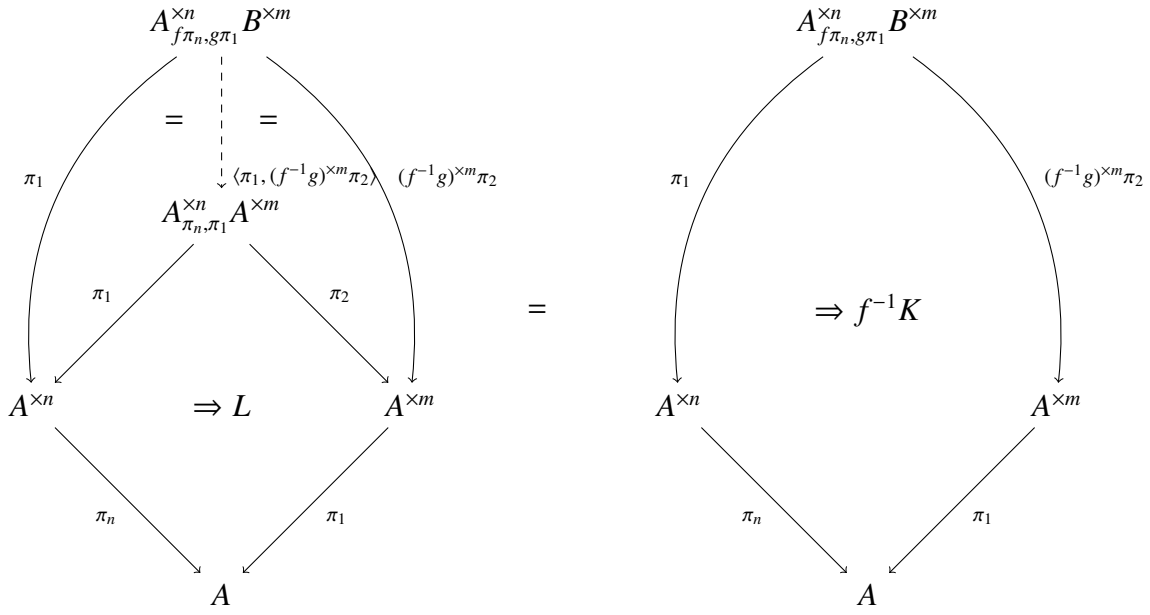




.

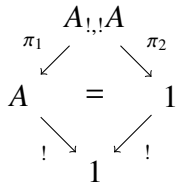
By the first universal property of weak pullbacks, there exist unique morphisms from $A^{\times n}_{\pi_n, \pi_1} A^{\times m}$ to $A^{\times n}_{f\pi_n, g\pi_1} B^{\times m}$ and back making the following diagrams commute. The unique morphisms are evidently inverses.



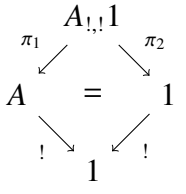


To see that the right-hand side in the first equation is really fL, consider the following: $A^{\times n}_{\pi_n,\pi_1} A^{\times m}$ is a pair of an n-element list and an m-element list. The projection π_2 picks out $A^{\times m}$, then $g^{-1}f$ is applied pointwise to each element. Next π_1 selects the first element of the list; finally, g is applied to it. Due to the functoriality of $(g^{-1}f)^{\times m}$, this is the same as picking out the first element of $A^{\times m}$ and then applying $gg^{-1}f = f$ to it. □

By the coherence theorem for bicategories, there is a unique isomorphism $a^{\circ *} : A^{\times n}_{\pi_n,\pi_1} A^{\times m} \to A^{\times(n+m)}$ built from associators for composition. Since we must choose weak pullbacks for each cospan, given a cospan $A^{\times n} \overset{f\pi_n}{\to} C \overset{g\pi_1}{\leftarrow} B^{\times m}$ where f and g are invertible, we choose the weak pullback to be equal to $A^{\times(n+m)}$. When A is terminal, for instance, A may not equal 1 but only be isomorphic. In that case, the weak pullback of $A \overset{!}{\to} 1 \overset{!}{\leftarrow} 1$ is



whereas when A is not terminal, the weak pullback is



.

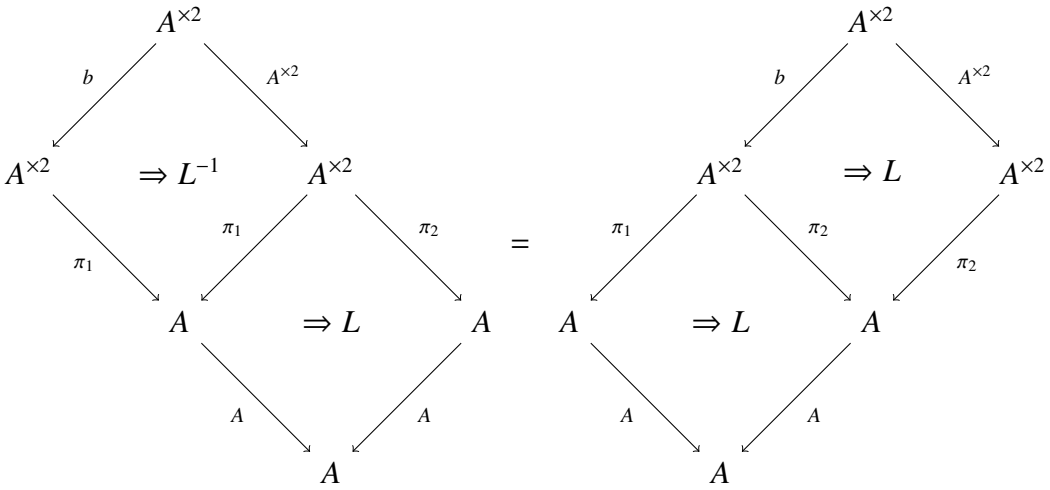
With that choice, we also have the following useful corollary.

Corollary 34 *Given an isomorphism $f : A \to B$ in T, the composite of the identity span on B and the span $B \overset{f}{\leftarrow} A \overset{A}{\rightarrow} A$ is equal to the composite of the identity span on B and the span $B \overset{B}{\leftarrow} B \overset{f^{-1}}{\rightarrow} A$; both result in the span $B \overset{\pi_1}{\leftarrow} B^{\times 2} \overset{f^{-1}\pi_2}{\rightarrow} A$.*

Because we mod out by isomorphisms of maps of spans, some spans that at first sight appear different are actually the same.

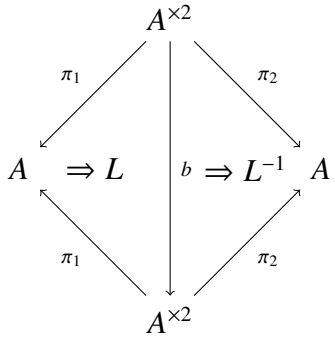
Lemma 2 *The braiding $b : A^{\times 2} \to A^{\times 2}$ in T is naturally isomorphic to the identity.*

Proof. The weak pullback of the identity cospan on A is $A^{\times 2}$ equipped with projections π_1, π_2 and a 2-morphism L. We have $\pi_1 b = \pi_2, \pi_2 b = \pi_1$, and $Lb = L^{-1}$. The following 2-morphisms are equal:



(note that on the right hand side, the lower use of L is whiskered by b, becoming L^{-1}), so by the second universal property of the weak pullback, there exists a unique 2-morphism $\gamma \colon b \Rightarrow A^{\times 2}$ such that $L^{-1} = \pi_1 \gamma$ and $L = \pi_2 \gamma$. □

Corollary 35 *The map of spans*



is in the same equivalence class as the identity map of spans.

Corollary 36 *For any permutation σ of n elements, the morphism $\langle \pi_{\sigma(1)}, \pi_{\sigma(2)}, \ldots, \pi_{\sigma(n)} \rangle \colon A^{\times n} \to A^{\times n}$ is naturally isomorphic to the identity.*

We are now ready to prove the main theorem.

Theorem 37 *If T is a 2-category with finite products and weak pullbacks, then $Span_2(T)$ is a compact closed bicategory.*

Proof. As noted, Hoffnung [99] showed that $\mathrm{Span}_3(T)$ is a monoidal tricategory. We refer the reader to Hoffnung's paper for the complete definition of a monoidal tricategory, but suffice it to say that it replaces the commuting polyhedra in the above definition of a monoidal bicategory with polyhedra that commute up to a specified 3-morphism, and then adds coherence law polytopes to govern them. When we mod out by 3-isomorphism classes of maps of spans, these 3-morphisms become trivial, so $\mathrm{Span}_2(T)$ is a monoidal bicategory.

