3
Processes as Diagrams

We haven't really paid much attention to thought as a process. We have engaged in thoughts, but we have only paid attention to the content, not to the process.

– David Bohm and David Peat, 1987

In this chapter we provide a practical introduction to basic diagrammatic reasoning, namely how to perform computations and solve problems using diagrams. We also demonstrate why diagrams are far better in many ways than traditional mathematical notation. The development and study of diagrammatic languages is a very active area of research, and intuitively obvious aspects of diagrammatic reasoning have actually taken many years to get right. Luckily, the hard work needed to formalise the diagrams in this book has already been done! So, all that remains to do is reap the benefits of a nice, graphical language.

Along the way, we will encounter *Dirac notaton*. Readers who have previously studied quantum mechanics or quantum information theory may have already seen Dirac notation used in the context of linear maps. Here, we'll explain how it arises as a one-dimensional fragment of the two-dimensional graphical language. Thus, readers not familiar with Dirac notation will learn it as a special case of the graphical notation we use throughout the book.

We also introduce the notion of a *process theory*, which provides a means of interpreting diagrams by fixing a particular collection of (physical, computational, mathematical, edible, etc.) systems and the processes that these systems might undergo (being heated up, sorted, multiplied by two, cooked, etc.).

As we pointed out in Chapter 1, taking process theories as our starting point represents a substantial departure from standard practice in many disciplines. Rather than forcing ourselves to totally understand single systems before even thinking about how those systems compose and interact, we will seek to understand systems primarily in terms of their interactions with others. Rather than trying to understand Dave the dodo by dissecting him (at which point, he'll look pretty much like any other fat bird), we will turn him loose in the world and see what he does.

This turns out to be very close in spirit to the aims of *category theory*, which we will meet briefly in the advanced material at the end of this chapter.

3.1 From Processes to Diagrams

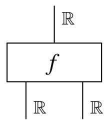
Let's have a look at how diagrammatic language gives us a general way to speak about processes and how they compose, and show that these diagrams provide a rigorous mathematical notation on par with traditional mathematical formulas.

3.1.1 Processes as Boxes and Systems as Wires

We shall use the term *process* to refer to anything that has zero or more inputs and zero or more outputs. For instance, the function

$$f(x,y) = x^2 + y \tag{3.1}$$

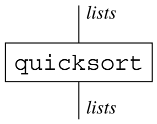
is a process that takes two real numbers as input and produces one real number as output. We represent such a process as a *box* with some *wires* coming in the bottom to represent input systems and some wires coming out the top to represent output systems. For example, we could write the function (3.1) like this:



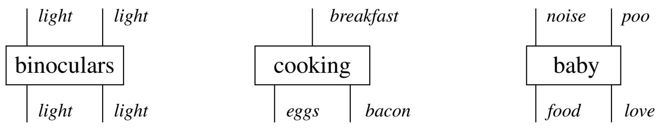
(3.2)

The labels on wires are called *system-types* or simply *types*.

Similarly, a computer program is a process that takes some data (e.g. from memory) as input and produces some new data as output. For example, a program that sorts lists might look like this:



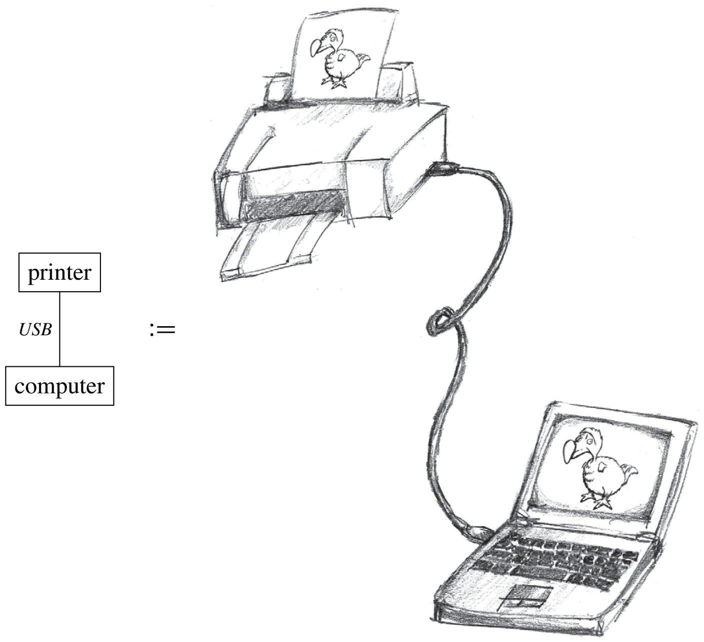
The following are also perfectly good processes:



Clearly the world around us is packed with processes!

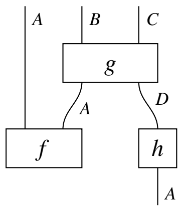
Note how sometimes a box is actually labelled with a process that occurs over time (e.g. 'the process of cooking breakfast'), while other times we label a box with an apparatus (e.g. 'binoculars' or 'baby'). Typically this means 'the process of using this apparatus to convert input systems into output systems'.

In some cases, wires in the diagram may even correspond to actual physical wires, for example:

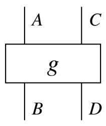


Though, of course, this needs not always be the case. Wires can also represent e.g. ‘aligning apertures’ on lab equipment, or shipping something by boat. The important thing is that wires represent the flow of data (or more general ‘stuff’) from one process to another.

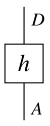
As we have just seen, we can *wire together* simple processes to make more complicated processes, which are described by *diagrams*:



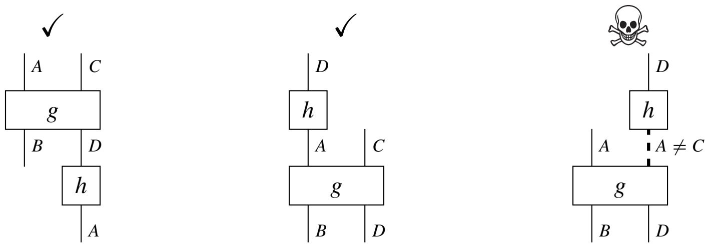
In such a diagram outputs can be wired to inputs only if their types match. For example, the following two processes:



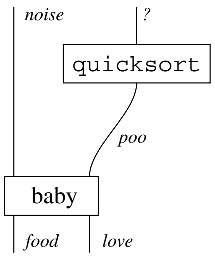
and



can be connected in some ways, but not in others, depending on the types of their wires:

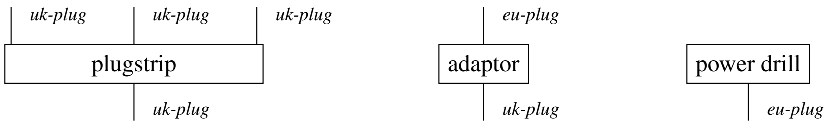


This restriction on which wirings are allowed is an essential part of the language of diagrams, in that it tells us when it makes sense to apply a process to a certain system and prevents occurrences like this:

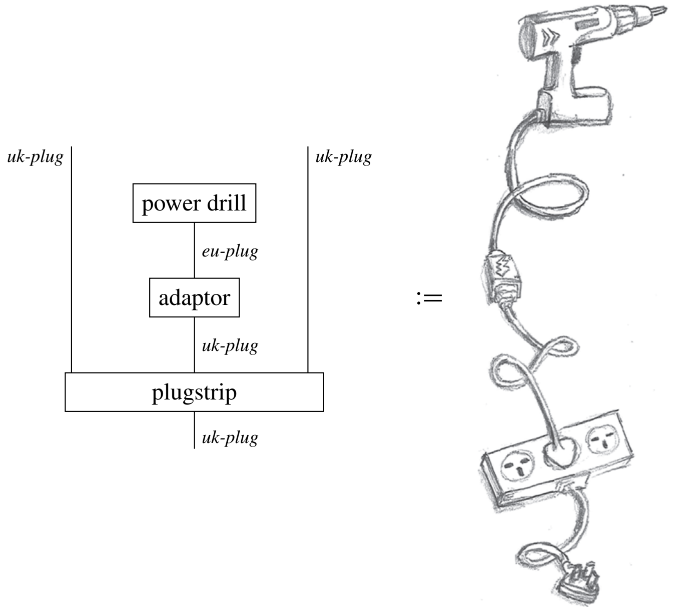


which probably wouldn't be very good for your computer! Much like *data types* in computer science, the types on wires tell us what sort of data (or stuff) the process expects as input and what it produces as output. For example, a calculator program expects numbers as inputs and produces numbers as outputs. Thus, it can't make sense of 'Dave' as input, nor will it ever produce, say, 'carrots' as an output.

Another useful example is that of electrical appliances. Suppose we consider processes to be appliances that have plugs as inputs (i.e. where electricity is 'input') and sockets as outputs:



Then, system-types are just the shape of the plug (UK, European, etc.). Clearly we can't connect a plug to a socket of the wrong shape, so the type information on wires gives us precisely the information we need to determine which wirings are possible. For example, one possible wiring is:



Though this is a toy example, it is often very useful to draw diagrams whose boxes refer to devices that actually exist in the real world (e.g. in a lab). We will refer to such diagrams as *operational*.

3.1.2 Process Theories

Usually one is not interested in all possible processes, but rather in a certain class of related processes. For example, practitioners of a particular scientific discipline will typically only study a particular class of processes: physical processes, chemical processes, biological processes, computational processes, mathematical processes, etc. For that reason, we organise processes into *process theories*. Intuitively, a process theory tells us how to interpret wires and boxes in a diagram (cf. (i) and (ii) below) and what it means to form diagrams, that is, what it means to wire boxes together (cf. (iii) below).

Definition 3.1 A *process theory* consists of:

(i) a collection T of *system-types* represented by wires,
(ii) a collection P of *processes* represented by boxes, where for each process in P the input types and output types are taken from T, and
(iii) a means of ‘wiring processes together’, that is, an operation that interprets a diagram of processes in P as a process in P.

In particular, (iii) guarantees that

process theories are ‘closed under wiring processes together’, since it is this operation that tells us what ‘wiring processes together’ means. In some cases this operation consists of literally plugging things together with physical wires, as in the example of the power drill. In other cases this will require some more work, and sometimes there is more than one obvious choice available. We shall see in Section 3.2 that in traditional mathematical practice one typically breaks down ‘wiring processes together’ in two sub-operations: parallel composition and sequential composition of processes.

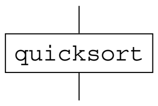
Example 3.2 Some process theories we will encounter are:

- **functions** (types = sets)
- **relations** (types = sets, again)
- **linear maps** (types = vector spaces, or Hilbert spaces)
- **classical processes** (types = classical systems)
- **quantum processes** (types = quantum and classical systems)

We will use the first two mainly in various examples and exercises, whereas the remaining three will play a major role in this book.

Remark 3.3 Note how we refer to a particular process theory by saying what the ‘processes’ are, leaving the types implicit. This tends to be a good practice, because the processes are the important part. For instance, we’ll see that the process theory of functions is quite different from that of relations, so it will not do to just call both ‘sets’.

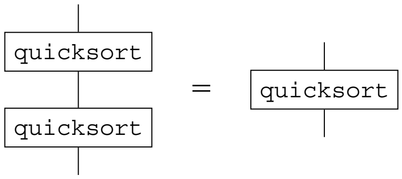
Since a process theory tells us how to interpret diagrams as processes, it crucially tells us when two diagrams represent the same process. For example, suppose we define a simple process theory for **computer programs**, where the types are data-types (e.g. integers, booleans, lists) and the processes are computer programs. Then, consider a short program, which takes a list as input and sorts it. It might be defined this way (don’t worry if you can’t read the code, neither can half of the authors):



:=

```
qs [] = []
qs (x :: xs) =
  qs [y | y <- xs; y < x] ++ [x] ++
  qs [y | y <- xs; y >= x]
```

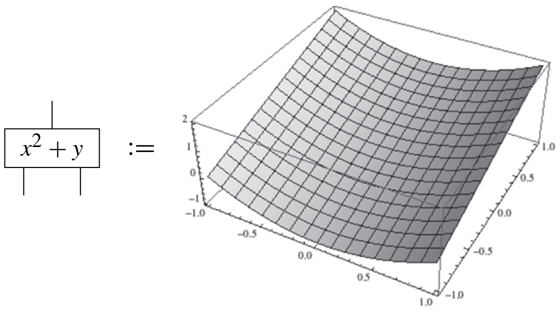
Wiring together programs means sending the output of one program to the input of another program. Taking two programs to be equal if they behave the same (disregarding some details like execution time, etc.), our process theory yields equations like this one:



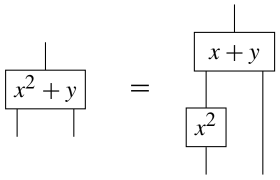
i.e. sorting a list twice has the same effect as sorting it once.

The reason we call a process theory a *theory* is that it comes with lots of such equations, and these equations are precisely what allows us to draw conclusions about the processes we are studying.

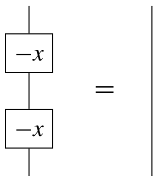
To take another example, the function we defined at the beginning of Section 3.1.1 lives in the process theory of **functions**. Types are sets and processes are functions between sets. We consider two functions equal if they behave the same; i.e. they have the same graph:



(Note how two input wires means a function of two variables, hence the three-dimensional plot.) Since they have the same graph, we have:



as well as:



Exercise 3.4 Cook up some of your own process theories. For each one, answer the following questions:

1. What are the system-types?
2. What are the processes?

3. How do processes compose?
4. When should two processes be considered equal?

One thing to note is we haven't yet been too careful to say what a diagram actually i. A complete description of a diagram consists of

1. what boxes it contains and
2. how those boxes are connected.

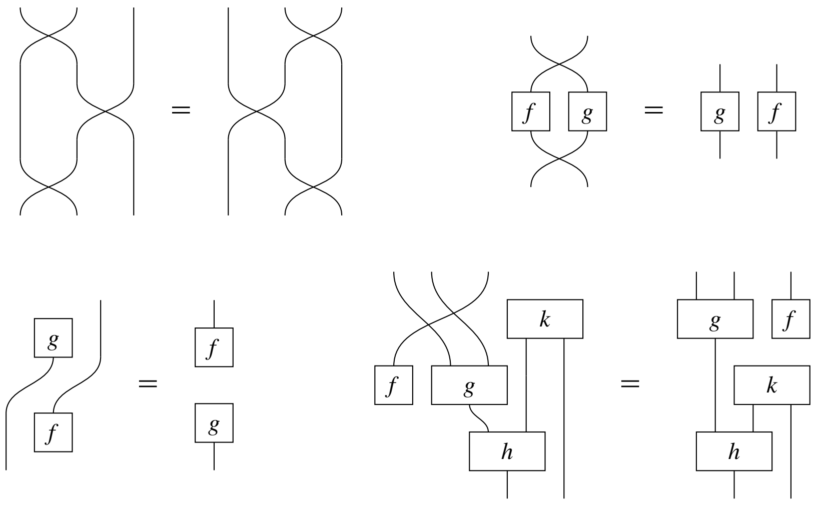
So the diagram refers to the 'drawing' of boxes and wires without the interpretation within a process theory. However, it makes no reference to where boxes are written on the page. An immediate corollary of this fact is the following.

