14
Quantomatic

We will encourage you to develop the three great virtues of a programmer: laziness, impatience, and hubris.
– *Larry Wall,* Programming Perl, 1st edition

So there you have it. Hundreds of pages and more than 3000 diagrams later, we've told you (pretty much) everything we know about quantum theory and diagrammatic reasoning. So, where to now? Is it time for everyone to start covering blackboards and filling papers with diagrams? Of course!

But even better, what if someone else did all the diagrammatic proving for you while you sit back, relax, and have a beer? That's even better! The fact that the diagrams we use are essentially made up of a finite number of ingredients (namely, spiders) means they are particularly well suited to *automated reasoning*. In this subfield of artificial intelligence, one develops software that allows a computer to do a whole range of things often associated with human mathematicians: from simply checking mathematical proofs for correctness to automatically searching for new proofs or even new and interesting conjectures to then try and prove automatically.

In the past, automated reasoning has typically concerned traditional, formula-based mathematics built on formal logics and set-based algebraic structures, and there it has been very successful. It has provided us with tools called *proof assistants*, which allow us to automatically construct proofs of mind-bending results, such as Gödel's incompleteness theorems, and rigorously check proofs that are way too big for a human mathematician to get totally correct, like Kepler's conjecture, the four-colour theorem, and the Feit–Thompson theorem (famously massive proofs in geometry, graph theory, and group theory, respectively).

In addition to serving essentially as 'robot teaching assistants', which tirelessly check the work of human mathematicians, techniques from automated reasoning can actually tell us something new, via *conjecture synthesis*. Much as a human mathematician would discover features and behaviours of an unfamiliar mathematical creature by 'poking at it' (i.e. making educated guesses about how it will behave and trying to prove them), there exist automated techniques that do this at high speed. When it succeeds in a proof, the result is a freshly minted theorem that no human has ever seen or even thought to ask about.

In fact, if you are a bit stuck for what to buy someone for Christmas, such theorems can be bought (and named after that special someone) online at theorymine.co.uk.

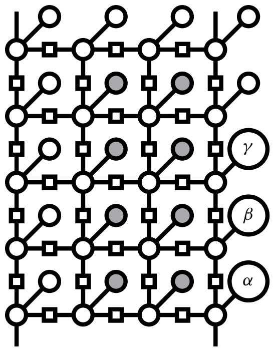
'Okay, okay,' you say, 'but what does this have to do with diagrams?' In this chapter, we will talk about how automated reasoning has made it into the diagrammatic realm and can be applied to all of the stuff we've done in this book, via a diagrammatic proof assistant called Quantomatic. This tool lets one create string diagrams, and string diagram equations, then apply those equations to prove theorems. This can be done either in a step-by-step fashion, as we've been doing in this book all along, or in an automated way, via *proof strategies*. In this way, we can easily handle string diagrams and diagram proofs that get way too big to work with by hand. We also see powerful theorems such as the completeness theorem for the ZX-calculus from Section 9.4.5 start to pay some serious dividends, as they can be translated into strategies that totally automate graphical proofs.

Diagrams are particularly well suited for conjecture synthesis, as we often have a collection of basic 'generators' we are interested in (e.g. spiders) and wish to discover their graphical calculus. At the end of this chapter, we will discuss how conjecture synthesis works in Quantomatic and examine the progress made towards automatic discovery of new theorems about physical theories.