The associator is the span

$$(A \times B) \times C \overset{(A\times B)\times C}{\leftarrow} (A \times B) \times C \overset{a}{\to} A \times (B \times C).$$

The left and right unitors are the spans

$$1 \times A \overset{1\times A}{\leftarrow} 1 \times A \overset{l}{\to} A$$

and

$$A \times 1 \overset{A\times 1}{\leftarrow} A \times 1 \overset{r}{\to} A,$$

respectively. The braiding is

$$A \times B \overset{A\times B}{\leftarrow} A \times B \overset{b}{\to} B \times A.$$

The cap is

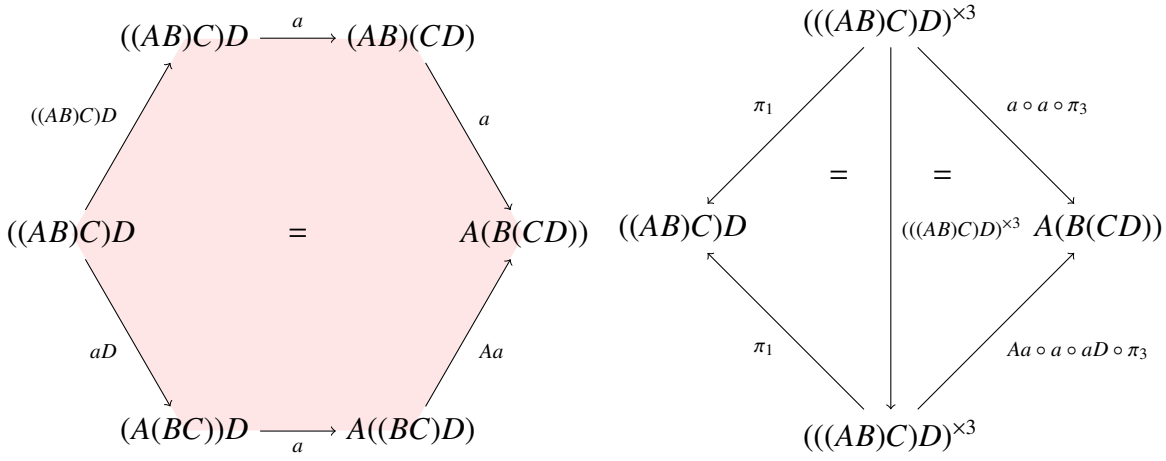
$$1 \overset{!}{\leftarrow} A \overset{\Delta}{\to} A \times A;$$

the cup is its reverse,

$$A \times A \overset{\Delta}{\leftarrow} A \overset{!}{\to} 1.$$

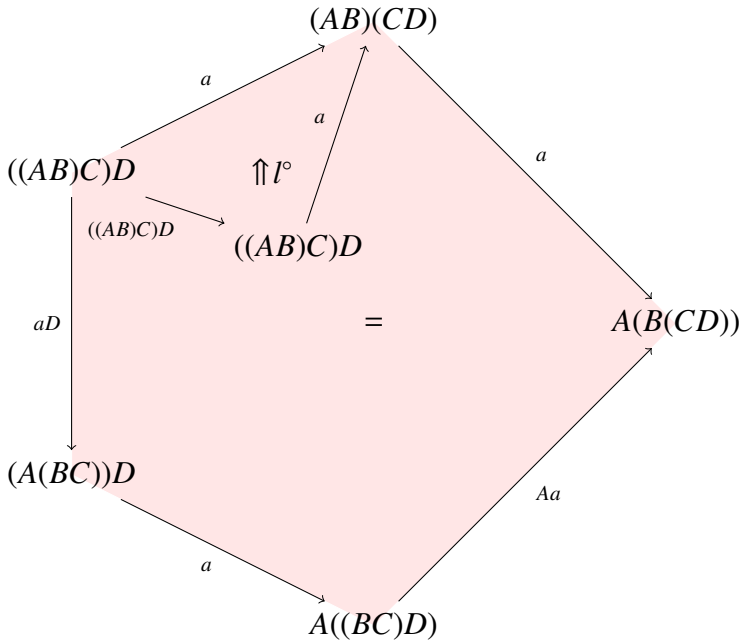
The "bulleted" morphisms like $a^\bullet$ are the reverse spans.

By adding an identity edge to the pentagonator, we get an identity map of spans: each edge is a span whose left leg is the identity and whose right leg is an isomorphism in T; the source and target composite spans are both the composition of three such edges, so by our choice of weak pullbacks and Lemma 1, their apexes are equal.

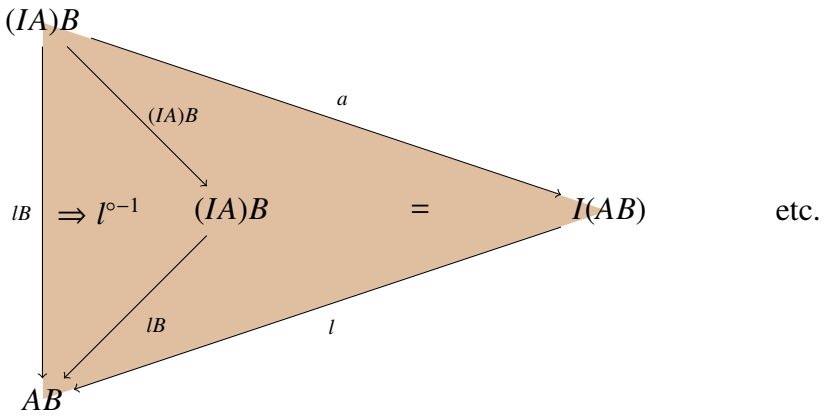


The right-hand 2-morphism in the map of spans is an identity because the pentagon equation holds in the underlying category of T. We recover a five-sided pentagonator by composing this identity map of spans with the unitor for composition.

The coherence theorem for bicategories [138] says that any diagram built out of a°, l°, and r° commutes, so any coherence law involving only pentagonators and identity 2-morphisms—like the associahedron—must hold in $\mathrm{Span}_2(T)$.



Similarly, the 2-unitors λ, μ, and ρ for the monoidal product are all equal to the inverse of the unitor for composition. Again, by the coherence theorem for bicategories, any coherence law involving only π, λ, μ, ρ and identity 2-morphisms—like the unitor prisms—must hold in $\mathrm{Span}_2(T)$.

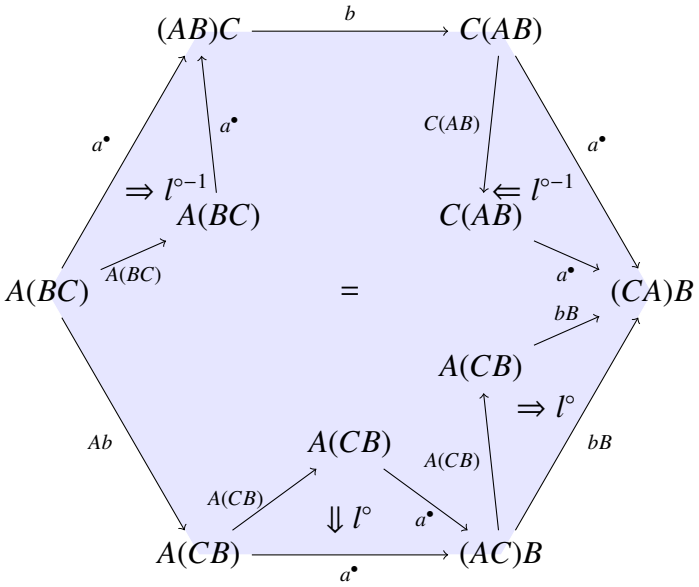


etc.

The hexagon modification R, like the "six-sided pentagonator" above, is an identity map of spans: each edge is, again, a span whose left leg is an identity and whose right leg is an isomorphism in T. The source and target spans are the composition of three such edges, so the apexes are equal; the right-hand 2-morphism in the map of spans is an equality because the hexagon equations hold in T.

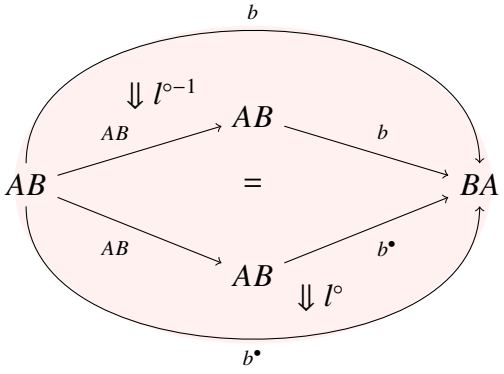
The hexagon modification S is more complicated because it has three uses of $a^\bullet$. If we add an identity edge before each one, then by Corollary 34 the apexes are the same as if we had used a^{-1} instead. To get the same number of edges in the source span as in the target span, we add another identity edge before the last edge in the target span. The resulting eight-sided 2-morphism is an identity. We recover a six-sided S by using two inverse unitors for composition on the source span and two unitors for composition on the target span.

By the coherence theorem for bicategories, any coherence law involving only π, R, S and identity 2-morphisms—like the shuffle and Breen polytopes—must hold in $\mathrm{Span}_2(T)$.



We can insert an identity morphism in the source and target of the syllepsis to get a four-sided iden-

tity map of spans, then use unitors for composition to recover a 2-sided syllepsis. By the coherence theorem for bicategories, any coherence law involving only R, S, ν and identity 2-morphisms—like those governing the syllepsis—must hold in $\mathrm{Span}_2(T)$.