Corollary 3.5 If two diagrams can be deformed into each other (without changing connections, of course), then they are equal.

Or put more succinctly:

$$\boxed{\textit{Only connectivity matters!}}$$

Example 3.6 Here are some pairs of equal diagrams:



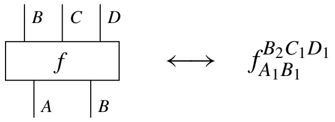
Remark* 3.7 The first pair of equal diagrams in Example 3.6 is called the *Yang–Baxter equation*, which some readers may have already encountered in the mathematical physics literature.

3.1.3 Diagrams Are Mathematics

Diagrams provide a rigorous language for doing mathematics on par with traditional mathematical formulas. Just as traditional formulas can be used to define new concepts and reason about many mathematical structures (e.g. sets, groups, topological spaces, vector spaces), diagrams can be used to define new concepts and reason about process theories.

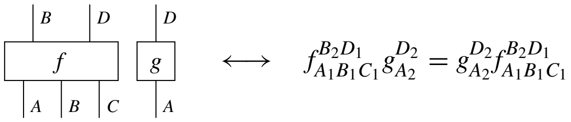
Perhaps as a result of their upbringing, some people will not accept something as a rigorous mathematical object unless it can be expressed as a formula of some kind. To please those people (and also in order to give another perspective on diagrams) we will briefly introduce a formula-like notation. However, once the correct intuition is in place, we shall drop this notation and (almost) always work with diagrams directly.

We can turn a box into a formula as follows:

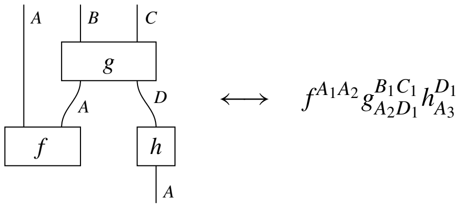


In the formula on the right, the subscripts A_1 and B_1 of f represent inputs, while the superscripts B_2, C_1, and D_1 represent outputs. Why did we add numbers to each of these? This numbering doesn't have any meaning in its own right, but it is necessary to eliminate ambiguity whenever we have more than one wire of the same type in a single diagram. For instance, the second input to f is a different wire from the first output, thus we refer to these wires as B_1 and B_2, respectively. We will call a type with some subscript a *wire name*, since it refers to a particular wire in the diagram. The subscripts are of course unnecessary in diagrams because it is already clear that B_1 and B_2 are two distinct wires, since they are in different positions on the page.

The entire expression $f_{A_1B_1C_1}^{B_2D_1}$ is called a *box name*, since it refers to a particular box in the diagram. We can express more than one box by writing down a sequence of box names, where the order we write them down doesn't matter:


B
D
D
f
g
A
B
C
A
$\longleftrightarrow$
$f_{A_1B_1C_1}^{B_2D_1} g_{A_2}^{D_2} = g_{A_2}^{D_2} f_{A_1B_1C_1}^{B_2D_1}$

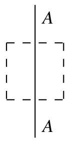
Note that all of the upper and lower wire names in the formula on the right are distinct. This is because no wires are connected to each other in the diagram. The way we represent connections is by repeating the name of a wire, once as an upper wire name (which corresponds with the output of a box) and once as a lower wire name (which corresponds with the input):



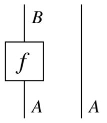
We can give any subscript to a repeated wire name as long as it doesn't clash with any of the other wire names. For example, the following two formulas represent exactly the same diagram:

$$f^{A_1A_2} g_{A_2}^{B_1} = f^{A_1A_4} g_{A_4}^{B_1} \tag{3.3}$$

We will also consider a single wire of type A as a 'special' box:

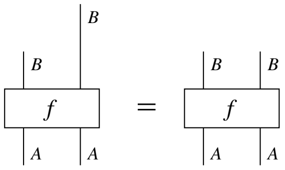


with a corresponding box name $1_{A_1}^{A_2}$. Hence we can write, e.g.:



$$\longleftrightarrow \quad f_{A_1}^{B_3} 1_{A_2}^{A_4}$$

In almost every respect, these special boxes are just boxes like any other and may be interpreted as the process that 'does nothing'. However, when we connect them to the input or output of another box, they vanish:



$$\longleftrightarrow \quad f_{A_1A_2}^{B_1B_3} 1_{B_3}^{B_2} = f_{A_1A_2}^{B_1B_2} \tag{3.4}$$

The following definition summarises all of above.

Definition 3.8 A *diagram formula* is a sequence of box names such that all wire names are unique, except for matched pairs of upper and lower names. Two diagram formulas are equal if and only if one can be turned into the other by:

(a) changing the order in which box names are written,
(b) adding or removing identity boxes, as in (3.4), or
(c) changing the name of a repeated wire name.

With this concept in hand, we can now provide a definition of a diagram that makes direct reference to a formula-like counterpart.

Definition 3.9 A *diagram* is a pictorial representation of a diagram formula where box names are depicted as boxes (or various other shapes), wire names are depicted as input and output wires of the boxes, and repeated wire names tell us which outputs are connected to which inputs.

Exercise 3.10 Draw the diagrams of the following diagram formulas:

$$f_{B_1C_2}^{C_4} g_{C_4}^{D_3} \qquad f_{A_1}^{A_1} \qquad g_{B_1}^{A_1} f_{A_1}^{B_1} \qquad 1_{A_1}^{A_6} 1_{A_2}^{A_5} 1_{A_3}^{A_4}$$

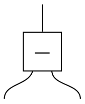
Use the convention that inputs and outputs are numbered from left to right.

The above construction shows that all of our work with diagrams could be translated, without ambiguity, into work on diagram formulas. We will mostly stick to diagrams since they are easier to visualise, are easier to work with, and do away with the bureaucratic overhead of extra subscripts. However, we will occasionally use diagram formulas as a handy tool for computing the process of a given diagram, as in Sections 3.3.3 and 5.2.4.

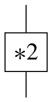
3.1.4 Process Equations

In Example 3.6 above, we wrote down many equations involving diagrams on the left-hand side (LHS) and the right-hand side (RHS). These *diagram equations* always hold, regardless of how we interpret the boxes and wires. That is, they are true in any process theory. On the other hand, within a particular process theory it may be possible to represent the same *process* using two distinct diagrams. We have already seen a couple of examples of this in Section 3.1.2.

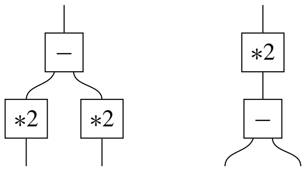
Returning to the process theory **functions**, suppose we define two processes: ‘minus’, which takes two inputs m, n and subtracts them, outputting $m - n$:



and ‘times 2’, which takes a single input and multiplies it by 2:



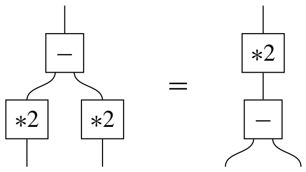
where all of the wires have type $\mathbb{R}$. Now, consider the following two diagrams:



As *diagrams*, these two are not equal. However, if we look at the *processes* they represent, we see that the process on the left multiplies both inputs by 2, then subtracts one from the other. The one on the right first subtracts the inputs, then multiplies the result by 2. Even though they have different diagrams, both processes compute the same function, namely:

$$2m - 2n = 2(m - n)$$

Thus, these distinct diagrams represent the same process when interpreted in the process theory of **functions**; i.e. they are *equal as processes*, which we write simply as:



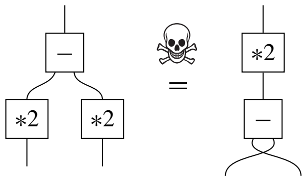
(3.5)

This is called a *process equation*. A diagram equation is a (trivial) special case of a process equation, since equal diagrams will always be interpreted as equal processes.

Remark* 3.11 There is a special class of theories where diagram equations and process equations coincide. That is, two processes are equal if and only if they have the same diagram. These are called *free process theories*, in analogy with the use of the term 'free' in algebra. Here, 'free' means 'no extra equations'. In general, imposing extra equations between processes puts a constraint on which interpretations of boxes are allowed (i.e. only those satisfying the extra equations). Free process theories have the property that any interpretation of their boxes as processes (in some other process theory) extends to a consistent interpretation of diagrams.

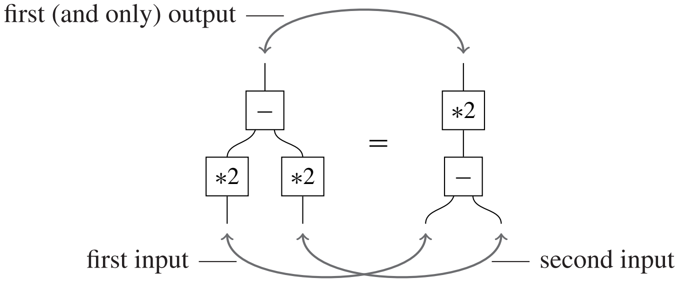
Exercise 3.12 Give the (diagram) equations that express the algebraic properties of associativity, unitality, and commutativity of a two-input, one-output process. Can you do the same for distributivity of a pair of two-input processes (e.g. 'plus' and 'times')? If not, what's the problem?

The notion of a diagram equation may seem completely obvious (because it is!), but it is important to note that such an equation gives a bit more information than you might first think. To see why, note that equation (3.5) is true, but the following equation is **false**:



(3.6)

The LHS will map the inputs m, n to $2m - 2n$, whereas the RHS will map those same inputs to $2(n - m) \neq 2m - 2n$. The only difference between this equation and the true equation (3.5) is that the inputs of the RHS are flipped. Even though we do not write down the names of wires in the diagrams (as we do for the diagram formulas of Definition 3.8), these inputs and outputs do have distinct identities. Furthermore, for a diagram equation to be well formed, both sides must have the *same* inputs and outputs, which suggests that there is a correspondence between them:



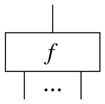
In terms of diagram formulas, this would be reflected by the fact that we call e.g. the first input A_1 and the second input A_2 on both sides of the equation. In a diagram, we show which inputs and outputs are in correspondence by their positions on the page. Bearing this rule in mind, there is clearly a difference between the correct equation (3.5) and the erroneous (3.6).

Equations between processes with only one ‘output’ should already be familiar from algebra. When we write an equation like:

$$2m - 2n = 2(m - n)$$

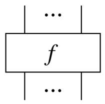
we distinguish the inputs m, n of the formulas on the LHS and RHS by giving them names: namely ‘m’ and ‘n’. In algebra, there is always exactly one ‘output’ of a formula, namely the value computed when all the variables have been substituted by numbers, so we don’t bother to give it a name. On the other hand, diagrams can have *many* outputs in general, so we need to distinguish those as well.

We typically think of multivariable functions, such as the algebraic operations above, as taking one or more inputs to a single output:



$$:: \quad (a_1, \ldots, a_m) \mapsto b$$

where the ‘$\mapsto$’ symbol means ‘maps to’ (cf. Appendix). However, in the process theory **functions** we can also consider functions of this form:

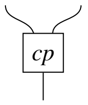


$$:: \quad (a_1, \ldots, a_m) \mapsto (b_1, \ldots, b_n) \tag{3.7}$$

Remark* 3.13 In functional programming, one might encounter a function with many outputs in the sense of (3.7) in expressions like:

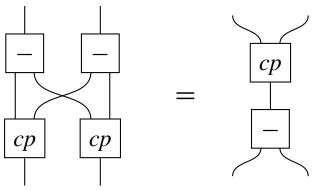
$$\text{‘}\mathbf{let}\ (b_1, \ldots, b_n) = f(a_1, \ldots, a_m)\ \mathbf{in}\ \ldots\text{’}$$

A simple example of a two-output function is *cp*, which takes in a number *n* and sends a copy of *n* down both of its output wires:



$$:: \quad n \mapsto (n, n)$$

The function *cp* satisfies the following process equation with minus:



(3.8)

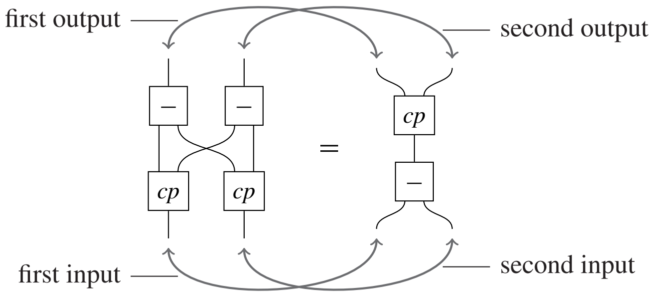
The process on the LHS takes in two inputs *m* and *n*, copies each of them, and sends one copy of each to two separate subtraction operations. So, the output is two copies of $m - n$, one on each wire:

$$(m, n) \mapsto (m, m, n, n) \mapsto (m, n, m, n) \mapsto (m - n, m - n)$$

The process on the RHS takes the inputs *m* and *n*, subtracts them, and copies the result. Again, this yields two copies of $m - n$:

$$(m, n) \mapsto m - n \mapsto (m - n, m - n)$$

As before, implicit in equation (3.8) is a correspondence between inputs and outputs:



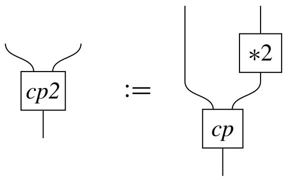
Exercise 3.14 Write (3.8) as an equation between diagram formulas.

Exercise* 3.15 Using *cp*, is it now possible to express the distributivity equation between 'plus' and 'times' mentioned in Exercise 3.12?

3.1.5 Diagram Substitution

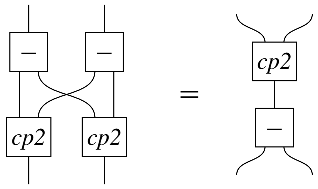
We can obtain new process equations from old ones via *diagram substitution*. Along with simple diagram deformation, diagram substitution will be the most common style of calculation in this book.