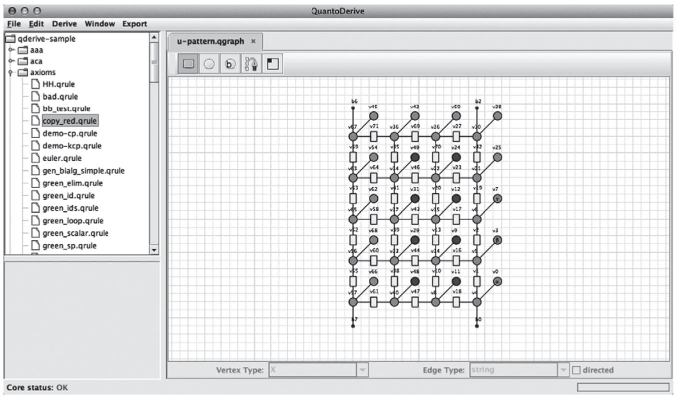
14.1 Taking Quantomatic for a Spin

We'll now give some idea of what it's like to work with Quantomatic and the type of calculations it can do. We will avoid going into too many details, as these will inevitably become out of date as the tool grows and changes over time. Of course, the best thing you can do is download the latest version from quantomatic.github.io and follow along for yourself. On the website, you will also find sample projects and tutorials.

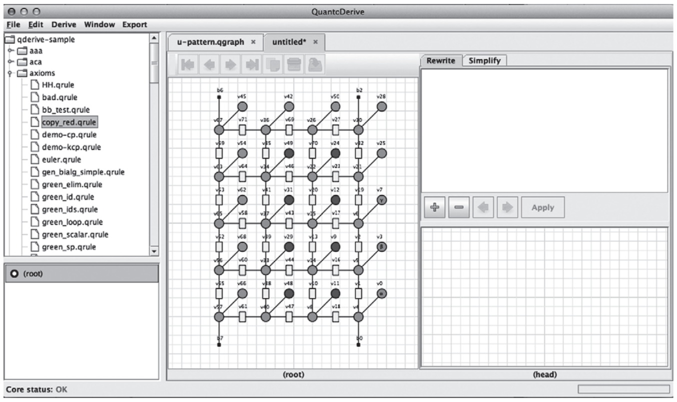
A good candidate for automation is MBQC computations, given how big the diagrams can become. Consider this computation:



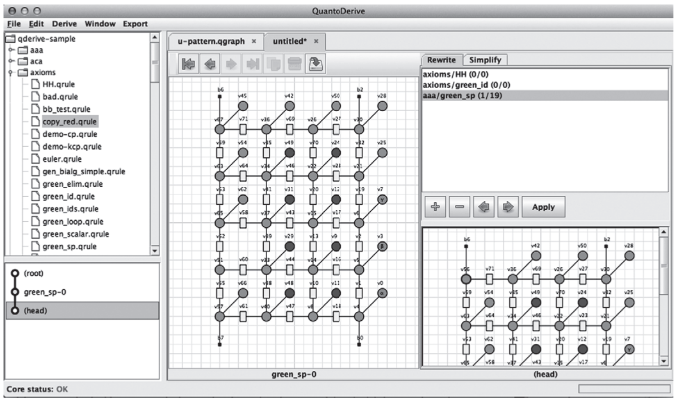
Here it is in Quantomatic:



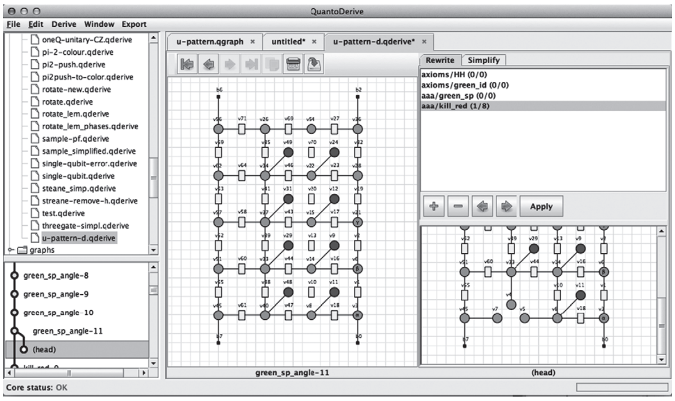
Now, let's see what this simplifies to by starting a *derivation*:



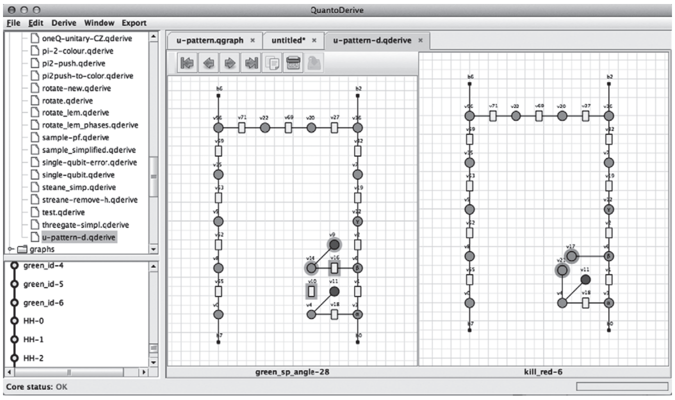
We add a few rules and start doing some rewriting:



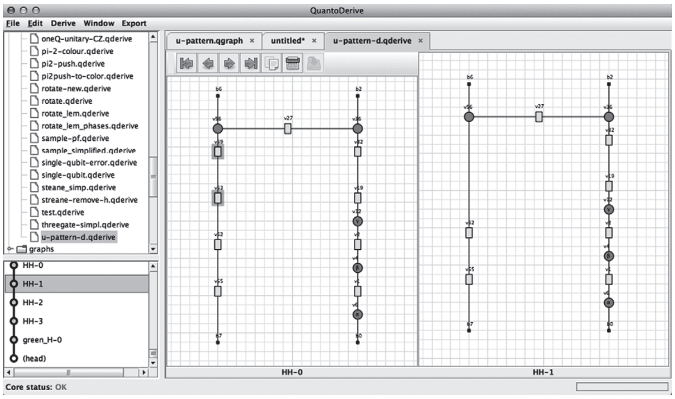
and keep rewriting:



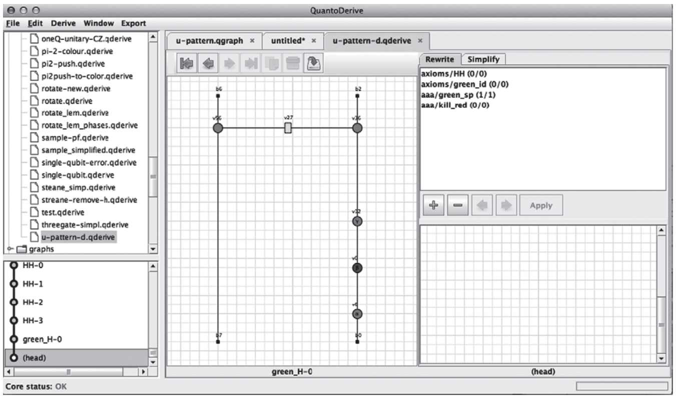
... still going:



... almost there ...

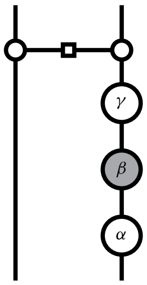


... and voila:

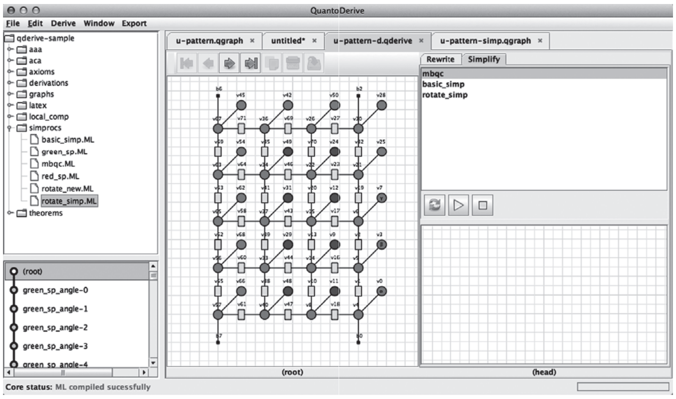


axioms/HH (0/0)
axioms/green_id (0/0)
aaa/green_sp (1/1)
aaa/kill_red (0/0)