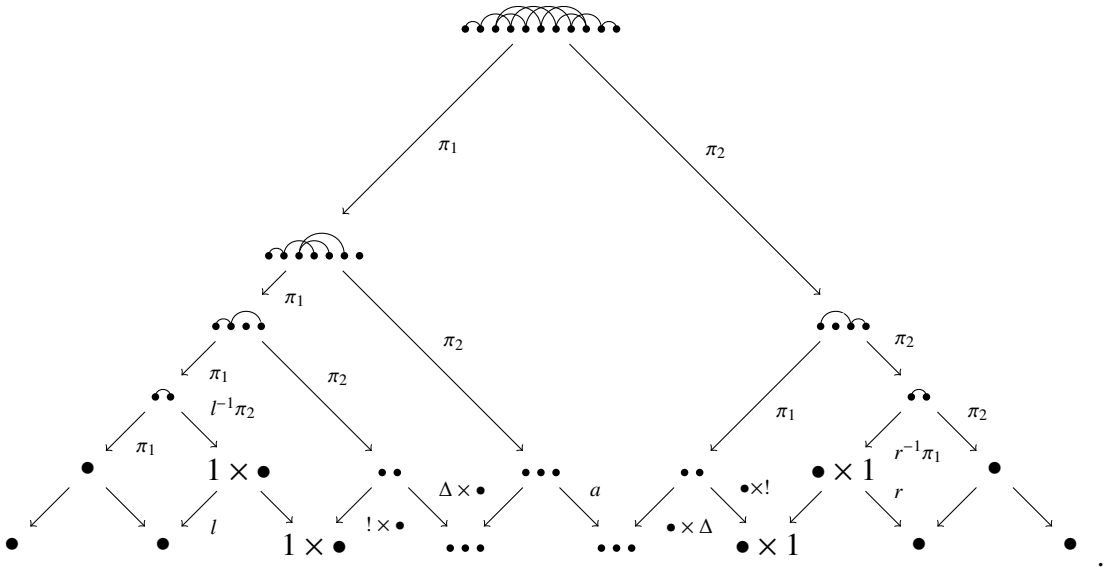


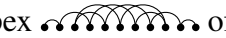
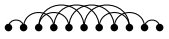
Because all these coherence laws hold in $\mathrm{Span}_2(T)$, it is a symmetric monoidal bicategory.

The swallowtail coherence law involves spans whose legs are not necessarily isomorphisms, so the approach taken above will not work to prove that the swallowtail coherence law holds. As a calculational aid, we introduce some topological notation for weak pullbacks. The notation is based on $\mathrm{Span}_2(\mathrm{Cat})$, where the weak pullback is a category. The objects of the weak pullback of the cospan $A \xrightarrow{f} C \xleftarrow{g} B$ in Cat are triples $(a, b, m\colon fa \leftrightarrow gb)$, so we will denote each object with a dot and an isomorphism between objects with an arc. The weak pullback of a cospan is denoted by a copy of the object on the left, a copy of the object on the right, and some new arcs between them. For example, the weak pullback $A^{\times 2} = A_{A,A}A$ will be denoted $\overset{\frown}{\bullet\bullet}$, the product $A \times A$ will be denoted $\bullet\ \bullet$ and the weak pullback of the cospan $A^{\times 2} \xrightarrow{\pi_2} A \xleftarrow{\pi_2} A \times A$ will be denoted $\overset{\frown}{\bullet}\overset{\frown}{\bullet\bullet}$.

The target 1-morphism of the yanking 2-morphism ζ_A is $r(A \otimes e)a(i \otimes A)l^\bullet$. In order to turn $l^\bullet$ into l^{-1}, we pre- and post-compose with an identity morphism.

When A is terminal, the source and target morphism of ζ_A are unique and ζ_A is trivial. When A is not terminal, the composite span is



Inspection of the composite span above shows that none of the cospans involving Δ are of the form $A^{\times n} \overset{f\pi_n}{\to} C \overset{g\pi_1}{\leftarrow} B$ where f and g are isomorphisms, so the choice of weak pullback for those cospans does not matter here. The apex  of this composition has eleven dots connected by ten arcs in a single chain. It is evident that  can be permuted to  , the apex of the composition of ten identity spans, so by Corollary 36, this composite span is equal to the composition of ten identity spans. With judicious use of unitors for composition we recover a six-sided 2-morphism ζ. The 2-morphism θ follows *mutatis mutandis*; by the coherence theorem for bicategories, any coherence law involving $\pi, \lambda, \mu, \rho, \zeta, \theta$ and identity 2-morphisms—like the swallowtail coherence law—must hold in $\mathrm{Span}_2(T)$. Therefore, $\mathrm{Span}_2(T)$ is compact closed. □

Corollary 38 *When C is a category with finite products and pullbacks, the bicategory Span(C) of objects of C, spans in C, and maps of spans is compact closed.*

Proof. When C is a category with finite products and pullbacks, $\mathrm{Span}(C)$ is a special case of Theorem 37 where all the 2-morphisms in the weak pullbacks are identities. □

Corollary 39 *The bicategories Cospan(ResNet) and Circ are compact closed.*

Proof. The coproduct of two resistor networks is given by juxtaposition; the pushout of a cospan $S \hookleftarrow R \hookrightarrow T$ of resistor networks is given by juxtaposition followed by identifying the images of R in S and T. Cospans in ResNet are spans in $\mathrm{ResNet}^{\mathrm{op}}$, where the coproduct and pushout become product and pullback, so Cospan(ResNet) is compact closed by the previous corollary. Since every object is self-dual in Cospan(ResNet), the subcategory Circ whose objects are resistor networks with no edges is also compact closed. □

2.6 Conclusion

Compact closed bicategories are symmetric monoidal bicategories in which each object has a pseduoadjoint. Given a 2-category T with finite products and iso-comma objects, the bicategory $\mathrm{Span}_2(T)$ of objects of T, spans in T, and isomorphism equivalence classes of maps of spans forms a compact closed bicategory. As a corollary, the bicategories Cospan(ResNet) and Circ are compact closed.

In the next chapter, we examine potential uses for compact closed bicategories in a generalization of universal algebra, in quantum physics, and in semantics for concurrent programming languages.

Chapter 3

Future Work

There is a deep relationship between logic, type theory, topology, and physics that centers on the notions of state and process. It lets us take concepts like the notion of a partition function from physics and transport it over to computer science to reason about the statistical properties of collections of programs. It lets us assign quantum meaning to drawings of particles' worldlines. It lets us use programming languages to construct proofs, and use types to prove that our programs are well-behaved. By generalizing this relationship to add another dimension, I believe we will reap several benefits beyond those we currently enjoy; here I describe three ideas for applications and future work.

3.1 Generalizing object-oriented programming

I began studying compact closed bicategories in order to write down "the higher theory of a symmetric monoidal closed category". To explain what I mean, I think it would be instructive to go down one dimension first and look at "universal algebra".

Many concepts in mathematics can be expressed as sets equipped with functions that satisfy relations. For example, a group is

- a set G equipped with
- a function $mult\colon G \times G \to G$,
- a function $id\colon 1 \to G$, and
- a function $inv\colon G \to G$ such that
- for all a, b, c in G, $mult(mult(a, b), c) = mult(a, mult(b, c))$,
- for all a in G, $mult(a, id()) = a = mult(id(), a)$, and
- for all a in G, $mult(a, inv(a)) = mult(inv(a), a) = id()$.

Pointed sets, monoids, groups, rings, directed graphs, and more all have presentations of this form. We can abstract away the sets and functions to objects and morphisms to get the definition of a "group object":

- an object G equipped with
- a morphism $mult\colon G \times G \to G$,
- a morphism $id\colon 1 \to G$, and
- a morphism $inv\colon G \to G$ such that
- $mult \circ (mult \times G) = mult \circ (G \times mult)$,
- $mult \circ (id \times G) \circ left^{-1} = mult \circ (G \times id) \circ right^{-1} = G$, and
- $mult \circ (inv \times G) \circ \Delta = mult \circ (G \times inv) \circ \Delta = id$,

where $left\colon G \to 1 \times G$, $right\colon G \to G \times 1$, and $\Delta\colon G \to G \times G$ are the unitors and diagonal morphism in any category with products. Bill Lawvere showed in his 1963 thesis [136] that this way of writing the definition of a group is a presentation of a particular category with products (now called a "Lawvere theory" in his honor), that any product-preserving functor from this category to the category of sets and functions picks out a group, and that any group can be expressed as such a functor. Under certain circumstances, these functors always have adjoints, which means that it makes sense to talk about the free group on a set and the underlying set of a group. Another benefit is that one is free to consider group objects in categories other than Set. For example, a group object in the category of manifolds and smooth maps is a Lie group [159].

In the years since Lawvere, other kinds of theory have been proposed; Lawvere's theories assumed the existence of cartesian products, but only a tensor product is needed to write down the notion of a monoid. The category Vect of vector spaces and linear maps has a tensor product as well as the cartesian product; a monoid in (Vect, $\times$) is just a vector space, but a monoid in (Vect, $\otimes$) is an algebra [159]. While Lawvere theories have models in categories with products, PROs have models in monoidal categories and PROPs have models in symmetric monoidal categories. The most general theory of a monoid is a PRO: we can, for instance, consider monoid objects in a category of endofunctors and natural transformations, where the tensor product is composition; the resulting concept is a monad.

In Section 1.1.1 we discussed one special kind of Lawvere theory called a "typed lambda-theory", where in addition to the cartesian product we are able to talk about the internal hom. Lambda-theories play a central role in modern object-oriented programming; what category theorists call a theory, programmers call an interface, and what category theorists call a structure-preserving functor between theories, programmers call an implementation of the interface. Granted, most popular programming languages sacrifice correctness in the form of proof-bearing code for convenience in the form of tests, but the spirit is there; and as theorem provers become more powerful, languages like Agda that require proofs are becoming more widely used.