Diagram substitution refers to the act of replacing some subdiagram of a given diagram with a new subdiagram using a process equation. The resulting diagram will then represent an equivalent process. It is easiest to see how this works by example. Since we are getting a bit bored with our copy map, which produces two identical copies of a number, we define a more exciting one that also multiplies the right copy by two:



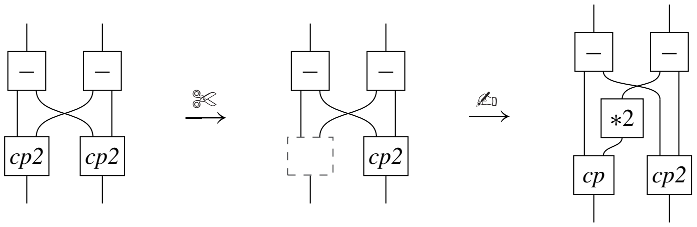
(3.9)

Now, we can ask which equations *cp2* satisfies. We hypothesise that it satisfies a similar equation to (3.8):

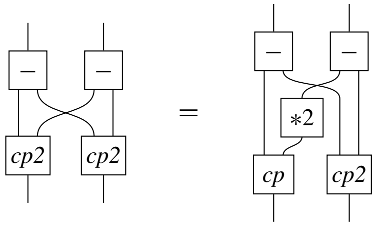


(3.10)

Starting with the LHS of (3.10), we apply equation (3.9) to expand the definition of *cp2*. So, let's cut out the LHS of (3.9) and replace it with the RHS of (3.9):

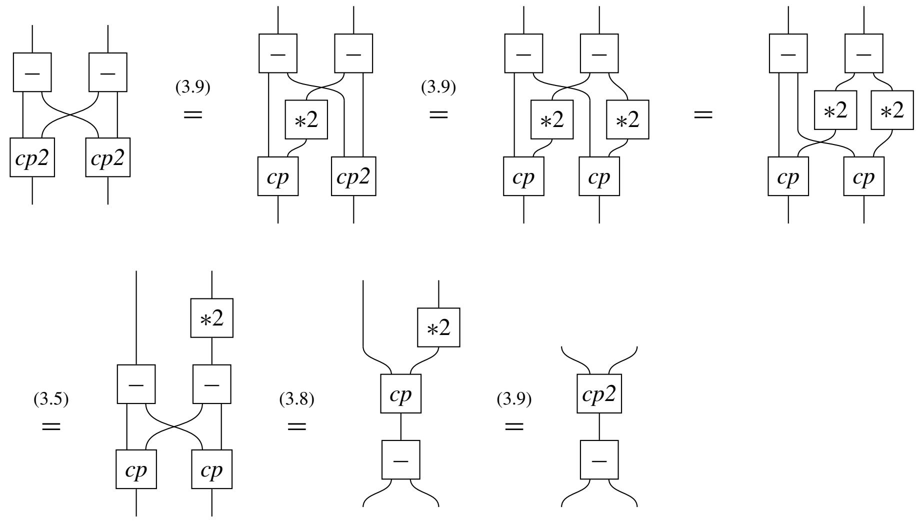


Thus, from (3.9), we can deduce the following equation, by diagram substitution:



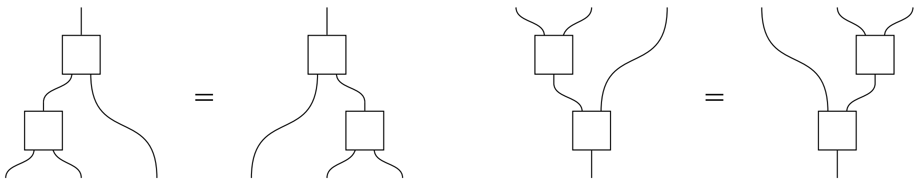
Remark 3.16 An important aspect of diagram substitution is that the RHS should be plugged into the *same place* as the LHS was. For example, the first input of the RHS should be connected to the same wire that the first input of the LHS was previously connected to, and so on. This is where the correspondence between the inputs/outputs in a diagram equation that we discussed in the previous section plays an important role.

Continuing this procedure, we can construct a proof of equation (3.10):



Each of the steps in the proof is marked with the process equation we used, except for step 3, which is just a diagram deformation. Also note that for the last step, (3.9) is being used backwards, which of course is okay.

Remark* 3.17 (algebra vs. coalgebra) While *cp* is very intuitive in what it does, it is nothing like the usual operations that one encounters in algebra, like ‘sum’ and ‘times’, exactly for the reason that it has one input and two outputs. The realisation that many interesting operations indeed have multiple outputs resulted in a new research area called *coalgebra*. In terms of diagrams, ‘co’ can be understood as ‘upside-down’. So, for instance, a ‘multiplication’ operation can be flipped upside-down to produce a ‘comultiplication’. Just as one can talk about associativity, unitality and commutativity in algebra, one can talk about coassociativity, counitality and cocommutativity in coalgebra, and these conditions are obtained simply by flipping the equations of algebra upside-down, e.g.:



associativity coassociativity

3.2 Circuit Diagrams

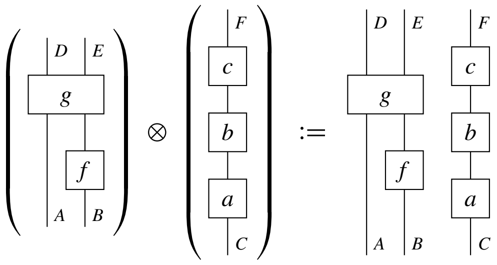
Recalling that boxes represent processes, we can define two basic *composition operations*on processes with the following interpretations:

$$f \otimes g := \text{‘process } f \text{ takes place } \underline{\text{while}} \text{ process } g \text{ takes place’}$$
$$f \circ g := \text{‘process } f \text{ takes place } \underline{\text{after}} \text{ process } g \text{ takes place’}$$

These operations allow us to define an important class of diagrams called *circuits*. Despite their importance, we will see in the next chapter that the most interesting processes are in fact those that fail to be circuits.

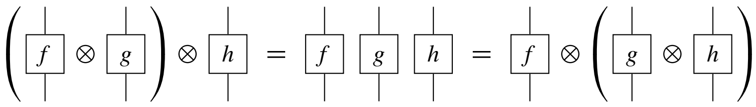
3.2.1 Parallel Composition

The *parallel composition* operation consists of placing a pair of diagrams side by side. We write this operation using the symbol ‘⊗’:



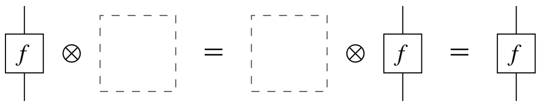
Any two diagrams can be composed in this manner, since placing diagrams side by side does not involve connecting anything. This reflects the intuition that both processes are happening independently of each other.

This composition operation is associative:



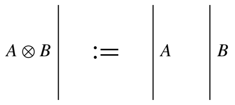
(3.11)

and it has a unit, the empty diagram:

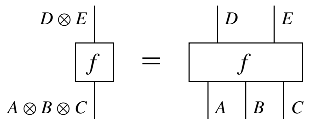


(3.12)

Parallel composition is defined for system-types as well. That is, for types A and B, we can form a new type $A \otimes B$, called the *joint system-type*:

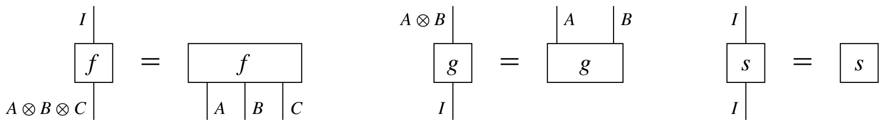


We have already made use of composition of system-types implicitly by drawing boxes that have many inputs and outputs. Formally, a box with three inputs A, B, C and two outputs D and E is the same as a box with a single input $A \otimes B \otimes C$ and a single output $D \otimes E$:



There is also a special ‘empty’ system-type, symbolically denoted I, which is used to represent ‘no inputs’, ‘no ouputs’, or both.

Examples 3.18 Here are some examples of boxes with no input and/or output wires, written both in terms of ‘$\otimes$’ and I and in the usual way:

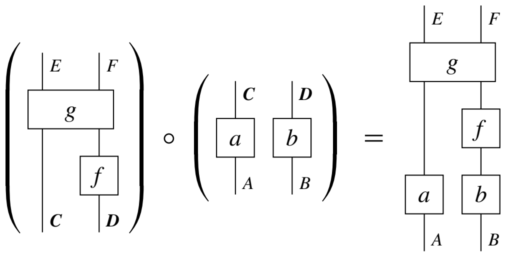


Processes with no inputs and/or outputs play a special role in this book, as we will see starting in Section 3.4.1.

In the diagrammatic notation, we will rarely use the $\otimes$ symbol explicitly, preferring instead to express joint systems as multiple wires.

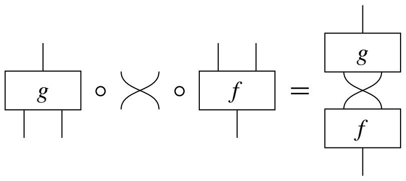
3.2.2 Sequential Composition

The *sequential composition* operation consists of connecting the outputs of a first diagram to the inputs of a second diagram. We write this operation using the symbol ‘$\circ$’:

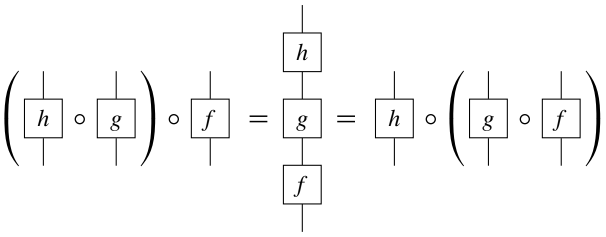


The intuition is that the process on the right happens first, and then the process on the left happens, taking the output of the first process as its input. Clearly not any pair of diagrams can be composed in this manner: the number and type of the inputs of the left process must match the number and type of the outputs of the right process.

We assume that sequential composition will connect outputs to inputs in order. If we wish to vary this order, we can introduce *swaps*:

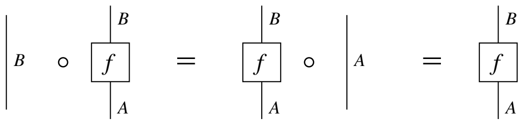


The LHS of this equation is well-defined of course, since the sequential composition operation is also associative:



(3.13)

and it also has a unit. This time, it's a plain wire of appropriate type:



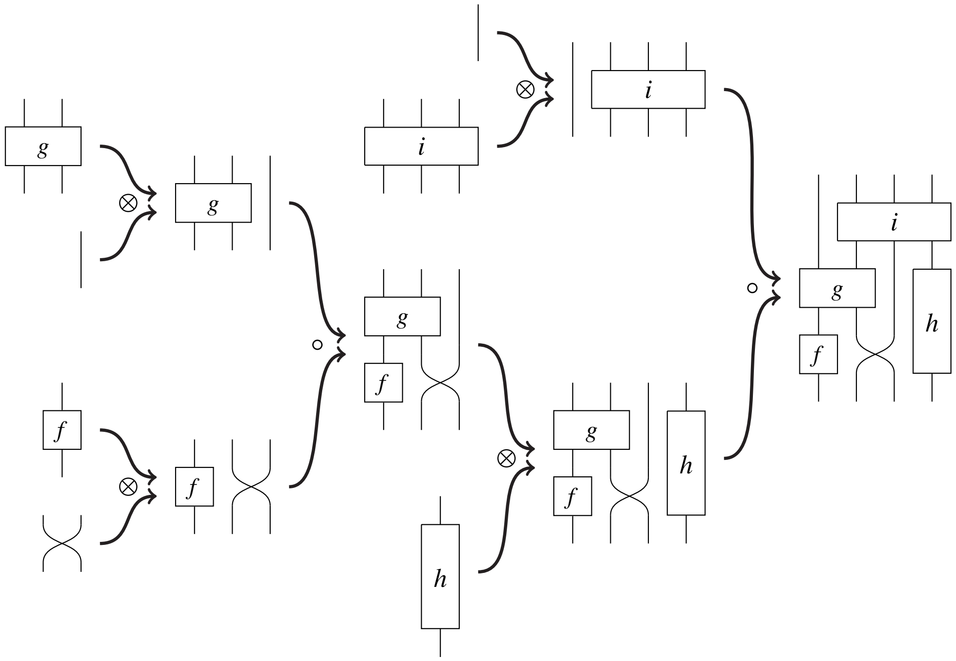
(3.14)

We already encountered these plain wires or *identities* in Section 3.1.3, where we mentioned that as a process, we can think of them as 'doing nothing' to the input. For a system of type A we will denote the identity on A symbolically as 1_A. The empty diagram is also an identity, but on the system-type I, so we denote it as 1_I.

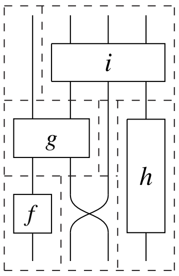
3.2.3 Two Equivalent Definitions of Circuits

Definition 3.19 A diagram is a *circuit* if it can be constructed by composing boxes, including identities and swaps, by means of $\otimes$ and $\circ$.

Every diagram that we have seen in this chapter is in fact a circuit. Here is an example of the assembly of such a circuit:



We can still recognise this assembly structure in the resulting diagram:



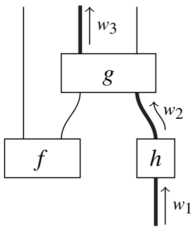
(3.15)

Note however that such a decomposition of a diagram does not uniquely determine the assembly process (e.g. associativity of ⊗ and ∘ allows for two manners of composing three boxes), and the same diagram may even allow for several distinct decompositions. We will see in the next section how this feature makes a non-diagrammatic treatment of circuits in terms of ⊗ and ∘ especially unwieldy.

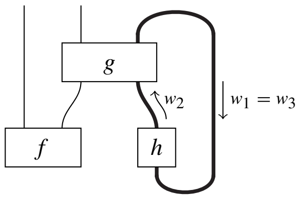
Not all diagrams are circuits. To understand which ones are, we provide an equivalent characterisation that doesn't refer to the manner that one can build these diagrams but instead to a property that has to be satisfied.

Definition 3.20 A *directed path* of wires is a list of wires $(w_1, w_2, \ldots, w_n)$ in a diagram such that for all $i < n$, the wire w_i is an input to some box for which the wire w_{i+1} is an output. A *directed cycle* is a directed path that starts and ends at the same box.