My intent in studying compact closed bicategories was not to describe a set with structure, but rather a category with structure and stuff: a symmetric monoidal closed category. I wanted to describe, as a programmer, what it meant to be a programming language in the general sense explained in Chapter II.1. The notion of a "higher" theory is an obvious extension of Lawvere's idea, but as far as I know, no one has worked out the details. A higher theory should be a bicategory of some kind, and its presentation should have objects, 1-morphisms, 2-morphisms, and equations among the 2-morphisms. Of course, if our theories are bicategories, we should take our models in bicategories, too. The first 2-category that comes to mind is Cat, as a categorified version of Set; however, the higher theory of a symmetric monoidal closed category needs to be able to describe the adjunction between $- \otimes X$ and $X \multimap -$. Since all our objects are small categories, this adjunction is most easily expressed as an isomorphism of hom sets, which indicates that the category Set should play a special role.

Prof is the bicategory of small categories, profunctors, and natural transformations. A profunctor $F\colon C \to D$ is a functor $F\colon C \to \mathrm{Set}^{D^{\mathrm{op}}}$. One can turn any functor $G\colon R \to S$ into a profunctor like this:

$$\begin{aligned} G\colon R &\to \mathrm{Set}^{S^{\mathrm{op}}} \\ r &\mapsto s \mapsto \mathrm{Hom}(s, G(r)). \end{aligned}$$

An object in $\mathrm{Set}^{C^{\mathrm{op}}}$ is called "a presheaf on C". Composing profunctors is very much like multiplying matrices, in that there is a "summation" (really a coend) over the middle category; composition is not associative, but there is an associator. If we restrict Prof to Cauchy-complete categories, then Prof is equivalent (as a mere category) to Cocont, the 2-category of presheaf categories, cocontinuous (*i.e.* small-colimit-preserving) functors, and natural transformations [113, 2.7.4]. So we can "strictify" composition if we'd like, and ignore the associator and unitors for composition. Also, if we restrict to Cauchy-complete categories in Prof, then we also have the nice property that a profunctor has a right adjoint if and only if it comes from a functor as described above [119, Theorem 4.51].

Prof is a compact closed bicategory. The monoidal product is the cartesian product in Cat and the monoidal unit is the category with a single object and only the identity morphism. Below, I denote the associator, symmetric braiding, currying, and unitor natural isomorphisms as **assoc**, **swap**, **curry**, **left** and **right**, respectively. Every object C has a dual $C^* = C^{\mathrm{op}}$. The internal hom $A \multimap B$ is equivalent to $A^{\mathrm{op}} \times B$. This means that through judicious use of currying and unitors we can move categories from one side of the arrow to the other:

$$\begin{aligned} F &: C \nrightarrow D \\ \mathbf{right}^{-1} \circ F &: C \nrightarrow D \times I \equiv D^{\mathrm{op}} \multimap I \\ \mathbf{curry}^{-1}(\mathbf{right}^{-1} \circ F) &: C \times D^{\mathrm{op}} \nrightarrow I \\ \mathbf{curry}^{-1}(\mathbf{right}^{-1} \circ F) \circ \mathbf{swap} &: D^{\mathrm{op}} \times C \nrightarrow I \\ \mathbf{curry}(\mathbf{curry}^{-1}(\mathbf{right}^{-1} \circ F) \circ \mathbf{swap}) &: D^{\mathrm{op}} \nrightarrow C \multimap I \equiv C^{\mathrm{op}} \times I \\ F^* = \mathbf{right} \circ \mathbf{curry}(\mathbf{curry}^{-1}(\mathbf{right}^{-1} \circ F) \circ \mathbf{swap}) &: D^{\mathrm{op}} \nrightarrow C^{\mathrm{op}} \end{aligned}$$

For profunctors arising from functors, we have

$$\begin{aligned} F'\colon D &\nrightarrow C \\ d &\mapsto c \mapsto \mathrm{Hom}(F(c), d) \end{aligned}$$

The theory given below should also be a compact closed 2-category. The monoidal product is denoted $\boxtimes$, the unit is J, and the dual is *. I believe that 2-functors preserving the compact closed structure pick out symmetric monoidal closed categories; I hope that the forgetful "underlying category" 2-functor has a left adjoint. If it does, then we can say "the free symmetric monoidal closed category on a category" and form a 2-monad (or pseudomonad) whose algebras (or pseudoalgebras) are all symmetric monoidal closed categories.

This is my guess as to what the higher theory of a symmetric monoidal closed category looks like:

- an object C
- morphisms
  - $\otimes\colon C \boxtimes C \to C$
  - $\otimes'\colon C \to C \boxtimes C$
  - $I\colon J \to C$
  - $I'\colon C \to J$
  - $\multimap\colon C^* \boxtimes C \to C$
  - $\multimap'\colon C \to C^* \boxtimes C$
- 2-morphisms
  - $i^{\otimes}\colon 1_{C\boxtimes C} \Rightarrow \otimes' \circ \otimes$
  - $e^{\otimes}\colon \otimes \circ \otimes' \Rightarrow 1_C$
  - $i^I\colon 1_J \Rightarrow I' \circ I$
  - $e^I\colon I \circ I' \Rightarrow 1_C$
  - $i^{\multimap}\colon 1_{C^*\boxtimes C} \Rightarrow \multimap' \circ \multimap$
  - $e^{\multimap}\colon \multimap \circ \multimap' \Rightarrow 1_C$
  - $a\colon \otimes \circ (\otimes \boxtimes 1_C) \tilde{\Rightarrow} \otimes \circ (1_C \boxtimes \otimes) \circ \mathbf{assoc}_{C,C,C}$
  - $b\colon \otimes \tilde{\Rightarrow} \otimes \circ \mathbf{swap}_{C,C}$
  - $c : \otimes' \tilde{\Rightarrow} \mathbf{curry}(\multimap)$
  - $l\colon \otimes \circ (I \boxtimes 1_C) \tilde{\Rightarrow} \mathbf{left}_C$
  - $r\colon \otimes \circ (1_C \boxtimes I) \tilde{\Rightarrow} \mathbf{right}_C$
  - inverses for a, b, c, l, r
- equalities between 2-morphisms
  - pentagon equation

- triangle equation
- hexagon equations
- zigzag equations for each of $\otimes, I, \multimap$
- a, b, c, l, r composed with their inverses equal the identity

It has one generating object, C, so it makes sense to talk about the underlying category. We want to be sure that $\otimes, I$, and $\multimap$ are really functors, not arbitrary profunctors, so the first three pairs of 2-morphisms specify the adjoint relation between each of those morphisms and their prime. The next five 2-morphisms are the natural isomorphisms in the definition of a symmetric monoidal closed category; that they are isomorphisms is enforced when we get to the equalities between 2-morphisms.

The currying natural isomorphism is the really interesting bit, since it is the only one that must mention the external hom. Start with the tensor

$$\begin{aligned} \otimes\colon C \times C &\not\to C \\ (a, b) &\mapsto c \mapsto \mathrm{Hom}(c, a \otimes b), \end{aligned}$$

then take its right adjoint to get

$$\begin{aligned} \otimes'\colon C &\not\to C \otimes C \\ c &\mapsto (a, b) \mapsto \mathrm{Hom}(a \otimes b, c), \end{aligned}$$

which is the first half of the currying isomorphism.

Now start with the internal hom:

$$\begin{aligned} \multimap\colon C^{op} \times C &\not\to C \\ (b, c) &\mapsto a \mapsto \mathrm{Hom}(a, b \multimap c), \end{aligned}$$

Curry:

$$\begin{aligned} \mathbf{curry}(\multimap)\colon C &\not\to C \otimes C \\ c &\mapsto (a, b) \mapsto \mathrm{Hom}(a, b \multimap c), \end{aligned}$$

which is the second half of the currying isomorphism. Explicitly,

$$c : \otimes' \tilde{\Rightarrow} \mathbf{curry}(\multimap).$$

The equalities of 2-morphisms are mostly the coherence laws from the definition of a symmetric monoidal closed category, though we do need to add equalitites $aa^{-1} = 1, a^{-1}a = 1$, etc. to make a, b, c, l, r into natural *iso*morphisms.

The upshot is that compact closed bicategories seem like the natural language to express what it means to be monoidal, symmetric, and closed. For instance, if we take models in the 2-category Rel of sets, relations, and implications, we get abelian groups: the tensor product becomes addition (which is "symmetric", *i.e.* commutative), the dual becomes negation, and currying becomes a bi-implication

$$x + y = z \quad \Longleftrightarrow \quad x = -y + z.$$

3.2 Categorical models of concurrency

Seely [174] suggested that the process of computation in the lambda calculus should be modeled by rewrite rules; Hilken [96] later did this explicitly, and Hirschowitz [98] categorified the whole categorical framework for talking about lambda calculus. However, none of these approaches moved beyond "confluence". An *evaluation strategy* assigns to each term a rewrite that tells how to get to the next term. The Church-Rosser theorem says that the lambda calculus is *confluent*: while an evaluation strategy may never reach an answer, if two of them do, they will agree on the answer. It is precisely because it does not matter which terminating strategy we use that we can mod out by the rewrite 2-morphisms to get a cartesian closed category. This confluence means that lambda calculus is a fundamentally serial approach to programming.

Modern programming, however, deals with many computers all operating on different clocks, communicating over networks with delays, and the order in which messages arrive is important: consider two people racing to get the last concert ticket, or the order in which a deposit and a withdrawal apply to an empty bank account. If rewrites involve moving messages around, then they will certainly not be confluent.