An example of a directed path is shown in bold here:



and an an example of a directed cycle is:

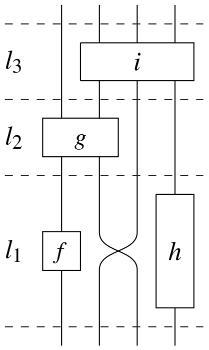


Remark* 3.21 In an area of mathematics called graph theory, graphs without directed cycles are called *directed acyclic*.

Theorem 3.22 The following are equivalent:

- a diagram is a circuit, and
- it contains no directed cycles.

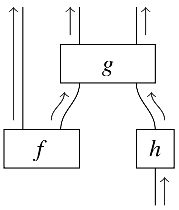
Proof There exists a (non-unique) way to express any diagram with no directed cycles in terms of $\otimes$, $\circ$, and swap. Since the diagram contains no directed cycles, we divide the diagram into 'layers' $l_1, \ldots, l_n$, where the outputs of each box (including identities and swaps) in layer l_i connect only to boxes in layer l_{i+1}. For example, one way to divide diagram (3.15) into layers is:



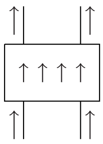
In each layer, $\otimes$ combines boxes into one, and $\circ$ combines the layers together, so we indeed have a circuit. Conversely, any diagram built up using just $\otimes$ and $\circ$ can be decomposed into such layers, which implies the absence of directed cycles. □

Remark* 3.23 In relativity theory, a particular 'division of a process into layers' used in the previous proof is called a *foliation*. We will have a closer look at foliations in Section 6.3.3.

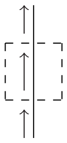
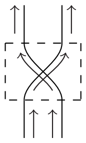
Since circuit diagrams contain no directed cycles, we can give them a 'temporal interpretation' by letting time flow from the bottom of the diagram to the top:



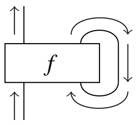
In particular, no wires go 'backwards in time'. This interpretation implicitly assumes that 'inside the boxes' the temporal flow is from the inputs to the outputs:



This is for example the case for identities as well as for swaps:

A canonical example of a diagram that is not a circuit is one involving a *feedback loop*; i.e. a box's output is connected to its own input:



which is indeed a directed cycle consisting of just one wire.

Remark* 3.24 Even more fundamental than foliations, all circuit diagrams admit a *causal structure*, which means that for any two boxes f and g, there are only three possibilities:

(1) f is in the causal past of g,
(2) g is in the causal past of f, or
(3) neither is in the causal past of the other.

We elaborate a bit more on the connection between circuits and the causal structure of spacetime in Section 6.3, when we explore the connections between quantum theory and the theory of relativity.

Circuit diagrams provide the correct language to discuss experimental setups that one can do in a laboratory, which take certain physical systems and/or data as input and which output other physical systems and/or data. 'Parallel' and 'sequential' then directly refer to how the devices are arranged.

Remark 3.25 Some might find the term 'circuit' somewhat oddly chosen for a kind of diagram that <u>excludes</u> feedback loops. This terminology is justified for two reasons. First, it mirrors the terminology used in quantum computing literature (cf. 'quantum circuits'). Second, while it is impossible to 'wire in' feedback loops, we can actually introduce feedback-like behaviour by introducing special boxes called 'caps' and 'cups' to circuits. We will see how this works in Chapter 4.

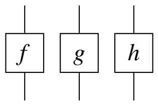
3.2.4 Diagrams Beat Algebra

In Section 3.1.3 we gave a representation of diagrams in terms of diagram formulas in order to convince the reader that one should think of diagrams as mathematical entities on par with traditional formulas. However, the diagram formulas caused an increase in bureaucracy and are not at all easy to parse. We have now seen that it is possible to represent (at least) circuit diagrams in a more traditional algebraic language using just $\otimes$ and $\circ$. One might now wonder if this is a good alternative to using diagrams. The answer is a resounding NO!

An algebraic syntax involving the $\otimes$- and $\circ$-symbols needs to be subjected to many additional equations, all of which are built in to the diagrammatic language. A first example of this is the fact that associativity and unitality of parallel composition are handled automatically:

- Drawing three boxes side by side implicitly assumes associativity:

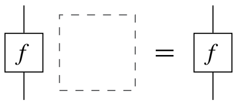
$$(f \otimes g) \otimes h =$$



$$= f \otimes (g \otimes h)$$

- Composition with empty space does nothing:

$$f \otimes 1_I =$$



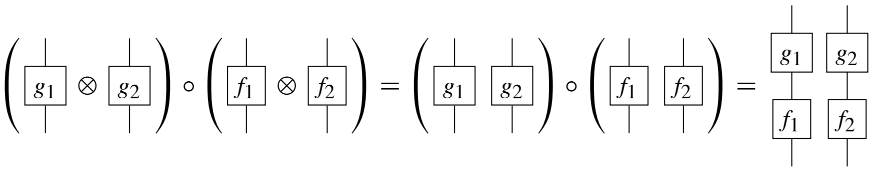
$$= f$$

Associativity and unitality of sequential composition are also built in: associativity is handled again by throwing away the brackets, and unitality is handled by depicting identities as a piece of wire with no box on it.

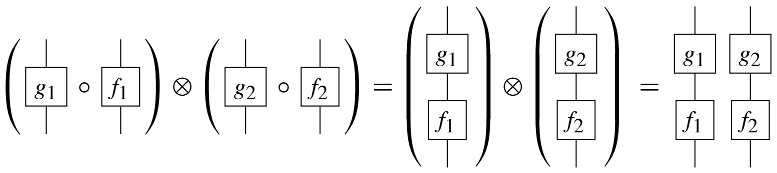
A more striking example occurs when we combine the two compositions. Consider the following two expressions:

$$(g_1 \otimes g_2) \circ (f_1 \otimes f_2) \qquad \text{and} \qquad (g_1 \circ f_1) \otimes (g_2 \circ f_2)$$

Evidently, in the absence of any additional equations on the ⊗- and ∘-symbols these are not equal. Now we compute their corresponding diagrams. For the first expression we obtain:



and for the second we obtain:



So we get the same diagram twice!

Since the ⊗- and ∘-symbols are supposed to be an algebraic syntax for diagrams, we need to impose a new equation:

$$(g_1 \otimes g_2) \circ (f_1 \otimes f_2) = (g_1 \circ f_1) \otimes (g_2 \circ f_2) \tag{3.16}$$

while in the diagrammatic language this is nothing but a tautology! So what is the actual content of (3.16)? It states that the composite process:

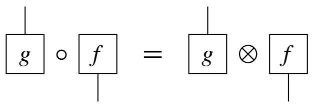
process g_1 takes place while process g_2 takes place,
after
process f_1 takes place while process f_2 takes place,

is the same as the composite process:

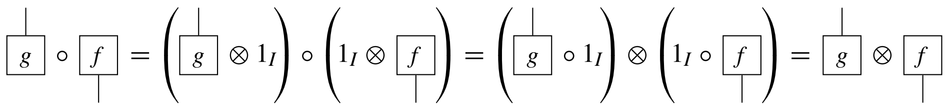
process g_1 takes place after process f_1 takes place,
while
process g_2 takes place after process f_2 takes place.

Evidently this is true, and it is something we get for free in the diagrammatic language, but in algebraic language we need to explicitly state this.

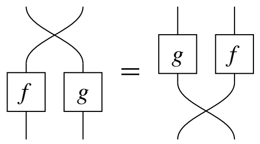
Given the equations we have stated thus far for the ⊗- and ∘-symbols, we can derive many new ones. For example:



can be derived by combining (3.12), (3.14), and (3.16):



while in the diagrammatic language it is again a tautology. Another example of an extra equation for the ⊗- and ∘-symbols involves a crossing:

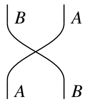


(3.17)

One can easily imagine how many more non-trivial equations are derivable when combining (3.11), (3.12), (3.14), (3.16), and (3.17), many of which would look not at all obvious without drawing the diagram.

Exercise* 3.26 Assume the following equation:

$$\sigma_{A\otimes B,C} = (\sigma_{A,C} \otimes 1_B) \circ (1_A \otimes \sigma_{B,C}) \qquad \text{where} \qquad \sigma_{A,B} :=$$



Prove the first and the last diagram equation from Example 3.6 as algebraic equations using just the above equation and the other algebraic equations introduced in this section.

So why do things become so complicated? In simple terms, one is trying to squeeze something that wants to live in two dimensions (a diagram) into one dimension (a 'linear' algebraic notation). When we do this, the horizontal and the vertical space on the piece of paper we used to draw our diagram suddenly coincide, and we need a bunch of extra syntax (e.g. brackets), to disambiguate the parallel and sequential compositions in their new, 'compressed' world. We then need to subject this extra syntax to a bunch of extra rules to enable the kinds of deformations we were doing quite naturally with diagrams before. The punchline:

$$\boxed{\textbf{Diagrams rule!}}$$

There is a lot more to say about the precise connection between algebraic equations and diagrammatic reasoning, particularly in the context of an area called *(monoidal) category theory*. Although the latter is not crucial to understanding this book, we refer the interested reader to the advanced material in Section 3.6.2.

3.3 Functions and Relations as Processes

We now introduce two simple process theories: the theory of **functions** and the theory of **relations**. Besides it being useful to have some concrete examples of process theories in hand in order to understand some of the concepts to come, these examples establish two important insights:

- that traditional mathematical structures (sets, in this case) can form the types of a process theory, and

- that, even for some fixed system-types, the choice of processes is in fact far more important in determining the character of a process theory. In particular, while the process theories **functions** and **relations** are both based on sets, we shall see in this and the next chapter that their properties as process theories couldn't be farther apart.

For these reasons, these example process theories and, maybe somewhat surprisingly, **relations** in particular, will prove a useful stepping stone towards the process theories of **linear maps** and **quantum maps**.

3.3.1 Sets

To define a process theory, we need to first say what the system-types are, then what the processes are. For both **functions** and **relations**, the system-types are just sets. We will encounter some familiar sets, like the natural numbers $\mathbb{N}$, the real numbers $\mathbb{R}$, the complex numbers $\mathbb{C}$ (whose properties we review in Section* 5.3.1), and:

$$\mathbb{B} := \{0, 1\}$$

which is called the set of *booleans* or *bit values*. Joint system-types are formed by taking the *Cartesian product* of sets, that is:

$$A \otimes B := A \times B = \{(a, b) \mid a \in A, b \in B\}$$

The definition is similar for three or more sets, in which case we replace pairs of elements by *tuples*:

$$A_1 \times \cdots \times A_n := \{(a_1, \ldots, a_n) \mid a_i \in A_i\}$$

If we ignore brackets on tuples of elements, e.g. letting:

$$((a, b), c) = (a, b, c) = (a, (b, c))$$

parallel composition of systems is associative:

$$A \times (B \times C) = (A \times B) \times C$$

as required.

Example 3.27 The set of *bitstrings* of length n can be expressed as an n-fold Cartesian product:

$$\underbrace{\mathbb{B} \times \cdots \times \mathbb{B}}_{n}$$

We can think of this as the set of natural numbers ranging from 0 to $2^n - 1$ by considering the bit strings as a binary representations of these numbers; e.g. for $n = 4$ we represent 0 as $(0, 0, 0, 0)$, 1 as $(0, 0, 0, 1)$, 2 as $(0, 0, 1, 0)$, ..., up to 15, which we represent as $(1, 1, 1, 1)$. This is a trick often used in computer science, and it is also important in this book.

The trivial type I is defined to be the set $\{*\}$ that contains just a single element, here called '$*$'. If we always drop $*$ from tuples, e.g. letting:

$$(a, *) = a = (*, a)$$

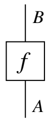
then $\{*\}$ becomes the unit for parallel composition of wires:

$$A \times \{*\} = A = \{*\} \times A$$

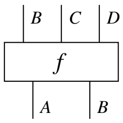
Remark* 3.28 Strictly speaking, the sets $(A \times B) \times C$ and $A \times (B \times C)$, as well as the sets $A \times \{*\}$ and A, are not equal, but *isomorphic*; that is, their elements are in one-to-one correspondence. In fact, they are not just isomorphic but isomorphic in a very strong sense, called *naturally isomorphic*, which pretty much means that for all practical purposes they can be treated as equal. The failure of strict equality is mainly due to the sort of 'algebraic bureaucracy' we complained about in Section 3.2.4, so it won't have any effect on working with diagrams. Section* 3.6.2 contain more details on this somewhat delicate issue.

3.3.2 *Functions*

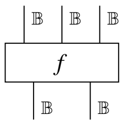
In the process theory of **functions** every process:



is a function from a set A to a set B. Since joint systems are formed using the Cartesian product, a function with many inputs and outputs, e.g.



is a function from the Cartesian product of sets $A \times B$ to the Cartesian product $B \times C \times D$. So f maps pairs $(a, b) \in A \times B$ to triples $(b', c, d) \in B \times C \times D$. More specifically, a function from $\mathbb{B} \times \mathbb{B}$ to $\mathbb{B} \times \mathbb{B} \times \mathbb{B}$ is a function from bit strings of length 2 to bit strings of length 3, for example:



$$:: \begin{cases} (0,0) \mapsto (0,0,0) \\ (0,1) \mapsto (0,1,1) \\ (1,0) \mapsto (1,0,1) \\ (1,1) \mapsto (0,1,1) \end{cases}$$

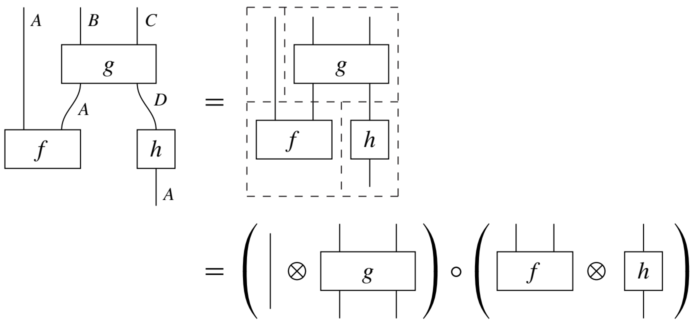
Now that we know what the system-types and processes are, we need to say what 'wiring processes together' means (cf. Definition 3.1). We will do so by specifying what parallel and sequential composition for functions means. Sequential composition is the usual composition of functions:

$$(g \circ f)(a) := g(f(a)) \tag{3.18}$$

Parallel composition is defined as applying each function to its corresponding element of a pair:

$$(f \otimes g)(a, b) := (f(a), g(b)) \tag{3.19}$$

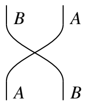
This tells us everything we need to know to compute the function represented by a circuit diagram, since we can decompose the diagram across $\otimes$ and $\circ$:



$$= (1_A \otimes g) \circ (f \otimes h)$$

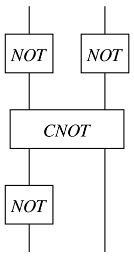
then applying equations (3.18) and (3.19). There might be more than one way to decompose a diagram, but the actual function computed will always be the same. Note that this will work for any diagram where the wires do not cross. To handle crossings, we need to define a ‘swap’ function.