Milner, Parrow, and Walker proposed the pi calculus in 1989 [153]. The pi calculus is to concurrent programming what the lambda calculus is to serial programming, and like lambda calculus, pi calculus has many many variants. Here's a linear fragment of the pi calculus that's common to several variants:

$$\begin{array}{rll} P, Q ::= & 0 & \text{do nothing} \\ | & P|Q & \text{concurrency} \\ | & x?(y_1, \ldots, y_n).P & \text{receive} \\ | & x!(y_1, \ldots, y_n) & \text{send} \end{array}$$

where x and y_i are taken from a countably infinite set of "names". Structural congruence says that concurrency and do-nothing form a commutative monoid. Reduction in the pi calculus is given by the comm rule, where $x?(y_1, \ldots, y_n).P \mid x!(z_1, \ldots, z_n)$ reduces to $P\{z_i/y_i\}$. The comm rule does not apply under a prefix, only to terms at the top level.

Jamie Vicary pointed out to me that if we relax the restriction on the comm rule so that it can apply anywhere, then we can model the fragment above in a compact closed bicategory with a strict monoid object M (where the unit and multiplication have right adjoints as above) and a family of adjunctions for each (name, object) pair. We model processes as 1-morphisms into M:

$$\begin{array}{rl} 0: & J \to M \\ |: & M \boxtimes M \to M \\ ?_{x,T}: & T^* \boxtimes M \to J \\ !_{x,T}: & J \to T^* \boxtimes M \end{array}$$

0 and | are the unit and multiplication in M, respectively. $!_{x,T}$ and $?_{x,T}$ are adjoints; the comm rule maps to the counit of the adjunction. Sending is tensoring $!_{x,T}$ with the value of type T to send, then composing with the counit for T. Receiving is the unit for T, the continuation process, the right adjoint of multiplication, and finally $?_{x,T}$.

All popular programming languages—from "eager" ones like Java, C, JavaScript, Perl, Python,

and Lisp, to "lazy" ones like Miranda, Lispkit, Lazy ML, Clean, and Haskell—do not reduce terms under a lambda abstraction: the JavaScript program

```
function loop() { while(1){} };
```

does not cause the web browser to lock up unless the function is invoked. If we model beta reduction as a 2-morphism like Seely suggested, it seems reasonable to try to model lambda calculus using a cartesian closed 2-category; however, the currying isomorphism

$$\hom(A \times B, C) \cong \hom(A, B \to C)$$

implies that if a term M rewrites to M' via beta reduction in the context $\Gamma, x : X$, then it is necessarily true that in the context Γ, the term $\lambda x.M$ must rewrite to $\lambda x.M'$ under beta—but that is forbidden in the languages listed above. If we mod out by beta and look for semantics in Set, then there is no real problem—but when we switch from lambda calculus to pi calculus, it breaks everything. Even if a pi calculus term M rewrites to M' under the comm rule, it *must not* be possible for the term $x(y).M$ to rewrite to $x(y).M'$: the semantics of the term $x(y).M$ *requires* a message to be received on the channel x before any progress is made on reducing M.

Restricting the comm rule to specific reduction contexts is not hard, but the resulting theory is not quite as pretty; it involves introducing a 1-morphism $C\colon M \to M$ to mark the reduction context and weakening the adjunction between $?_{x,T}$ and $!_{x,T}$ to a mere 2-morphism

$$comm\colon (T^* \boxtimes C) \circ (T^* \boxtimes |) \circ (!_{x,T} \boxtimes M) \circ (?_{x,T} \boxtimes M) \circ (T^* \boxtimes |') \Rightarrow T^* \boxtimes C.$$

Greg Meredith and I [3] used these ideas to prove full abstraction for the asynchronous polyadic pi calculus using a symmetric monoidal closed 2-category. There is much more work to be done along these lines, particularly with respect to the categorical semantics of Caires' behavioral / spatial types [42].

3.3 Principle of least action

In Chapter II.1, we made an analogy between programs and Feynman diagrams. The analogy is precise as far as it goes, but it is unsatisfactory in the sense that Feynman diagrams describe processes happening over time, while Lambek and Scott mod out by the process of computation that occurs over time. If we use 2-categories that explicitly model rewrites between terms, we get something that could potentially be interpreted in $3\mathrm{Cob}_2$. In an extended TQFT, there ought to be three notions of partition function: one for each direction of 1-morphism and another for 2-morphisms. Horizontal morphisms in $3\mathrm{Cob}_2$ are space-like; a program has a length. Vertical morphisms are time-like; a rewrite occurs over time. The partition function for horizontal morphisms is concerned with algorithmic information, the shortest program that computes a result. The partition function for vertical morphisms should be concerned with the fastest way to compute a result. The partition function for 2-morphisms should have something to do with getting the best time-space tradeoff.

The analogy above used a double category, but we might also consider bicategories. In Hamiltonian mechanics, we think of the state of the system as having a position and a momentum; the position

describes a particular arrangement of parts of the system and the momentum describes how that arrangement is changing. It is my contention that position and momentum are analogous to a term and a rewrite. Evaluation strategies describe systems in which the momentum depends on the position; a more general notion of evaluation strategy would express both in terms of some other parameter like time. From a Lagrangian viewpoint, we would get a notion of stationary action; if each rewrite came with an associated computational cost, something like the variational principle would need to be used to minimize the cost of a computation. Indeed, dynamic programming is a standard technique in computer science for minimizing the cost of a computation; the canonical example is choosing an association of matrices that minimizes the number of multiplications to perform.

Conclusion

When we think about "stuff happening", we are naturally led to a consideration of states and processes, which form symmetric monoidal closed categories. Looking for states and processes in logic, topology, physics, and computer science gives us propositions and proofs, strings and nodes, particles and interactions, and types and programs, respectively, which is a big generalization of the Curry-Howard isomorphism. By summing over states or processes, we get partition functions, where entropy plays a major role and we can do thermodynamics—including with programs instead of particles. When summing over halting programs, even the bits of the value of the partition function have entropy.

The analogy between physical processes and programs is not quite as nice as we would like, but if we add one dimension to get symmetric monoidal closed *bi*categories, there is promise for an even deeper analogy between type theory and physics with potential applications in resistor networks, circuits, programming language design, the semantics of concurrency, and a better understanding of spacetime.

Bibliography

[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 at 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] S. Abramsky and N. Tzevelokos, Introduction to categories and categorical logic, in this volume. Also available at http://web.comlab.ox.ac.uk/people/Bob.Coecke/AbrNikos.pdf.

[7] V. A. Adamyan, C. S. Calude, B. S. Pavlov. Transcending the limits of Turing computability, in T. Hida, K. Saitô, S. Si (ed.). *Quantum Information Complexity*. Proceedings of Meijo Winter School 2003, World Scientific, Singapore, 2004, 119–137.

[8] S. Ambler, First order logic in symmetric monoidal closed categories, Ph.D. thesis, U. of Edinburgh, 1991. Available at http://www.lfcs.inf.ed.ac.uk/reports/92/ECS-LFCS-92-194/ .

[9] S. O. Anderson and A. J. Power, A representable approach to finite nondeterminism, Proc. MFPS '94, *Theoretical Computer Science* (1997) 3–25.

[10] K. Asada, Arrows are strong monads, *MSFP '10 Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming*. (2002), 33–42. Also available at www.ipl.t.u-tokyo.ac.jp/~asada/papers/arrStrMnd.pdf.

[11] R. Atkey, What is a Categorical Model of Arrows? *Electronic Notes in Theoretical Computer Science* **229** (5), 19–37. Also available at http://www.sciencedirect.com/science/article/pii/S157106611100051X.

[12] M. Atiyah, Topological quantum field theories, *Publications Mathématiques de l'Institut des Hautes Études Scientifiques.* **68:1** (1988), 175–186.

[13] 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.

[14] J. Baez, Classical versus quantum computation, U. C. Riverside seminar notes by D. Wise. Available at http://math.ucr.edu/home/baez/qg-fall2006/ and http://math.ucr.edu/home/baez/qg-winter2007.

[15] J. Baez, Groupoidification. http://math.ucr.edu/home/baez/groupoidification/

[16] 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 at arXiv:gr-qc/9902017.

[17] 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 at arXiv:quant-ph/0404040.

[18] J. Baez and J. Dolan, Higher-dimensional algebra and topological quantum field theory, *J. Math. Phys.* **36** (1995), 6073–6105. Also available at q-alg/9503002v2.

[19] J. Baez and J. Dolan, Higher-dimensional algebra III: n-categories and the algebra of opetopes, *Adv. Math.* **135** (1998), 145–206. Also available at q-alg/9702014.

[20] J. Baez and L. Langford, Higher-dimensional algebra IV: 2-Tangles, *Adv. Math.* **180** (2003), 705–764. Also available at math.QA/9811139.

[21] 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.

[22] J. Baez and M. Neuchl, Higher-dimensional algebra I: Braided monoidal 2-categories, *Adv. Math.* **121** (1996), 196–244. Also available at q-alg/9511013.

[23] J. Baez, D. Wise and A. Crans, Exotic Statistics for Strings in 4d BF Theory, *Adv. Theor. Math. Phys.* **11** (2007), 707–749. Also available at arXiv:gr-qc/0603085.

[24] 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.

[25] H. Barendregt, *The Lambda Calculus, its Syntax and Semantics*, North–Holland, Amsterdam, 1984.

[26] 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.

[27] B. Bartlett, Quasistrict symmetric monoidal 2-categories via wire diagrams. Available at arXiv:1409.2148.

[28] M. A. Batanin, The Eckman-Hilton argument and higher operads. Available at arXiv:math/0207281.

[29] J. S. Bell, On the Einstein-Podolsky-Rosen paradox, *Physics* **1** (1964), 195–200.