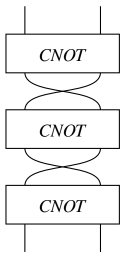
Exercise 3.29 Which function represents the swap:



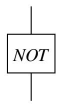
where A and B can be any set?

Exercise 3.30 Compute the functions:

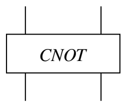




where:



$$:: \begin{cases} 0 \mapsto 1 \\ 1 \mapsto 0 \end{cases}$$

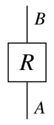


$$:: \begin{cases} (0,0) \mapsto (0,0) \\ (0,1) \mapsto (0,1) \\ (1,0) \mapsto (1,1) \\ (1,1) \mapsto (1,0) \end{cases}$$

After reading this book, you will be able to straightforwardly write down the answer for each of these two diagrams (not from memory, but because the calculation is so easy with the tools we will provide!).

3.3.3 Relations

In the process theory of **relations** every process:



is a *relation* from a set A to a set B, that is, a subset:

$$R \subseteq A \times B$$

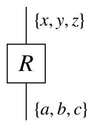
We say a and b are *related* by R if $(a, b) \in R$. It will be useful to use a more function-like notation, writing:

$$R :: a \mapsto b \qquad \text{instead of} \qquad (a, b) \in R \tag{3.20}$$

and as a counterpart to the notation $f(a)$ for functions, we can set:

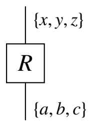
$$R(a) := \{b \mid R : a \mapsto b\}$$

Thus relations that happen to be functions can be expressed in the same notation as before. However, for more general relations, we are allowed to map a single element in the input to any number of elements in the output. For example:



$$:: \begin{cases} a \mapsto x \\ a \mapsto y \\ b \mapsto z \end{cases}$$

which we can also write as:



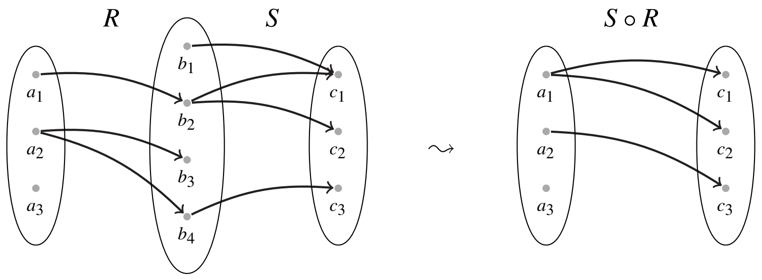
$$:: \begin{cases} a \mapsto \{x, y\} \\ b \mapsto z \\ c \mapsto \emptyset \end{cases}$$

Interpreted as a process, a relation is like a function with added *non-determinism*: a single input can map to several outputs or none at all.

Relations are composed sequentially as follows: an element a is related to c by $(S \circ R)$ if and only if there exists some b such that $R :: a \mapsto b$ and $S :: b \mapsto c$. Symbolically:

$$\begin{array}{c}|\\ \boxed{S}\\ |\\ \boxed{R}\\ |\end{array} :: a \mapsto c \iff \exists b \left(\begin{array}{c}|\\ \boxed{R}\\ |\end{array} :: a \mapsto b \text{ and } \begin{array}{c}|\\ \boxed{S}\\ |\end{array} :: b \mapsto c \right) \tag{3.21}$$

Note how this ‘if and only if’ expression is being used to define a new relation; that is, it tells us precisely which elements a and c are related by $S \circ R$. You may have seen a picture of relation composition in school that looks something like this:

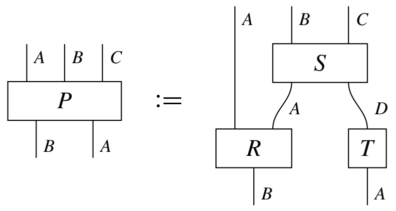


Here, we can see that elements are related by $S \circ R$ if and only if there is a path connecting them in the diagram on the left.

Parallel composition of relations is defined as follows:

$$\begin{array}{c}|\\ \boxed{R}\\ |\end{array}\ \begin{array}{c}|\\ \boxed{S}\\ |\end{array} :: (a,b) \mapsto (c,d) \iff \left(\begin{array}{c}|\\ \boxed{R}\\ |\end{array} :: a \mapsto c \text{ and } \begin{array}{c}|\\ \boxed{S}\\ |\end{array} :: b \mapsto d \right) \tag{3.22}$$

As in the case of functions, we can now compute a relation represented by a diagram by decomposing it over $\otimes$ and $\circ$ then applying (3.21) and (3.22), but this can be quite a bit of work. Luckily, there is a general procedure for computing diagrams of relations directly. Here’s how it goes: suppose we have some known relations R, S, and T, and we want to compute a new relation P as their composition:



(3.23)

Step 1: Write P as a diagram formula:

$$P^{A_2B_2C_1}_{B_1A_1} = R^{A_2A_3}_{B_1} S^{B_2C_1}_{A_3D_1} T^{D_1}_{A_1}$$

Step 2: Change wire names to elements of the appropriate set. For example, write B_2 as an element $b_2 \in B$:

$$P^{a_2b_2c_1}_{b_1a_1} \iff R^{a_2a_3}_{b_1} S^{b_2c_1}_{a_3d_1} T^{d_1}_{a_1}$$

Step 3: Change box names to actual relations that map the element(s) in the subscript to the element(s) in the superscript, and adjoin '$\exists x$' for every repeated element x:

$$P :: (b_1, a_1) \mapsto (a_2, b_2, c_1) \iff \exists a_3 \exists d_1 \left(\begin{array}{l} R :: b_1 \mapsto (a_2, a_3) \text{ and} \\ S :: (a_3, d_1) \mapsto (b_2, c_1) \text{ and} \\ T :: a_1 \mapsto d_1 \end{array} \right)$$

In the absence of inputs and/or outputs, one must use '$*$', i.e. the unique element in the set $I = \{*\}$.

Step 4: The expression in the RHS of '$\iff$' now uniquely characterises all related tuples in the LHS, and hence can now be used to compute P.

For the particular cases of sequential and parallel composition of relations, we recover (3.21) and (3.22), respectively. To see this, just draw the diagrams of $f \otimes g$ and $g \circ f$, apply steps 1–3, and see what comes out.

Exercise 3.31 Suppose A, B, C, and D are the following sets:

$$\begin{aligned} A &= \{a_1, a_2, a_3\} \\ B &= \mathbb{B} \\ C &= \{\textbf{red}, \textbf{green}\} \\ D &= \mathbb{N} \end{aligned}$$

Compute P as defined by (3.23), for R, S, T defined as follows:

$$R :: \begin{cases} 1 \mapsto (a_1, a_1) \\ 1 \mapsto (a_1, a_2) \end{cases} \qquad S :: \begin{cases} (a_1, 5) \mapsto (0, \textbf{red}) \\ (a_1, 5) \mapsto (1, \textbf{red}) \\ (a_2, 6) \mapsto (1, \textbf{green}) \end{cases} \qquad T :: \begin{cases} a_1 \mapsto 200 \\ a_3 \mapsto 5 \end{cases}$$

and for R, S, T defined as follows:

$$R :: \begin{cases} 0 \mapsto A \times \{a_2, a_3\} \\ 1 \mapsto A \times \{a_2, a_3\} \end{cases} \qquad S :: \begin{cases} (a_1, 0) \mapsto \mathbb{B} \times \{\textbf{red}, \textbf{green}\} \\ (a_1, 1) \mapsto \mathbb{B} \times \{\textbf{red}, \textbf{green}\} \\ (a_1, 2) \mapsto \mathbb{B} \times \{\textbf{red}, \textbf{green}\} \\ \quad \vdots \end{cases} \qquad T :: \begin{cases} a_1 \mapsto \mathbb{N} \\ a_2 \mapsto \mathbb{N} \\ a_3 \mapsto \mathbb{N} \end{cases}$$

Remark* 3.32 Due to the close resemblance between the process theories **relations** and **linear maps**, and in particular, the fact that they both admit a *matrix calculus* (see Chapter 5), this 'trick' for computing diagrams of relations applies also for computing diagrams of linear maps, as we shall see in Section 5.2.4, and hence will also be relevant for **quantum processes**. In fact, if we treat relations as matrices with boolean entries, the computation is essentially identical.

3.3.4 Functions versus Relations

When we restrict the definitions (3.21) and (3.22) to functions, we obtain (3.18) and (3.19), respectively. In both cases, we took advantage of the fact that functions relate each input element to a unique output to obtain a simpler form. For example, in the case of parallel composition of functions $f : A \to C$ and $g : B \to D$, for each $a \in A$ and each $b \in B$ there is a unique $c \in C$ and $d \in D$ such that $f :: a \mapsto c$ and $g :: b \mapsto d$, and hence, building $f \otimes g$ amounts to merely pairing these two elements as we did in (3.19). On the other hand, for relations, $R \otimes S :: (a, b) \mapsto (c, d)$ whenever there exist both a $c \in C$ and a $d \in D$ such that $R :: a \mapsto c$ and $S :: b \mapsto d$, and there could of course be more than one of such pair (c, d).

Now, since:

1. both **functions** and **relations** have sets as system-types, so in particular, the system-types of **relations** include those of **functions**;
2. functions are a special case of relations, and hence the processes of **relations** include the processes of **functions**; and
3. sequential composition $\circ$ of **functions** is a special case of sequential composition of **relations**, and parallel composition $\otimes$ of **functions** is a special case of parallel composition of **relations**,

we say that the process theory of **functions** is a *subtheory* of **relations**. We can write this as follows:

$$\textbf{functions} \subseteq \textbf{relations}$$

Of course, there are many more relations than there are functions, and throughout this book we will encounter many important relations that aren't functions. In fact, these extra relations are precisely what make the process theory of **relations** behave much more like **linear maps** than **functions**.

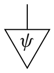
3.4 Special Processes

In this section, we single out several types of process that will play a central role throughout this book.

3.4.1 States, Effects, and Numbers

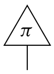
So far, we have been handling boxes without any inputs and/or without any outputs just like any other boxes. However, these particular boxes have a very special status when we interpret them as processes.

- *States* are processes without any inputs. In operational terms they are 'preparation procedures'. We represent them as follows:



Whereas a system A could be in any number of possible different states, a preparation procedure will produce a system of type A in a particular state ψ taken from these possibilities. The fact that there is no wire in means we don't know (or care) where this system came from, just that we have it now and it is in a particular state. For example, if we say 'here is a bit initiated in the 0 state', then we don't really care about the previous history of that bit as long as we have the guarantee that it is currently in state 0 and that it is available to perform some further computations. Or, in the case of saying 'here is a fresh raw potato', we don't really care whose farm it came from or which animal's manure was used, as long as we know that we have a fresh raw potato that we can process further into fries.

- *Effects* are processes without any outputs. We have borrowed this terminology from quantum theory, where effects play a key role. From now on we represent them as follows:



The notion of effect is dual to the notion of state, in that one starts with a system and has nothing left afterwards – or, we simply don't care what we have afterwards. The simplest example of an effect consists of *discarding* a system, where the system is destroyed (or simply disregarded). Less trivial effects refer to things that may or may not happen, depending on the actual state of a system. For example, 'the bomb exploded', 'the photon was absorbed', 'all the food was eaten', or 'the dodo became extinct' could all be considered effects, since afterwards there is no bomb, no photon, no food, and no dodo.

Operationally speaking, effects are used to model 'tests'. When we say an effect 'happens', it means we tested whether a system satisfies a certain property and have obtained 'yes' as the answer, after which we discard that system, or – as is often the case – the system is destroyed in the verification process. Altogether, a test consists of:

1. a question about a system,
2. a procedure that enables one to verify this question, and
3. the event of obtaining 'yes' as the answer; i.e. the effect 'happened'.

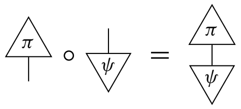
So, there is more to the notion of a test than just answering a question. For example, when making a drawing, one could ask the question: 'Is this a red pen?' A corresponding test would then consist of taking a piece of paper, then writing with the pen on it, and observing that the colour is indeed red. Of course, in this case one would (usually) not destroy the pen. Genuine destruction takes place when one asks the question: 'Is this a working match?' A corresponding test would then consist of trying to light the match and succeeding, after which we end up with no match. The act of discarding a system is also a test, albeit a trivial one. After verifying some property that holds for any state of a system (e.g. ‘does this system exist?’), we discard it. This is of course a very silly test, since no information is gained, but the system is lost.

At first glance, it seems like tests are more than just ‘preparation procedures in reverse’. However, if we assume that preparation procedures can fail (as is often the case in the lab), we see that the notions really are symmetric. A preparation procedure consists of:

1. a state we would like our system to be in,
2. a procedure that enables one to put a system in that state, and
3. the event of obtaining ‘yes’; i.e. the state was successfully prepared.

Of course, unlike tests, we could just keep retrying until a preparation succeeds (or we run out of funding), so we tend to disregard part 3.

When we compose a state and an effect, a third kind of special box arises:



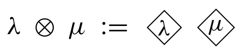
- *Numbers* are processes without any inputs or outputs. From now on we represent them as:

  or sometimes simply as:

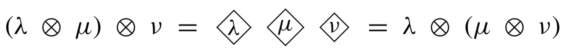
$$\lambda$$

At this point, you’re probably saying ‘Hang on! I already know what numbers are, and this doesn’t have anything to do with processes with no inputs and outputs!’ Well, the numbers you learned about in school (e.g. whole numbers, real numbers, possibly complex numbers) are all instances of this very general kind of ‘number’.