[30] J. L. Bell, *The Development of Categorical Logic*, available at http://publish.uwo.ca/~jbell/catlogprime.pdf.

[31] J. Bénabou, Distributors at work. Available at http://www.mathematik.tu-darmstadt.de/~streicher/FIBR/DiWo.pdf.

[32] J. Bénabou, Introduction to bicategories, in *Reports of the Midwest Category Seminar*, Springer, Berlin, (1967), 1–77.

[33] J. Bénabou, Les distributeurs, *Sem. Math. Pure Univ. Catholique de Louvain*, **33** (1973).

[34] C. H. Bennett, P. Gacs, M. Li, M. B. Vitányi and W. H. Zurek, Information distance, *IEEE Trans. Inform. Theor.* **44** (1998), 1407–1423.

[35] 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

[36] 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.

[37] G. Bierman, *On Intuitionistic Linear Logic*, PhD Thesis, Cambridge University. Available at http://research.microsoft.com/~gmb/Papers/thesis.pdf.

[38] 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 at http://www.site.uottawa.ca/~phil/papers/catsurv.web.pdf.

[39] F. Borceux and D. Dejean, Cauchy completion in category theory, *Cahiers Topologie Géom. Différentielle Catégoriques*, **27** (1986), 133–146.

[40] L. Breen, On the classification of 2-gerbes and 2-stacks, *Astérisque* **225** (1994), 1–160.

[41] S. N. Burris and H. P. Sankappanavar, *A Course in Universal Algebra*, Springer, Berlin, 1981. Also available at http://www.math.uwaterloo.ca/~snburris/htdocs/ualg.html.

[42] LĊaires, Logical Semantics of Types for Concurrency, *Proceedings of CALCO'07*, Springer-Verlag, Lecture Notes in Computer Science (2007). Also available at http://ctp.di.fct.unl.pt/~lcaires/papers/CALCO-Caires-1.0.pdf

[43] C. S. Calude, *Information and Randomness: An Algorithmic Perspective*, Springer, Berlin, 2002.

[44] C. S. Calude, Elena Calude, M. J. Dinneen. A new measure of the difficulty of problems, J. Mult.-Valued Logic Soft Comput. 12 (2006), 285–307.

[45] C. S. Calude, M. J. Dinneen and C.-K. Shu. Computing a glimpse of randomness, Experiment. Math. 11, 2 (2002), 369–378.

[46] C. S. Calude, H. Jürgensen. Is complexity a source of incompleteness? Adv. in Appl. Math. 35 (2005), 1–15.

[47] C. S. Calude, H. Jürgensen, S. Legg. Solving finitely refutable mathematical problems, in C. S. Calude, G. Păun (eds.). Finite Versus Infinite. Contributions to an Eternal Dilemma, Springer-Verlag, London, 2000, 39–52.

[48] C. S. Calude, B. Pavlov. Coins, quantum measurements, and Turing's barrier, Quantum Inf. Process. 1, 1–2 (2002), 107–127.

[49] C. S. Calude, L. Staiger, S. A. Terwijn, On partial randomness, *Ann. Appl. Pure Logic* **138** (2006) 20–30. Also available at http://www.cs.auckland.ac.nz/CDMTCS//researchreports/239cris.pdf.

[50] C. S. Calude, M. A. Stay. From Heisenberg to Gödel via Chaitin, Internat. J. Theoret. Phys. 44, 7 (2005), 1053–1065. Also available at arXiv:quant-ph/0402197.

[51] C. S. Calude and M. A. Stay, Most Programs Stop Quickly or Never Halt, *Adv. Appl. Math.* **40** (3), 295–308. Also available as arXiv:cs/0610153.

[52] C. S. Calude and M. A. Stay, Natural halting probabilities, partial randomness, and zeta functions, *Inform. and Comput.*, **204** (2006), 1718–1739.

[53] A. Carboni, G. M. Kelly, R. F. C. Walters, and R. J. Wood, Cartesian bicategories II, *Theory and Applications of Categories*, **19:6** (2008) 93–124. Available at http://www.emis.de/journals/TAC/volumes/19/6/19-06abs.html.

[54] A. Carboni and R. F. C. Walters, Cartesian bicategories I, *J. Pure Appl. Algebra* **49** (1987), 11–32.

[55] G. L. Cattani and G. Winskel, Profunctors, open maps and bisimulation, *Mathematical Structures in Computer Science*, **15:3** (2005) 553–614.

[56] G. Chaitin, A theory of program size formally identical to information theory, *Journal of the ACM* **22** (1975), 329–340. Also available at ⟨http://www.cs.auckland.ac.nz/~chaitin/acm75.pdf⟩.

[57] G. Chaitin, Algorithmic entropy of sets, *Comput. Math. Appl.* **2** (1976), 233–245. Also available at
⟨http://www.cs.auckland.ac.nz/CDMTCS/chaitin/sets.ps⟩.

[58] G. J. Chaitin. Computing the busy beaver function, in T. M. Cover, B. Gopinath (eds.). *Open Problems in Communication and Computation*, Springer-Verlag, Heidelberg, 1987, 108–112.

[59] V. Chari and A. Pressley, *A Guide to Quantum Groups*, Cambridge University Press, Cambridge, 1995.

[60] E. Cheng and A. Lauda, *Higher-Dimensional Categories: an Illustrated Guidebook*. Available at http://www.dpmms.cam.ac.uk/~elgc2/guidebook/.

[61] A. Church, An unsolvable problem of elementary number theory, *Amer. Jour. Math.* **58** (1936), 345–363.

[62] 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 at arXiv:quant-ph/0506134.

[63] B. Coecke, Kindergarten quantum mechanics, to appear in *Proceedings of QTRF-III*. Also available at arXiv:quant-ph/0510032.

[64] 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 (QPL 2006)*. Also available at arXiv:quant-ph/0608072.

[65] 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 at arXiv:quant-ph/0608035.

[66] S. Crans, Generalized centers of braided and sylleptic monoidal 2-categories, *Adv. Math.* **136** (1998), 183–223.

[67] R. L. Crole, *Categories for Types*, Cambridge U. Press, Cambridge, 1993.

[68] H. B. Curry and R. Feys, *Combinatory Logic*, Vol. I, North–Holland, Amsterdam, 1958.

H. B. Curry, J. R. Findley and J. P. Selding, *Combinatory Logic*, Vol. II, North–Holland, Amsterdam, 1972.

[69] P. Cvitanovic, *Group Theory*, Princeton U. Press, Princeton, 2003. Available at http://www.nbi.dk/GroupTheory/.

[70] G. D'Abramo. Asymptotic behavior and halting probability of Turing Machines, April 2006. Available at arXiv:math.HO/0512390.

[71] B. Day. On closed categories of functors. In Saunders Mac Lane, editor, Reports of the Midwest Category Seminar, *Lecture Notes in Mathematics*, **137** (1970) 1–38.

[72] B.J. Day and R. Street, Monoidal bicategories and Hopf algebroids, *Adv. Math.*, **129** (1997), 99-157.

[73] R. Di Cosmo and D. Miller, Linear logic, *Stanford Encyclopedia of Philosophy*, available at http://plato.stanford.edu/entries/logic-linear/

[74] M. Dorca and A. van Tonder, Quantum computation, categorical semantics and linear logic, available at arXiv:quant-ph/0312174.

[75] 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.

[76] S. Eilenberg and S. Mac Lane, General theory of natural equivalences, *Trans. Amer. Math. Soc.* **58** (1945), 231–294.

[77] M. Fiore, T. Leinster, Objects of Categories as Complex Numbers, *Adv. Math.* **190** (2005), 264–277. Available at arXiv:math/0212377.

[78] E. Fredkin and T. Toffoli, Conservative logic, *Intl. J. Theor. Phys.* **21** (1982), 219–253. Also available at ⟨http://strangepaths.com/wp-content/uploads/2007/11/conservativelogic.pdf⟩.

[79] M. Freedman, A. Kitaev, M. Larsen and Z. Wang, Topological quantum computation. Available at arXiv:quant-ph/0101025v2.

[80] M. Freedman, A. Kitaev and Z. Wang, Simulation of topological field theories by quantum computers, *Comm. Math. Phys.* **227** (2002), 587–603. Also available at 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 at arXiv:quant-ph/0001108.

[81] P. Freyd and D. Yetter, Braided compact monoidal categories with applications to low dimensional topology, *Adv. Math.* **77** (1989), 156–182.

[82] N. Gambino and J. Kock, Polynomial functors and polynomial monads, *Math. Proc. Cambridge Phil. Soc.* **154** (2013), 153–192. Also available at http://arxiv.org/abs/0906.4931arxiv:0906.4931.

[83] J.-Y. Girard, Linear logic, *Theor. Comp. Sci.* **50** (1987), 1–102. Also available at http://iml.univ-mrs.fr/ girard/linear.pdf.

[84] 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.

[85] K. Gödel, Zur intuitionistischen Arithmetik und Zahlentheorie, *Ergebnisse eines mathematischen Kolloquiums* **4** (1933), 34–38.

[86] 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.

[87] D. Gottesman and I. L. Chuang, Quantum teleportation is a universal computational primitive, *Nature* **402** (1999), 390–393. Also available at arXiv:quant-ph/9908010.

[88] J. Gray, *Formal category theory: Adjointness for 2-Categories,* Springer Lecture Notes in Mathematics **391**, Berlin, 1974.

[89] R. Gordon, A. J. Power, and R. Street, Coherence for tricategories, *Memoirs Amer. Math. Soc.* **117** (1995) Number 558.

[90] N. Gurski, An algebraic theory of tricategories, PhD thesis, University of Chicago, (2007). Available at http://gauss.math.yale.edu/~mg622/tricats.pdf

[91] N. Gurski and A. Osorno, Infinite loop spaces, and coherence for symmetric monoidal bicategories, Advances in Mathematics **246** (2013) 1–32. Also available at arXiv:1210.1174v1.

[92] J. D. Hamkins, A. Miasnikov. The halting problem is decidable on a set of asymptotic probability one, *Notre Dame J. Formal Logic* **47** (4) (2006), 515–524.

[93] 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.

[94] M. Hasegawa, On traced monoidal closed categories, *Mathematical Structures in Computer Science* **19** (2), (April 2009) 217–244.

[95] A. Heyting, ed., *L. E. J. Brouwer: Collected Works 1: Philosophy and Foundations of Mathematics*, Elsevier, Amsterdam, 1975.

[96] B. Hilken, Towards a proof theory of rewriting: the simply-typed 2λ-calculus, *Theor. Comp. Sci.* **170** (1996), 407–444. Also available at http://www.pps.univ-paris-diderot.fr/ mellies/mpri/mpri-ens/articles/hilken-2-lambda-calculus.pdf

[97] J. R. Hindley and J. P. Seldin, *Lambda-Calculus and Combinators: An Introduction*, Cambridge U. Press, Cambridge, 2008.

[98] Tom Hirschowitz, Cartesian closed 2-categories and permutation equivalence in higher-order rewriting, *Logical Methods in Computer Science*, IfCoLog (International Federation of Computational Logic), **9** (3) (2013), 10–28. Also available at https://hal.archives-ouvertes.fr/hal-00540205/file/macy.pdf

[99] A. Hoffnung, Spans in 2-categories: A monoidal tricategory. To appear. Also available at arXiv:1112.0560

[100] R. Houston, *Linear Logic without Units*, PhD Thesis, University of Manchester (2007). Available at arXiv:1305.2231

[101] 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.

[102] J. Hughes, Generalising Monads to Arrows, *Science of Computer Programming* **37** (May 2000) 67–111. Preprint available at http://www.cse.chalmers.se/~rjmh/Papers/arrows.pdf.

[103] B. Jacobs, C. Heunen and I. Hasuo, Categorical semantics for arrows, *Journal of Functional Programming* **19:3-4** (2009) 403–438. The results appeared in an earlier conference paper C. Heunen, B. Jacobs, Arrows, like Monads, are Monoids, *ENTCS* **158** (2006), 219–236.

[104] C. B. Jay, Languages for monoidal categories, *Jour. Pure Appl. Alg.* **59** (1989), 61–85.

[105] C. B. Jay, The structure of free closed categories, *Jour. Pure Appl. Alg.* **66** (1990), 271–285.

[106] E. T. Janyes, Information theory and statistical mechanics, *Phys. Rev.* **106** (1957), 620–630. Also available at ⟨http://bayes.wustl.edu/etj/articles/theory.1.pdf⟩.

[107] E. T. Janyes, *Probability Theory: The Logic of Science*, Cambridge U. Press, Cambridge, 2003. Draft available at ⟨http://omega.albany.edu:8008/JaynesBook.html⟩.

[108] 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.

[109] 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.

[110] A. Joyal, R. Street and D. Verity, Traced monoidal categories, *Math. Proc. of the Cambridge Phil. Soc.* **119:3** (1996) 447–468.

[111] D. Kaiser, *Drawing Theories Apart: The Dispersion of Feynman Diagrams in Postwar Physics*, U. Chicago Press, Chicago (2005).

[112] M. Kapranov and V. Voevodsky, 2-categories and Zamolodchikov tetrahedra equations, *Proc. Symp. Pure Math.* **56** (1994) 177–259.

[113] M. Kashiwara, P. Schapira, *Categories and Sheaves*, Grundlehren der Mathematischen Wissenschaften **332**, Springer (2006)

[114] C. Kassel, *Quantum Groups*, Springer, Berlin (1995).

[115] P. Katis, N. Sabadini, R.F.C. Walters, On partita doppia, *Theory and Applications of Categories* (Written in 1998, to appear). Available at arXiv:0803.2429v1.

[116] P. Katis, R.F.C. Walters, The compact closed bicategory of left adjoints, *Math. Proc. of the Cambridge Philosophical Society* **130:1** (2001) 77–87. Also available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.23.4825.

[117] L. H. Kauffman, *Knots and Physics*, World Scientific, Singapore (1991).

[118] L. H. Kauffman and S. Lins, *Temperley–Lieb Recoupling Theory and Invariants of 3-Manifolds*, Princeton U. Press, Princeton (1994).

[119] G. M. Kelly, *Basic Concepts of Enriched Category Theory*, Cambridge University Press, Lecture Notes in Mathematics **64** (1982). Also available at http://www.tac.mta.ca/tac/reprints/articles/10/tr10abs.html.

[120] G. M. Kelly, Many variable functorial calculus I, *Coherence in Categories, Lecture Notes in Math.*, **281** (1972) 66–105.

[121] G. M. Kelly, Basic concepts of enriched category theory, *Lecture Notes in Mathematics*, **64** (1982). Republished in *Reprints in Theory and Applications of Categories*, **10** (2005), 1–136. Available at http://www.tac.mta.ca/tac/reprints/articles/10/tr10abs.html.

[122] G. M. Kelly, M. L. Laplaza, Coherence for compact closed categories, *Journal of Pure and Applied Algebra* **19** (1980) 193–213.

[123] G. M. Kelly and S. Mac Lane, Coherence in closed categories, *Jour. Pure Appl. Alg.* **1** (1971), 97–140 and 219.

[124] S. Kleene, λ-definability and recursiveness, *Duke Math. Jour.* **2** (1936), 340–353.

[125] J. Kock, *Frobenius Algebras and 2D Topological Quantum Field Theories*, London Mathematical Society Student Texts **59**, Cambridge U. Press, Cambridge, 2004.

[126] A. Kock, Strong functors and monoidal monads, *Archiv der Math.* **23** (1972) 113–120. Available at http://home.imf.au.dk/kock/SFMM.pdf.

[127] S. Köhler, C. Schindelhauer, M. Ziegler. On approximating real-world halting problems, in M. Liśkiewicz, R. Reischuk (eds.). Proc. FCT 2005, Lectures Notes Comput. Sci. 3623, Springer, Heidelberg, 2005, 454–466.

[128] T. Kohno, ed., *New Developments in the Theory of Knots*, World Scientific, Singapore, 1990.

[129] A. N. Kolmogorov, Three approaches to the definition of the quantity of information, *Probl. Inf. Transm.* **1** (1965), 3–11.

[130] S. Kullback and R. A. Leibler, On information and sufficiency, *Ann. Math. Stat.* **22** (1951), 79–86.

[131] Stephen Lack, A 2-categories companion, in *Towards Higher Categories*, eds. J. Baez and P. May, Springer, Berlin, 2009. Also available at arXiv:math/0702535.

[132] 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.

[133] J. Lambek and P. J. Scott, *Introduction to Higher-order Categorical Logic*, Cambridge U. Press, Cambridge, 1986.

[134] P. Landin, A correspondence between ALGOL 60 and Church's lambda-notation, *Comm. ACM* **8** (1965), 89–101, 158–165.

[135] W. B. Langdon, R. Poli. The halting probability in Von Neuamnn architectures, in P. Collet et a. (eds.). EuroGP 2006, Lectures Notes Comput. Sci. 3905, Springer, Heidelberg, 2006, 225–237.

[136] 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.

[137] T. Leinster, A survey of definitions of n-category, *Th. Appl. Cat.* **10** (2002), 1–70. Also available at arXiv:math/0107188.

[138] T. Leinster, Basic bicategories. (1998) Available at http://arxiv.org/abs/math/9810017.

[139] L. A. Levin, Universal sequential search problems, *Probl. Inf. Transm.* **9** (1973), 265–266.

[140] L. A. Levin, Laws of information conservation (non-growth) and aspects of the foundation of probability theory. *Probl. Inf. Transm.* **10** (1974), 206–210.

[141] L. A. Levin and A. K. Zvonkin, The complexity of finite objects and the development of the concepts of information and randomness by means of the theory of algorithms, *Russian Mathematics Surveys* **256** (1970), 83–124 Also available at ⟨http://www.cs.bu.edu/fac/lnd/dvi/ZL-e.pdf⟩.

[142] M. Li and P. Vitányi, *An Introduction to Kolmogorov Complexity Theory and its Applications*, Springer, Berlin, 2008.

[143] S. Mac Lane, *Categories for the Working Mathematician*, Springer, Berlin, 1998.

[144] S. Mac Lane, Natural associativity and commutativity, *Rice Univ. Stud.* **49** (1963) 28–46.

[145] Y. Manin, *Renormalization and computation I: motivation and background.* Available as arxiv:0904.4921.

[146] Y. Manin, *Renormalization and computation II: Time Cut-off and the Halting Problem.* Available as arxiv:0908.3430.

[147] Y. Manin, M. Marcolli, *Error-correcting codes and phase transitions.* Available as arxiv:0910.5135.

[148] N. Margolus, L. B. Levitin. The maximum speed of dynamical evolution, Phys. D 120 (1998), 188–195.

[149] J. McCarthy, Recursive functions of symbolic expressions and their computation by machine, Part I, *Comm. ACM* **4** (1960), 184–195. Also available at http://www-formal.stanford.edu/jmc/recursive.html.