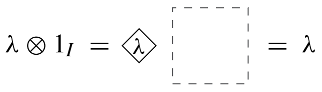
So, what are processes with no inputs or outputs? Well, they are just a set of things that can be ‘multiplied’, thanks to ⊗-composition (or equivalently in this case, ∘-composition). That is, if λ and μ are numbers, then:



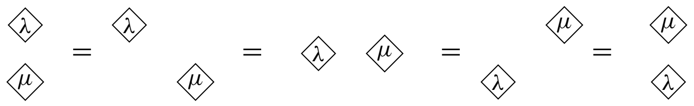
is also a number. Also, we saw in Section 3.2.4 that diagrams rule, so this ‘multiplication’ is associative:



and has a unit, given by the empty diagram:



And finally, because of the lack of inputs and outputs numbers can move around liberally within a diagram, and even can do a little dance:

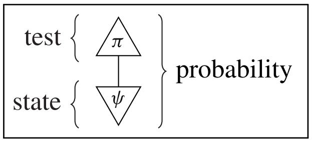


(3.24)

This little dance shows that multiplication of numbers is furthermore *commutative*. So, what we call 'numbers' consists of a set of things that can be 'multiplied', and that multiplication operation is associative, commutative, and has a unit (i.e. a chosen number that means '1'). Sound familiar? Mathematicians call this a *commutative monoid*. Other people would call this 'virtually any kind of numbers you can imagine.'

Remark* 3.33 This remarkably simple little proof shown in (3.24) is known as the *Eckmann–Hilton argument*. It shows in general that, whenever we have a pair of associative operations (in this case $\otimes$ and $\circ$, applied specifically to numbers) that share the same unit (1_I in both cases) and satisfy the interchange law (3.16) (which of course comes for free with diagrams), both of these operations are commutative and are in fact the same operation. Here, we used it to prove that $\otimes$ (or equivalently $\circ$) gives us a commutative 'multiplication' operation for the numbers of any process theory.

So, how should we interpret the fact that, when a state meets an effect, a number pops out? One extremely useful interpretation is that this number gives the *probability* that the effect happens, given the system is in a particular state. Or, in terms of tests, it is the probability that when we test the state ψ for an effect π we get 'yes' as the answer:



(3.25)

We refer to this as the *generalised Born rule*. Here, we say 'generalised' because it makes sense in any process theory. We will see in Chapter 6 that by restricting to the theory of **quantum maps**, we obtain the rule that is used to compute all probabilities within quantum theory, which is called simply the *Born rule*.

Remark* 3.34 A reader familiar with quantum theory may think that rather than a probability, in (3.25) we obtain a complex number (a.k.a. an 'amplitude'), which we then have to multiply with its conjugate in order to obtain a probability. This is indeed the case if we are dealing with plain old **linear maps**. However, **quantum maps** have this step 'built in', so the numbers are always positive real numbers. In Example 3.37 below we indicate how (3.25) indeed produces the correct probabilities in quantum theory, and in Section 6.1.1 we explain how the passage from complex numbers to probabilities, via multiplication of a number with its conjugate, is nicely built in to the theory of **quantum maps**.

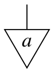
In many physical theories, and quantum theory in particular, states are thought of as elements in some set (or space), and processes as functions between sets and/or spaces.

Numbers are yet again another thing, and effects in many cases don't even seem to have a clear counterpart. However, in the diagrammatic language, we treat states, effects, and numbers all on the same footing as special cases of processes. This is very convenient because we can define many concepts for arbitrary processes that will apply immediately to all of these special cases.

We will now see how all of these special kinds of processes look in the process theories we have encountered so far.

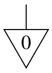
Example 3.35 (**functions**)

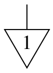
1. *States correspond to the elements of a set*. Since 'no wire' means the single element set $\{*\}$ (see Section 3.3.1), a state is a function from $\{*\}$ into another set A. This function 'points to' a single element $a \in A$, namely, the image of $*$:



$$:: * \mapsto a$$

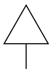
   Conversely, for each $a \in A$, there is a unique function that sends $*$ to a, so elements of A and functions from $\{*\}$ into A are essentially the same thing, which justifies us simply calling this function a. If we let $A := \mathbb{B}$, there are exactly two states:



$$:: * \mapsto 0$$


$$:: * \mapsto 1$$

2. *Effects are boring*. For any set A, there is exactly one function from A to $\{*\}$, namely, the function that sends everything to $*$. Therefore, we shouldn't even bother to give it a name. Again letting $A := \mathbb{B}$ this unique effect is:


$$:: \begin{cases} 0 \mapsto * \\ 1 \mapsto * \end{cases}$$

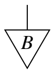
3. *Numbers are boring* – for the same reason that effects are boring. Since a number is a special case of an effect, there is only one number, namely, the function that sends $*$ to $*$. This unique number is already represented by the empty diagram, so we don't even need to draw it!

So in summary, states of type A are the elements of A, but there isn't enough freedom with functions to get interesting effects and numbers.

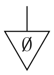
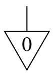
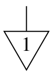
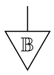
Let's see what happens if we generalise to relations.

Example 3.36 (**relations**)

1. *States correspond to subsets of a set (a.k.a. 'non-deterministic' elements).* A state is a relation from the single element set $\{*\}$ to another set A, thus it will relate $*$ to zero or more elements of A. Just as we named states in **functions** by set elements, we can name states in **relations** by the subset $B \subseteq A$ that $*$ 'points to':

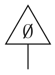
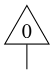
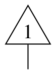
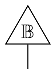
 $:: * \mapsto B$

When $A := \mathbb{B}$ there are four states in **relations** corresponding to each of the four subsets of $\{0, 1\}$:

 $:: * \mapsto \emptyset$  $:: * \mapsto \{0\}$  $:: * \mapsto \{1\}$  $:: * \mapsto \mathbb{B}$

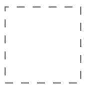
As with **functions**, we have the states 0 and 1, which correspond to the system being definitely in each of these respective states. We can interpret the state $\mathbb{B}$ as the system being in either of the two states 0 or 1. The state $\emptyset$ corresponds to the system being in neither 0 nor 1, i.e. an ‘impossible’ state of the system.

2. *Effects also correspond to subsets*. Relations from a set A to $\{*\}$ are uniquely defined by the subset $B \subseteq A$ consisting of all $a \in A$ where $a \mapsto *$. For $A := \mathbb{B}$ all the possible effects are:

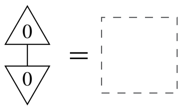
 $:: \begin{cases} 0 \mapsto \emptyset \\ 1 \mapsto \emptyset \end{cases}$  $:: \begin{cases} 0 \mapsto \{*\} \\ 1 \mapsto \emptyset \end{cases}$  $:: \begin{cases} 0 \mapsto \emptyset \\ 1 \mapsto \{*\} \end{cases}$  $:: \begin{cases} 0 \mapsto \{*\} \\ 1 \mapsto \{*\} \end{cases}$

The effects 0 and 1 can be interpreted as tests that verify if the system is in the state 0 or 1, respectively. The effect $\mathbb{B}$ can be interpreted as the trivial test that gives ‘yes’ as the answer for any of the three ‘possible’ states 0, 1, and $\mathbb{B}$, whereas the effect $\emptyset$ is the ‘impossible’ effect, which will never happen.

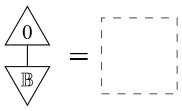
3. *There are two numbers corresponding to ‘impossible’ and ‘possible’*. There are two relations from the set $\{*\}$ to itself, namely the empty relation and the identity relation:

$\emptyset :: * \mapsto \emptyset$  $:: * \mapsto \{*\}$

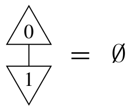
We interpret these two numbers as impossible and possible, which makes perfect sense when we look at what happens when we test whether the system is in a particular state. First consider:



If the system is in the state 0 it should be *possible* to get a positive outcome when testing for effect 0. Of course it’s possible, and in fact this should always happen! However, as there is no number ‘certain’ in **relations**, ‘possible’ is the most we can get. Next, consider:



If the state is either 0 or 1, then it is *possible* (but no longer certain) that testing for 0 will yield a positive outcome. Finally consider:

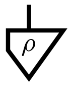


If the system is in the state 0, then it is indeed *impossible* for the test 1 to give a positive outcome.

The following starred example uses some notation that will be introduced over the next two chapters. However, those already familiar with quantum theory may find it useful. Other readers should skip this example for now.

Example* 3.37 (**quantum maps**) Quantum maps give a particularly elegant way to express completely positive maps, of which mixed quantum states, quantum effects, and probabilities are all special cases. Most of Chapter 6 is devoted to them, but here we provide a first glimpse. For the benefit of those already familiar with quantum theory, we use traditional terminology and notation here, rather than the more process-theoretic versions introduced in Chapter 6.

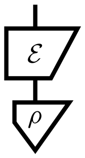
1. *States correspond to density operators.* These are depicted as:



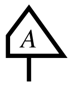
The action of a *completely positive map* $\mathcal{E}$ on a state ρ, that is:

$$\mathcal{E} :: \rho \mapsto \sum_i A_i^\dagger \rho A_i$$

is depicted as:



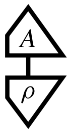
2. *Effects are positive linear functionals.* These are depicted as:



for some positive operator A. Their action on a state ρ, that is:

$$\rho \mapsto \mathrm{tr}(A\rho)$$

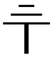
(where tr is the trace) is depicted as:


(3.26)

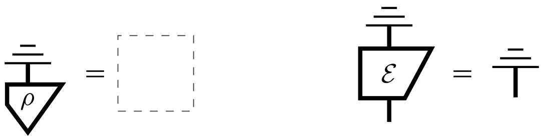
A special case is the trace itself:

$$\rho \mapsto \mathrm{tr}(\rho)$$

which represents 'discarding the system' and is depicted as:



The conditions of being trace-1 (for states) and trace-preserving (for completely positive maps) are written respectively as:

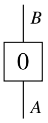


3. *Numbers are positive reals.* The Born rule (3.25) now takes the form (3.26), and it indeed produces positive reals, which are interpreted as probabilities.

What may surprise the reader is that density matrices are depicted as states, i.e. no input, rather than as input-output processes (cf. density 'operator') and that completely positive maps are depicted as input-output processes, rather than as something that takes input-output processes to input-output processes (cf. 'superoperator'). This is a consequence of the manner in which we define *quantum maps*, and this is of course an upshot, since now the diagrams reflect the true nature of quantum states/effects/processes.

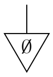
3.4.2 Saying the Impossible: Zero Diagrams

An extreme example of (non-function) relations are *zero relations*. These are relations in which 'nothing relates to nothing'; i.e. there are no $a \in A$ and $b \in B$ such that $R :: a \mapsto b$, or equivalently:

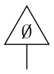


$:: a \mapsto \emptyset$

We already encountered a few of these, namely:



$:: * \mapsto \emptyset$



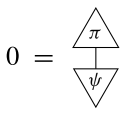
$:: \begin{cases} 0 \mapsto \emptyset \\ 1 \mapsto \emptyset \end{cases}$

$\emptyset :: * \mapsto \emptyset$

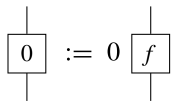
Each of these represented something 'impossible'.

Since an effect represents a test in which we successfully verify a certain question, it may be the case that for certain states this effect is simply impossible. So what do we get if we compose that state and that effect in a diagrammatic language? We should of course get a number that corresponds to ‘impossible’. These impossible numbers, and more generally impossible processes, have an elegant characterisation within a process theory.

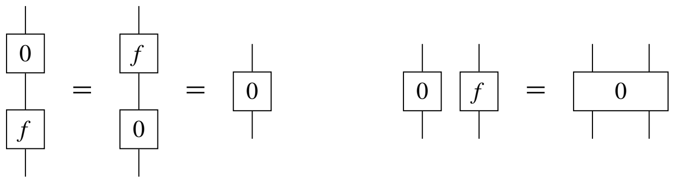
For the specific case of **relations**, it is easily seen that both the sequential and parallel composition of a zero relation with any relation is again a zero relation. For general process theories, assuming that an effect π is impossible given a state ψ, the number:



should represent the impossible. Of course, if part of a larger process is impossible, then the entire process is impossible, hence:



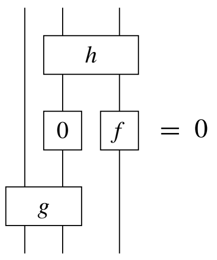
should also be impossible for any f. So if we want to accommodate the impossible in our language, for every possible input and output type there should be a *zero process*, which obeys the following composition rules:



(3.27)

Exercise 3.38 Prove that if a zero process satisfying (3.27) exists, then for any given types of input and output systems, it is unique.

So just as multiplying by the plain old number zero always yields zero, zero processes ‘absorb’ any diagram in which they occur. In other words, any diagram containing a zero process is itself a zero process. Due to the uniqueness of the zero process, we will just write it as ‘0’ and ignore its input and output wires:



The main point to remember here is that 0 is not the empty diagram, since the empty diagram can be adjoined to any diagram without changing it. Zeros, on the other hand, 'eat' everything around them!

3.4.3 Processes That Are Equal 'Up to a Number'

There will be many instances in this book where numbers are crucially important, most notably when the Born rule (3.25) is used to compute probabilities. However, on other occasions, they are just a bit of a nuisance. For example, we may only be interested in some qualitative aspect of a process, such as whether it separates into disconnected pieces (cf. Section 4.1.1 in the next chapter), in which case numbers play no role. In other cases, we can make life a bit easier by ignoring numbers throughout a calculation and figuring out at the end what they should be. Therefore, we'll introduce some notation to handle these kinds of situations:

Definition 3.39 Two processes are *equal up to a number*, written:

$$\boxed{f} \approx \boxed{g}$$

if there exist non-zero numbers λ and μ such that:

$$\lambda \boxed{f} = \mu \boxed{g} \tag{3.28}$$

We require λ and μ to be non-zero because otherwise everything would be $\approx$-related to everything else:

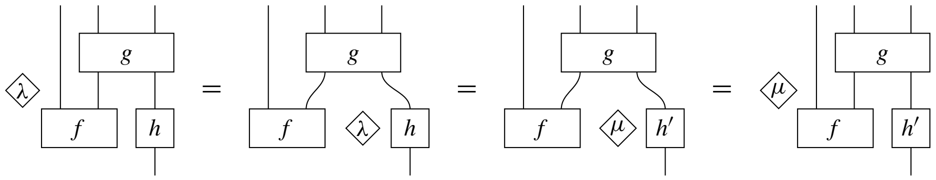
$$0 \boxed{f} = 0 = 0 \boxed{g}$$

which is not particularly useful. What should instead be true is that the only thing ≈ 0 is zero itself.