[150] P. McCrudden, Balanced coalgebroids, *Theory and Applications of Categories*, **7** 6 (2000), 71–147. Available at http://www.emis.de/journals/TAC/volumes/7/n6/7-06abs.html.

[151] C. McLarty, *Elementary Categories, Elementary Toposes*, Clarendon Press, Oxford, 1995.

[152] P. Melliès, Axiomatic rewriting theory I: a diagrammatic standardisation theorem, in *Processes, Terms and Cycles: Steps on the Road to Infinity*, Lecture Notes in Computer Science **3838**, Springer, 2005, pp. 554–638. Also available at http://www.pps.jussieu.fr/ mellies/papers/jwkfestschrift.pdf.

[153] R. Milner, J. Parrow and D. Walker, A Calculus of Mobile Processes Pt.1, *Information and Computation* **100:1** (1992), 1–40. Also available at http://www.lfcs.inf.ed.ac.uk/reports/89/ECS-LFCS-89-85/

[154] G. Moore and N. Read, Nonabelions in the the fractional quantum Hall effect, *Nucl. Phys. B* **360** (1991), 362–396.

[155] J. Morton, Double bicategories and double cospans, *Journal of Homotopy and Related Structures*, **4:1** (2009), 389–428. Also available at arxiv:math/0611930.

[156] J. Moschovakis, Intuitionistic logic, *Stanford Encyclopedia of Philosophy*, available at http://plato.stanford.edu/entries/logic-intuitionistic/.

[157] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, Cambridge U. Press, Cambridge, 2000.

[158] *2-pullbacks*, http://ncatlab.org/nlab/show/2-pullback#strict_weighted_limits

[159] *group object*, http://ncatlab.org/nlab/show/group+object#examples

[160] 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.

[161] M. B. Pour-El and J. I. Richards, *Computability in Analysis and Physics*, Springer, Berlin, 1989. Also available at ⟨http://projecteuclid.org/euclid.pl/1235422916⟩.

[162] J. Power, Premonoidal categories as categories with algebraic structure, *Theoretical Computer Science* **278** 1–2 (2002), 303–321.

[163] A. Preller, J. Lambek, Free compact 2-categories, *Mathematical Structures in Computer Science* **17** 2 (April 2007), 309–340. Also available at ftp://ftp.math.mcgill.ca/pub/lambek/freecompact.pdf.

[164] F. Reif, *Fundamentals of Statistical and Thermal Physics*, McGraw–Hill, New York, 1965.

[165] A. Rényi, On measures of information and entropy, *Proceedings of the 4th Berkeley Symposium on Mathematics, Statistics and Probability*, 1960, pp. 547–561. Also available at ⟨http://digitalassets.lib.berkeley.edu/math/ucb/text/math_s4_v1_article-27.pdf⟩.

[166] C. P. Roberts, *The Bayesian Choice: From Decision-Theoretic Foundations to Computational Implementation*, Springer, Berlin, 2001.

[167] E. P. Robinson and J. Power. Premonoidal categories and notions of computation. *Math. Struct. in Comp. Sci.* **7** 5 (1997), 453–468. Also available at http://www.eecs.qmul.ac.uk/~edmundr/pubs/mscs97/premoncat.ps.

[168] E. Rowell, R. Stong and Z. Wang, On classification of modular tensor categories, available at arXiv:0712.1377.

[169] S. Sawin, Links, quantum groups and TQFTs, *Bull. Amer. Math. Soc.* **33** (1996), 413-445. Also available at arXiv:q-alg/9506002.

[170] A. Schalk, What is a categorical model for linear logic? Available at http://www.cs.man.ac.uk/~schalk/notes/llmodel.pdf.

[171] C. Schommer-Pries, The Classification of two-dimensional extended topological field theories, PhD thesis. Available at http://sites.google.com/site/chrisschommerpriesmath/Home/Schommer-Pries-Thesis.pdf.

[172] M. Schönfinkel, Über die Bausteine der mathematischen Logik, *Math. Ann.* **92** (1924), 305–316. Also available at 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.

[173] 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.

[174] 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.

R. A. G. Seely, Modeling computations: a 2-categorical framework, in *Proc. Symp. Logic Comp. Sci. 1987*, Computer Society of the IEEE, pp. 65–71. Also available at http://www.math.mcgill.ca/rags/WkAdj/LICS.pdf.

[175] 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.

[176] P. Selinger, Lecture notes on the lambda calculus, available at http://www.mscs.dal.ca/~selinger/papers/#lambdanotes.

[177] 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/#dagger.

[178] P. Selinger and B. Valiron, A lambda calculus for quantum computation with classical control, *Math. Struct. Comp. Sci.* **13** (2006), 527–552. Also available at http://www.mathstat.dal.ca/~selinger/papers/#qlambda.

[179] M. Shulman, Constructing symmetric monoidal bicategories. Avaliable at http://arxiv.org/abs/1004.0993.

[180] M. Shulman, Re: Inevitability in Mathematics, *n-Category Café* (2010). Available at https://golem.ph.utexas.edu/category/2010/06/inevitability_in_mathematics.html#c033741

[181] M.-C. Shum, Tortile tensor categories, *Jour. Pure Appl. Alg.* **93** (1994), 57–110.

[182] 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 at arXiv:gr-qc/9702030.

[183] R. J. Solomonoff, A formal theory of inductive inference, part I, *Inform. Control* **7** (1964), 1–22. Also available at ⟨http://world.std.com/~rjs/1964pt1.pdf⟩.

[184] S. Soloviev, Proof of a conjecture of S. Mac Lane, *Ann. Pure Appl. Logic* **90** (1997), 101–162.

[185] J. Stasheff, Homotopy associativity of H-spaces I, II, *Trans. Amer. Math. Soc.* **108** (1963), 275–312.

[186] Michael Stay and L. G. Meredith, Higher category models of the pi-calculus, to appear. Available at arXiv:1504.04311.

[187] A. Stern, Anyons and the quantum Hall effect – a pedagogical review, *Ann. Phys.* **323** (2008), 204–249. Available at arXiv:0711.4697.

[188] M. Stone, ed., *Quantum Hall Effect*, World Scientific, Singapore, 1992.

[189] R. Street, Fibrations in bicategories, *Cahiers de topologie et géométrie différentielle catégoriques*, **21** 2 (1980), 111–160. Also available at http://www.numdam.org/item?id=CTGDC_1980__21_2_111_0

[190] R. Street, Functorial calculus in monoidal bicategories, *Applied Categorical Structures* **11** (2003) 219–227. Also available at http://www.maths.mq.edu.au/~street/Extraord.pdf.

[191] R. Street, Frobenius monads and pseudomonoids, *J. Math. Physics* **45 (10)** (October 2004) 3930–3948. Also available at http://www.maths.mq.edu.au/~street/Frob.pdf.

[192] E. J. Stoddard, Apparatus for obtaining power from compressed air, US Patent 1,926,463. Available at
⟨http://www.google.com/patents?id=zLRFAAAAEBAJ⟩.

[193] M. E. Szabo, ed., *Collected Papers of Gerhard Gentzen*, North–Holland, Amsterdam, 1969.

[194] L. Szilard, On the decrease of entropy in a thermodynamic system by the intervention of intelligent beings, *Zeit. Phys.* **53** (1929) 840–856. English translation in H. S. Leff and A. F. Rex (eds.) *Maxwell's Demon. Entropy, Information, Computing,* Adam Hilger, Bristol, 1990.

[195] K. Tadaki, A generalization of Chaitin's halting probability Ω and halting self-similar sets, *Hokkaido Math. J.* **31** (2002), 219–253. Also available at arXiv:nlin.CD/0212001.

[196] K. Tadaki, A statistical mechanical interpretation of algorithmic information theory. Available at arXiv:0801.4194.

[197] K. Tadaki, A statistical mechanical interpretation of algorithmic information theory III: Composite systems and fixed points. *Proceedings of the 2009 IEEE Information Theory Workshop, Taormina, Sicily, Italy*, to appear. Also available at arXiv:0904.0973.

[198] T. Trimble, *Linear Logic, Bimodules, and Full Coherence for Autonomous Categories*, Ph.D. thesis, Rutgers University, 1994.

[199] A. S. Troelstra, *Lectures on Linear Logic*, Center for the Study of Language and Information, Stanford, California, 1992.

[200] V. G. Turaev, *Quantum Invariants of Knots and 3-Manifolds*, de Gruyter, Berlin, 1994.

[201] A. van Tonder, A lambda calculus for quantum computation, *SIAM Jour. Comput.* **33** (2004), 1109–1135. Also available at arXiv:quant-ph/0307150.

[202] S. Willerton, Two 2-Traces. Available at
http://www.simonwillerton.staff.shef.ac.uk/ftp/TwoTracesBeamerTalk.pdf.

[203] E. Witten, Quantum field theory and the Jones polynomial, *Comm. Math. Phys.* **121** (1989), 351–399.

[204] W. K. Wootters and W. H. Zurek, A single quantum cannot be cloned, *Nature* **299** (1982), 802–803.

[205] D. N. Yetter, *Functorial Knot Theory: Categories of Tangles, Coherence, Categorical Deformations, and Topological Invariants*, World Scientific, Singapore, 2001.

[206] M. Ziegler. Does quantum mechanics allow for infinite parallelism? Internat. J. Theoret. Phys. 44, 11 (2005), 2059–2071.