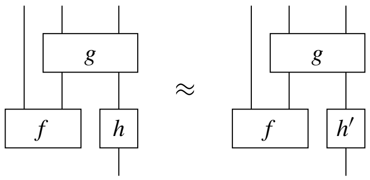
Exercise 3.40 Assume a process theory has *no zero-divisors*; i.e. for all processes f and numbers λ we have $\lambda f = 0$ if and only if $\lambda = 0$ or $f = 0$. Then show that:

$$\boxed{f} \approx 0 \quad \Longrightarrow \quad \boxed{f} = 0$$

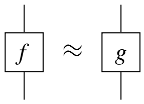
A fortunate feature of the relation $\approx$ is that it plays well with diagrams. Suppose for example that $h \approx h'$, then:



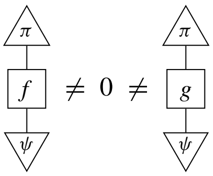
So, for any diagram containing h, we have:



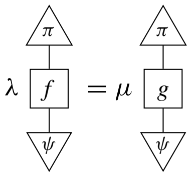
Even if we ignore numbers in a calculation, we might still recover them at the end. Suppose we have established that:



If there exists a state ψ and an effect π such that:



then by sandwiching both f and g in equation (3.28) we obtain:

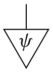


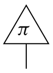
where all numbers are non-zero. As a consequence, if the numbers of a process theory are not too crazy, then we can figure out what λ and μ must be in order to obtain an equality.

3.4.4 Dirac Notation

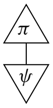
Throughout this chapter we have extolled the virtues of diagrams, most coming from the freedom to write down processes in two dimensions. With modern technology, these diagrams are pretty easy to stick in, say, a textbook. However, things weren't always this good. Let's see now how far we get if we try to turn diagrams into a notation that can easily be churned out on a good old-fashioned `typewriter`.

Our initial conventions for states and effects are as follows:

D1:  is written as $|\psi\rangle$ and called a 'Dirac ket'

D2:  is written as $\langle\pi|$ and called a 'Dirac bra'

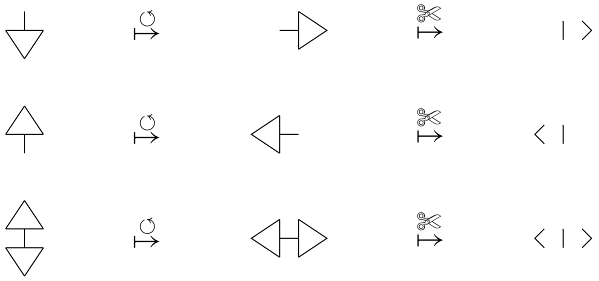
We can now also consider compositions of a state and an effect:

D3:  is written as $\langle\pi|\psi\rangle$ and called a 'Dirac bra(c)ket'

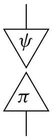
As our naming indicates this notation was introduced by a physicist called Paul Dirac, specifically for the purpose of describing quantum theory. It is still widely used in most textbooks on quantum theory today.

Remark 3.41 We have slightly deviated from the usual Dirac notation since we are missing one essential refinement of the diagrammatic language to give a complete account here, namely the ability to turn a ket into a bra. We will do this in Section 4.3.3 where we provide modified versions of **D2** and **D3**.

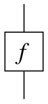
Note that there's a pretty easy recipe for turning diagrams of states and effects into Dirac notation:

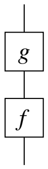


So, up to some cutting and rotating, Dirac notation is actually a (one-dimensional) subset of our diagrammatic notation. With that in mind, let's continue our game of fitting diagrams on a line:

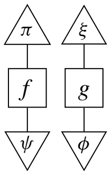
D4:  is written as $|\psi\rangle\,\langle\pi|$

For processes that are not states or effects, we cut the whole box off and write sequential composition just by writing processes from right to left:

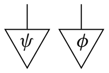
D5:  is written as f

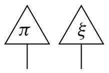
D6:  is written as gf

Then, exploiting the fact that for numbers λ, μ we have that $\lambda \otimes \mu = \lambda \circ \mu$, we can obtain expressions like:

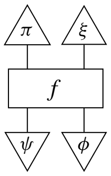
Ex:  can be written as $\langle \pi | f | \psi \rangle \langle \xi | g | \phi \rangle$

But things become of course much trickier when wires and boxes are genuinely side by side, for example, when considering boxes with two wires in and/or out. We can make some progress by writing multiple states or effects inside of a ket or bra to indicate parallel composition:

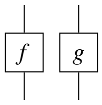
D8:  is written as $|\psi\phi\rangle$

D9:  is written as $\langle \pi\xi |$

Hence we obtain:

Ex:  which can be written as $\langle \pi\xi | f | \psi\phi \rangle$

But once the diagrams become more involved, we will have to make parallel composition explicit using '$\otimes$' or employing some other trickery, like letting $f_A := f \otimes 1_B$ and $g_B := 1_A \otimes g$ then setting:

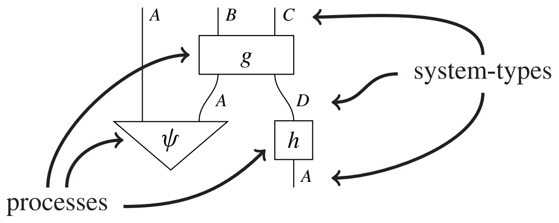
D10:  can be written as $f_A g_B$ $(= g_B f_A)$

Of course, this is ambiguous unless we already know that we are working with some system $A \otimes B$, and pretty soon we need to start requiring extra equations, as in Section 3.2.4.

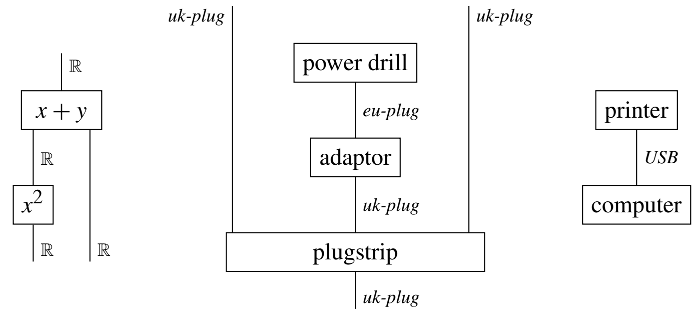
By sticking to diagrams all of this is immediately solved, and fortunately, very few of us are still using `typewriters`. Technology marches on, and now there are some great tools for drawing diagrams (pretty much) as easily as typing out a Dirac-style formula on a keyboard.

3.5 Summary: What to Remember

1. *Diagrams* consist of boxes that are labelled by *processes* and wires that are labelled by *system-types*:

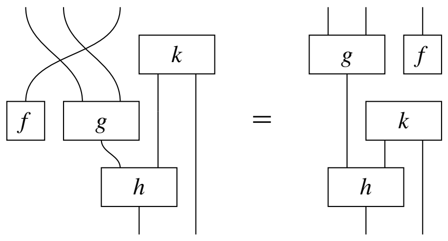


Diagrams are all around us:

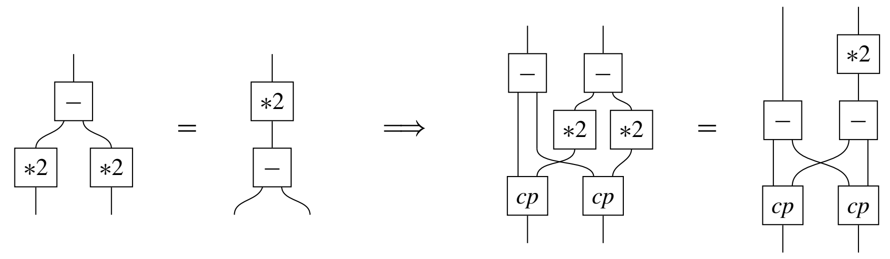


We can obtain new *diagram equations* from old ones in two ways:

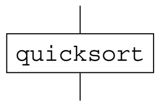
1. *Diagram deformation*:



2. *Diagram substitution*:



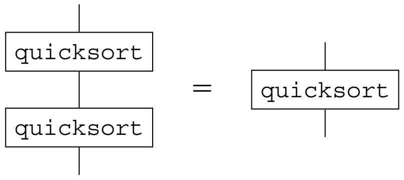
2. A *process theory* is a collection of processes that can be plugged together. It tells us how to interpret the boxes and wires in a diagram, e.g.:



:=

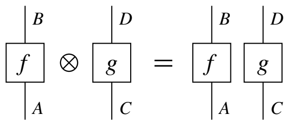
```
qs [] = []
qs (x :: xs) =
   qs [y | y <- xs; y < x] ++ [x] ++
   qs [y | y <- xs; y >= x]
```

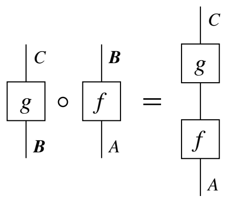
and what it means to 'wire processes together'. By doing so it also tells us which diagrams represent the same process, e.g.:



Examples are **functions**, **relations**, **linear maps** (defined in Chapter 5), **classical processes** (defined in Chapter 8), **quantum maps** (defined in Chapter 6), and **quantum processes** (also defined in Chapter 8).

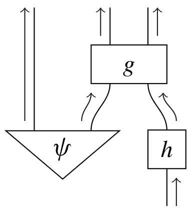
3. *Parallel composition* '⊗' and *sequential composition* '∘' are defined as:



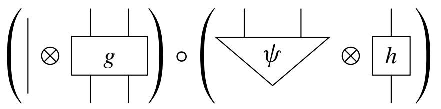


4. *Circuit diagrams* are an important class of diagrams. They have the following two equivalent characterisations:

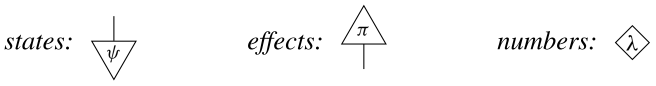
The diagram contains no directed cycles. That is, information flows from bottom to top without 'feeding back' to an earlier process:



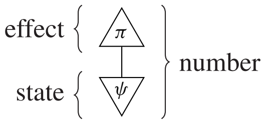
The diagram can be constructed by composing boxes, including identities and crossings, by means of the operations ⊗ and ∘:



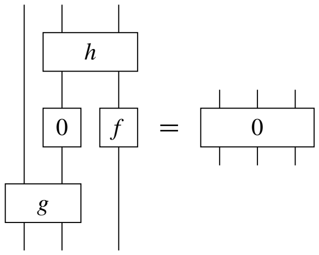
5. Processes lacking inputs, outputs, or both have special names:



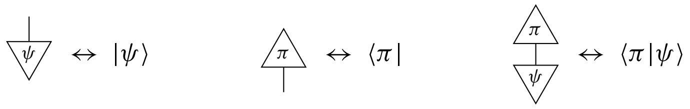
These special processes admit a *generalised Born rule*:



6. *Zero processes* 'eat everything':



7. *Dirac notation* is common language for depicting quantum processes. It is a fragment of the diagrammatic language:



3.6 Advanced Material*

Central to this book are process theories, which tell us how to interpret wires and boxes in a diagram, and what it means to form diagrams, i.e. what it means to wire boxes together. Here we present two alternative ways of defining process theories that make no direct reference to diagrams, namely, *abstract tensor systems* and *symmetric monoidal categories*.

In fact, we already saw a glimpse of each of these, respectively, when we encountered diagram formulas in Section 3.1.3 and when we defined circuits by means of the operations parallel and sequential composition in Section 3.2.

So how does one go about constructing a symbolic counterpart to the definition of process theories? In the first two parts of the definition:

(i) a collection T of *system-types* represented by wires,
(ii) a collection P of *processes* represented by boxes, where for each process in P the input types and output types are taken from T ...

diagrams don't play an essential role and we can just drop the 'represented by ...' bit, provided that we explicitly require that each process comes with a list of input types and a list of output types, all taken from T. The third part of the definition, which concerns what 'forming diagrams' means:

(iii) a means of 'wiring processes together', that is, an operation that interprets a diagram of processes in P as a process in P

is the one that requires some non-trivial effort. In particular, we need to provide a symbolic counterpart to 'wiring processes together' that does not make reference to diagrams. Moreover, the symbolic operation(s) should produce processes of that theory.

3.6.1 Abstract Tensor Systems*

We introduced diagram formulas as a symbolic counterpart to diagrams. Can we define a corresponding notion of process theory? The definition below does exactly that. For notational simplicity we will assume that all system-types are the same, and consequently, distinct wire names are merely distinguishing occurrences of the same system.

Definition 3.42 An abstract tensor system (ATS) consists of:

1. For all wire names $A_1, \ldots, A_m$ and $B_1, \ldots, B_n$, a set of *tensors*:

$$f_{A_1 \ldots A_m}^{B_1 \ldots B_n} \in \mathcal{T}(\{A_1, \ldots, A_m\}, \{B_1, \ldots, B_n\})$$

where $A_1, \ldots, A_m$ are called *inputs* and $B_1, \ldots, B_n$ are called *outputs*.
2. A *unit* tensor:

$$1 \in \mathcal{T}(\{\}, \{\})$$

and for all wire names A and B *identity* tensors:

$$\delta_A^B \in \mathcal{T}(\{A\}, \{B\})$$

3. An operation *tensor product* that combines tensors provided their inputs and outputs are distinct, which is denoted by concatenation:

$$f_{A_1 \ldots A_m}^{B_1 \ldots B_n} g_{C_1 \ldots C_k}^{D_1 \ldots D_l} \in \mathcal{T}(\{A_1, \ldots, A_m, C_1, \ldots, C_k\}, \{B_1, \ldots, B_n, D_1, \ldots, D_l\})$$

4. For any wire names A and B, an operation c_A^B called *tensor contraction*, which is denoted either explicitly or by repeating an upper/lower wire name:

$$f_{A_1\ldots A_m}^{B_1\ldots B_{j-1}A_iB_{j+1}\ldots B_n} := c_{A_i}^{B_j}\left(f_{A_1\ldots A_m}^{B_1\ldots B_n}\right)$$

which, intuitively, 'connects' output B_j to input A_i.

5. *Re-indexing* operations that change wire names:

$$f_{A_1\ldots A_m}^{B_1\ldots B_n}[A_i \mapsto A_i'] = f_{A_1\ldots A_{i-1}A_i'A_{i+1}\ldots A_m}^{B_1\ldots B_n}$$

where these operations satisfy the following conditions:

1. the tensor product is associative, commutative, and has unit 1:

$$(fg)h = f(gh) \qquad fg = gf \qquad 1f = f$$

2. the order of tensor contractions/products is irrelevant:

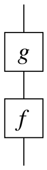
$$c_A^B(c_C^D(f)) = c_C^D(c_A^B(f)) \qquad c_A^B(f)g = c_A^B(fg)$$

3. contracting with the identity does nothing except change wire names:

$$c_A^B(\delta_C^B f_{\ldots A\ldots}^{\cdots}) = f_{\ldots C\ldots}^{\cdots} \qquad c_B^A(\delta_B^C f_{\cdots}^{\ldots A\ldots}) = f_{\cdots}^{\ldots C\ldots}$$

4. re-indexing respects identities, tensor product, and contraction.

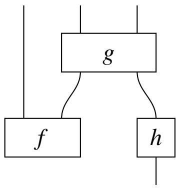
The two operations, tensor product and tensor contraction, together play the role of 'wiring processes together'. For example, the symbolic counterpart to the diagram:



is obtained as follows:

$$c_{B'}^{B}\left(f_A^B g_{B'}^C\right)$$

More generally, we can compute the tensor corresponding to a diagram by first writing it down as a diagram formula and then decomposing it further in terms of the tensor product and tensor contraction. For example:



$$\rightsquigarrow \quad f^{AB}h_C^D g_{BD}^{EF} := c_{D'}^{D}\left(c_{B'}^{B}\left(f^{AB}\left(h_C^D g_{B'D'}^{EF}\right)\right)\right)$$

An expression like $f^{AB}h_C^D g_{BD}^{EF}$ is called *abstract tensor notation*. Just like the decomposition of diagrams into $\otimes$ and $\circ$, the expression above is not unique. However, the requirements in Definition 3.42 guarantee that any two such decompositions will be equal.

Just like the decomposition of diagrams into $\otimes$ and $\circ$ we met in Section 3.2, here 'wiring processes together' is decomposed into two suboperations. This notation has been employed extensively in mathematical physics (and especially general relativity), where one often deals with operations with many inputs or outputs that are connected in ways that would be unwieldy to express using just $\otimes$ and $\circ$.

The reason that abstract tensor notation is better for these applications is that, by simply listing boxes and their connections, it embraces the idea that 'only connectivity matters'. This also explains why the translation between abstract tensor notation (a.k.a. diagram formulas) we saw in Section 3.1.3 is so straightforward.

Exercise* 3.43 Define abstract tensor systems for the more general case that there are distinct system-types.

Exercise* 3.44 How would you define the process theory **relations** as an abstract tensor system? (Hint: see the 'algorithm' for computing relations of a diagram at the end of Section 3.3.3.)

*3.6.2 Symmetric Monoidal Categories**

Another symbolic way to define process theories is what a category-theorist would call a *strict symmetric monoidal category*. In essence this boils down to axiomatising the operations of parallel composition $\otimes$ and sequential composition $\circ$ of Section 3.2. Doing so is a bit much to take in all at once, so let's drop the 'symmetric' part for the moment, and just look at the following definition.

Definition 3.45 A *strict monoidal category* (SMC) $\mathcal{C}$ consists of:

1. a collection $\mathrm{ob}(\mathcal{C})$ of *objects*,
2. for every pair of objects A, B, a set $\mathcal{C}(A, B)$ of *morphisms*,
3. for every object A, a special identity morphism:

$$1_A \in \mathcal{C}(A, A)$$

4. a sequential composition operation for morphisms:

$$\circ : \mathcal{C}(B, C) \times \mathcal{C}(A, B) \to \mathcal{C}(A, C)$$

5. a parallel composition operation for objects:

$$\otimes : \mathrm{ob}(\mathcal{C}) \times \mathrm{ob}(\mathcal{C}) \to \mathrm{ob}(\mathcal{C})$$

6. a unit object

$$I \in \mathrm{ob}(\mathcal{C})$$

7. and a parallel composition operation for morphisms:

$$\otimes : \mathcal{C}(A,B) \times \mathcal{C}(C,D) \to \mathcal{C}(A \otimes C, B \otimes D)$$

satisfying the following conditions:

1. $\otimes$ is associative and unital on objects:

$$(A \otimes B) \otimes C = A \otimes (B \otimes C) \qquad A \otimes I = A = I \otimes A$$

2. $\otimes$ is associative and unital on morphisms:

$$(f \otimes g) \otimes h = f \otimes (g \otimes h) \qquad f \otimes 1_I = f = 1_I \otimes f$$

3. $\circ$ is associative and unital on morphisms:

$$(h \circ g) \circ f = h \circ (g \circ f) \qquad 1_B \circ f = f = f \circ 1_A$$

4. $\otimes$ and $\circ$ satisfy the *interchange law*:

$$(g_1 \otimes g_2) \circ (f_1 \otimes f_2) = (g_1 \circ f_1) \otimes (g_2 \circ f_2)$$

Wow, that looks like a lot! However, if we stare at this definition for a bit, some of it should start to look fairly familiar. First, note that what we have been calling types or system-types are called *objects* in category theory. Similarly, what we call processes are called *morphisms*. The set $\mathcal{C}(A,B)$ should be thought of as the set of all morphisms with input type A and output type B. Typically we write $f : A \to B$ instead of $f \in \mathcal{C}(A,B)$. A *category* is a collection of such sets with sequential composition. A *monoidal category* is the same, but with sequential and parallel composition. We'll say more about what 'strict' means shortly. First, note that we have almost all of the ingredients we need to build circuit diagrams – all except swaps. This is where the *symmetric* part comes in.

Definition 3.46 A *strict symmetric monoidal category* is a strict monoidal category with a *swap morphism*:

$$\sigma_{A,B} : A \otimes B \to B \otimes A$$

defined for all objects A, B, satisfying:

$$\sigma_{B,A} \circ \sigma_{A,B} = 1_{A \otimes B} \qquad \sigma_{A,I} = 1_A$$

$$(f \otimes g) \circ \sigma_{A,B} = \sigma_{B',A'} \circ (g \otimes f)$$

$$(1_B \otimes \sigma_{A,C}) \circ (\sigma_{A,B} \otimes 1_C) = \sigma_{A,B \otimes C}$$

Saying that a monoidal category is *strict* means that $\otimes$ is associative and unital 'on the nose', whereas a (non-strict) monoidal category only requires them to be *isomorphic*.

Definition 3.47 An object A is *isomorphic* to an object B, denoted:

$$A \cong B$$

if and only if there exists a pair of morphisms:

$$f : A \to B \qquad\qquad f^{-1} : B \to A$$

such that:

$$f^{-1} \circ f = 1_A \qquad\qquad f \circ f^{-1} = 1_B$$

The morphism f is then called an *isomorphism*.

Most monoidal categories we find in the wild tend to be of the non-strict variety. That is, their parallel composition only satisfies:

$$(A \otimes B) \otimes C \cong A \otimes (B \otimes C) \qquad\qquad A \otimes I \cong A \cong I \otimes A$$

We already briefly encountered this situation in Section 3.3.1. Recall that $\otimes$ was defined in terms of the Cartesian product. When we compare the sets $(A \times B) \times C$ and $A \times (B \times C)$ (and the sets A and $A \times \{*\}$), we do not get *equal* sets, but rather sets whose elements are in one-to-one correspondence:

$$((a, b), c) \leftrightarrow (a, (b, c)) \qquad\qquad a \leftrightarrow (a, *)$$

To define a non-strict monoidal category, we must include these correspondences in the definition of the category as so-called *structural isomorphisms*, and we must assume several (more!) equations called *coherence equations* that guarantee that they behave sensibly when they are composed together.

However, we happily glazed over this fact when we defined the process theories of functions and relations in Section 3.3. Why? Because of the following *coherence theorem*.

Theorem 3.48 Every (symmetric) monoidal category $\mathcal{C}$ is equivalent to a strict (symmetric) monoidal category $\mathcal{C}'$.

Equivalent categories are, for all practical purposes, the same. We'll say more about what this means in Section* 5.6.4. The proof of Theorem 3.48 consists of an explicit construction of the category $\mathcal{C}'$ by a procedure called *strictification*. This procedure is used implicitly when, e.g. we decide to treat elements $((a, b), c)$, $(a, (b, c))$ and (a, b, c) as 'the same', which we do without further comment throughout this text.

Furthermore, we are justified in using circuit diagrams to talk about symmetric monoidal categories because of the following theorem.

Theorem 3.49 Circuit diagrams are sound and complete for symmetric monoidal categories. That is, two morphisms f and g are provably equal using the equations of a symmetric monoidal category if and only if they can be expressed as the same circuit diagram.

To summarise, it is possible to translate many concepts between our purely diagrammatic language and category theory, using this correspondence:

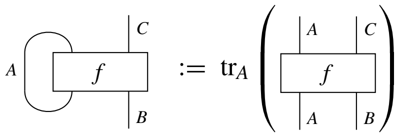
$$\begin{aligned}\text{process theory} &\leftrightarrow \text{(strict) symmetric monoidal category}\\ \text{process} &\leftrightarrow \text{morphism}\\ \text{system-type} &\leftrightarrow \text{object}\end{aligned}$$

*3.6.3 General Diagrams versus Circuits**

Some readers may have noted that abstract tensor systems and symmetric monoidal categories aren't exactly one and the same. While symmetric monoidal categories correspond to circuits, abstract tensor systems represent more general kinds of diagrams in which directed cycles (cf. Definition 3.20) are allowed. However, we can make the two concepts coincide.

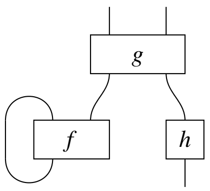
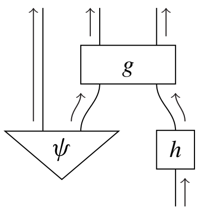
In one direction, one can force abstract tensor systems to be equivalent to strict symmetric monoidal categories by only requiring the contraction operation c_A^B to be well-defined if it does not introduce a directed cycle.

Conversely, it is possible to introduce 'feedback loops' into symmetric monoidal categories, obtaining so-called *traced symmetric monoidal categories*. These categories assume an additional operation called the *trace*:



which we will meet in Section 4.2.3 of the next chapter. The trace satisfies a handful of axioms to ensure that it behaves like a 'feedback loop' and that it respects the other categorical structure. Then, strict traced symmetric monoidal categories are equivalent to abstract tensor systems, in that it is possible to turn an abstract tensor system into a strict traced symmetric monoidal category and vice versa without losing any meaningful information.

We can summarise the above in the following table:

	'General' diagram	Circuit diagram
Diagrams	 i.e. outputs to inputs	 i.e. admits causal structure
ATS	ATS	ATS without cycles
SMC	traced SMC	SMC

3.7 Historical Notes and References

The idea of representing the composition of mathematical objects using nodes and wires dates back a very long time. A prominent example is the flow chart, which goes back (at least) to the 1920s (Gilbreth and Gilbreth, 1922). In physics, the most notable example is Feynman diagrams. Despite their name, it is now recognised that they originated in the work of Ernst Stueckelberg around 1941 and were only later independently re-derived by Richard Feynman around 1947. In a lecture at CERN after having been awarded the Nobel Prize Richard Feynman referred to this fact the moment that Stueckelberg left the room: 'He did the work and walks alone toward the sunset; and, here I am, covered in all the glory, which rightfully should be his!' (Mehra, 1994).

Penrose (1971) was the first to introduce the particular kind of diagrams we use in this book as an alternative to the abstract tensor notation (see Section* 3.6.1), which he also introduced. In fact, he already started using diagrams when he was still an undergraduate student (Penrose, 2004).

Paul Dirac introduced his new notation in Dirac (1939). The connection between Dirac notation and the corresponding use of the triangle notation for states and effects in string diagrams was observed in Coecke (2005). The idea of treating diagrams themselves as the basic structure rather than categories has recently become prominent in the quantum foundations community. It has been used, for example, in efforts towards crafting alternative formulations of quantum theory and a theory of quantum gravity (Chiribella et al., 2010; Coecke, 2011; Hardy, 2011, 2013b).

The unwillingness of many to accept diagrams as a rigorous mathematical language (alluded to in Section 3.1.3) may be largely due to the Bourbakimindset in which mathematics has been taught for much of the previous century. The Bourbaki collective aimed to found all mathematics on a set-theoretic basis (Bourbaki, 1959–2004). Great credit should be given to John Baez, the pioneer of scientific blogging, who by means of his outstanding didactical skills managed to convince many of the amazingness of diagrams. His old weekly column 'This Week's Finds in Mathematical Physics' is still a great resource on diagrammatic reasoning (Baez, 1993–2010).

The connection between circuit diagrams and algebraic structure (in the form of symmetric monoidal categories, cf. Theorem 3.49) was established in Joyal and Street (1991), where circuit diagrams are referred to as 'progressive polarised diagrams'. Traced symmetric monoidal categories and their graphical notation were introduced in Joyal et al. (1996); however, the closely related notion of a 'compact closed category' (whose diagrams are string diagrams, which we will encounter very soon) had already been known for some time. The historical notes in the next chapter provide appropriate references.

The proof of correctness of the graphical language for traced symmetric monoidal categories has been sketched in various papers (e.g. Hasegawa et al., 2008), and a full proof is given by Kissinger (2014a). Following on from Joyal and Street's work came a plethora of variations of these diagrams and corresponding algebraic structures (see Selinger, 2011b).

Monoidal categories themselves were first introduced in Benabou (1963). Theorem 3.48 on coherence is crucial for connecting monoidal categories to diagrams and was first stated and proved by Mac Lane (1963). The ‘little dance’ in (3.24), a.k.a. the Eckmann–Hilton argument, is from Eckmann and Hilton (1962).

Abstract tensor systems, and their associated ‘abstract index notation’, were introduced by Penrose (1971) at the same time as the corresponding diagrammatic notation, in order to talk about various kinds of multilinear maps without needing to fix bases in advance. This notation is now very common in theoretical physics and especially general relativity. Penrose provides an introduction for a general audience in Penrose (2004). The equivalence between abstract tensor systems and traced symmetric monoidal categories is given by Kissinger (2014a).

If you want to read more on general category theory, some good places to start are Abramsky and Tzevelekos (2011) for the connection between categories and logic, Coecke and Paquette (2011) for the generality of the notion of symmetric monoidal categories, and Baez and Lauda (2011) for the role categorical structures have played in physics. Standard textbooks for mathematicians are Borceux (1994a,b) and Mac Lane (1998); standard textbooks for computer scientists are Barr and Wells (1990) and Pierce (1991); and standard textbooks for logicians are Lambek and Scott (1988) and Awodey (2010). In the context of category theory, the idea that morphisms represent processes emerged mostly in the context of applications in computer science, where programs are represented by morphisms in a category and data types are the objects. Similarly in logic, proofs can be represented by morphisms in a category, with propositions as the objects. This trifecta of morphisms, programs, and proofs are all connected by what’s known as the *Curry–Howard–Lambek isomorphism* (Lambek and Scott, 1988).

The quote at the beginning of this chapter is taken from Bohm and Peat (1987). As discussed in Section 1.3, Bohm’s views played a key role in making process ontology more prominent in physics.