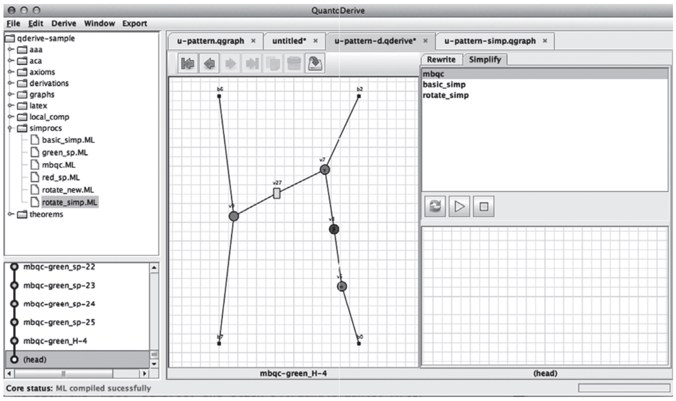
After a little over 50 steps, we end up with phase gates and a CZ-gate:



Wow, that seems like a lot of work! Actually, it only took about 15 minutes to do in Quantomatic. However, we would prefer to do this in 15 seconds, so let's switch to the *simplifier*:

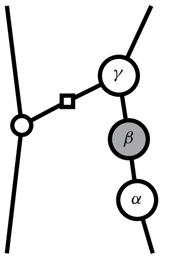


The Quantomatic's simplifier does automated rewriting, using one or more programmable proof strategies. These are little programs that tell Quantomatic how to choose the next rule to apply. We pick the `mbqc` strategy, and click ▷. After about 15 seconds and a trippy light show, we get:



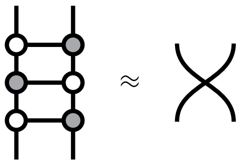
Core status: ML compiled sucessfully

which is exactly what we expected, modulo a bit of extra spider fusion:

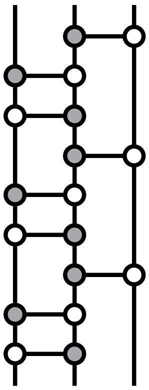


Despite having lots of steps, this calculation was really pretty straightforward. After a bit of spider fusion, the only thing left to do was eliminate the ●-spiders coming from Z-measurements, as we explained in Section 12.3.2, and eliminate the remaining H-gates with the colour-change rule.

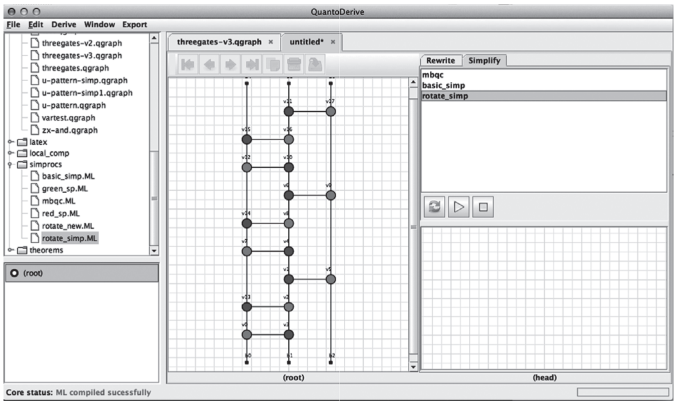
So, let's try something a bit less obvious, coming from the circuit model. We already saw that three CNOT-gates equals swap:



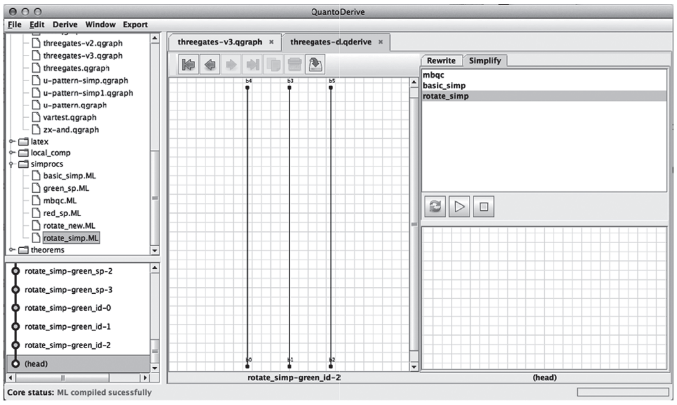
But how about this much more complicated configuration of CNOT gates?



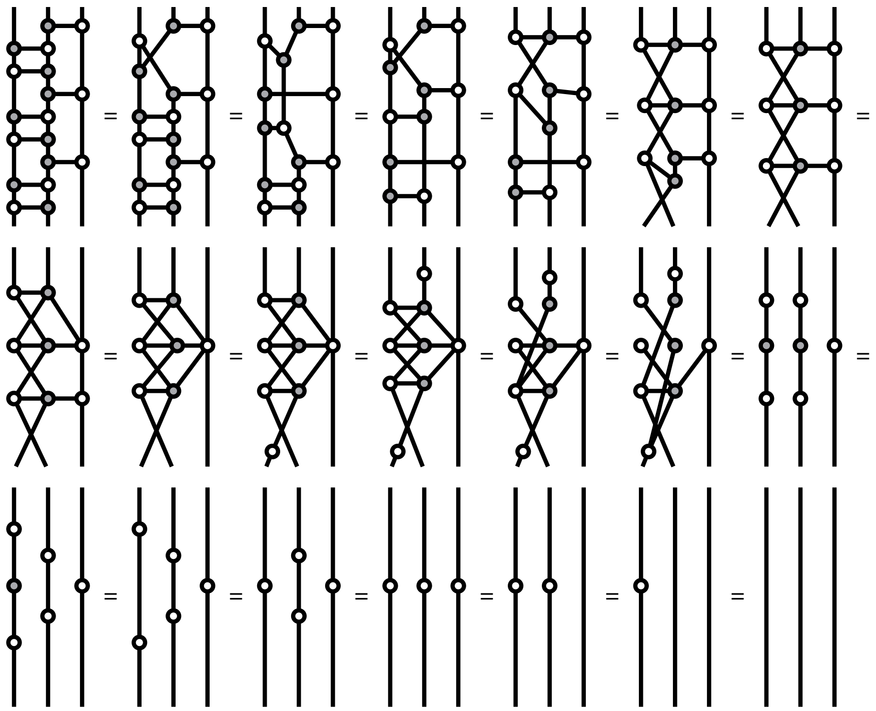
Well, we can start to do some spider fusion, or maybe use strong complementarity to get rid of the four-cycles, but even after we do that, it's far from obvious how to proceed. So, let's feed it to Quantomatic's simplifier:



This time, we choose `rotate_simp`, which is a proof strategy that always reduces any ZX-diagram without phases into canonical form:



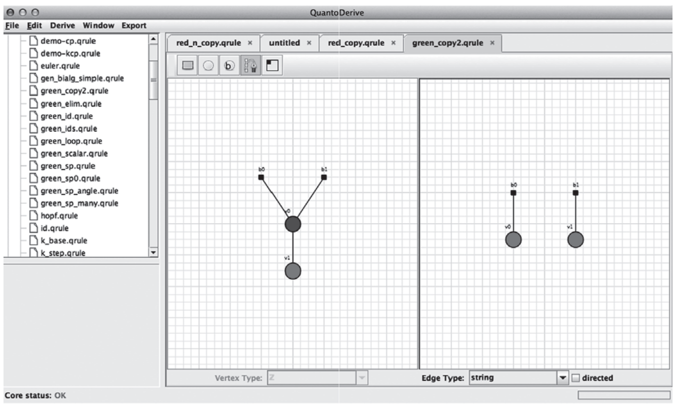
So, this is in fact just the identity. To see how Quantomatic got there, we can export the proof that the simplifier came up with:



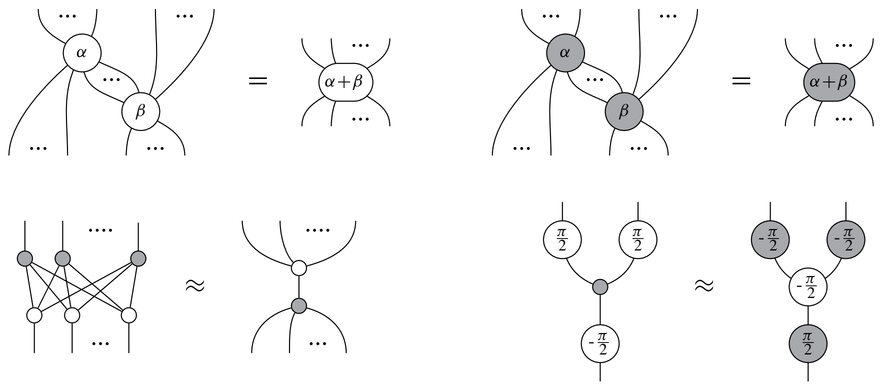
Even though we were a bit stuck on how to proceed, Quantomatic happily reduced this to the identity. And we get to take all the credit!

14.2 !-Boxes: Replacing the ‘Dot, Dot, Dot’

One thing we’ve happily skirted over so far is what rules look like in Quantomatic. Simple rules are basically what you’d expect: a pair of graphs with identical inputs/outputs on each side:

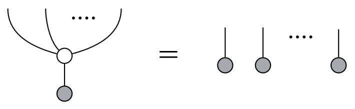


but looking at the ZX-calculus:

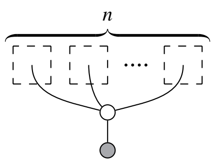


three out of the four rules are actually whole families of rules, as evident in the use of '. . .' in the LHS and RHS. When we give such a rule, we implicitly rely on the fact that a human with a brain is reading it, so either it is obvious what we mean or we can pretty easily explain it in words.

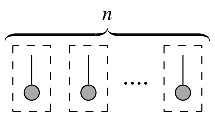
Clearly this isn't going to work for Quantomatic, so we need to formalise the concept of 'dot, dot, dot' in a way that a machine can understand. To see how this works, let's start with a slightly simpler '. . .' rule, the n-fold copy rule:



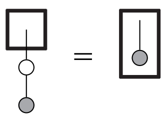
The LHS and the RHS both have some subdiagram that has been copied n times. On the LHS, this subdiagram just consists of a single output wire:



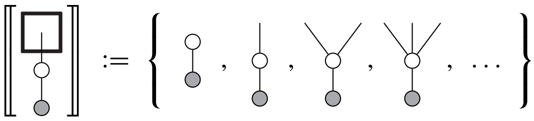
where every copy is connected to the same ○-spider, whereas the RHS contains n copies of an entire ●-spider with one output:



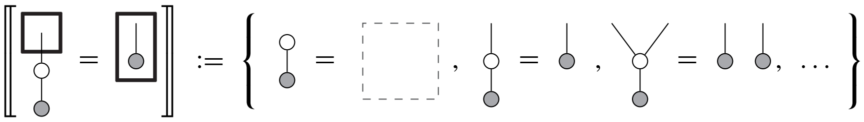
We can indicate this repetition by enclosing part of a diagram in a *!-box*:



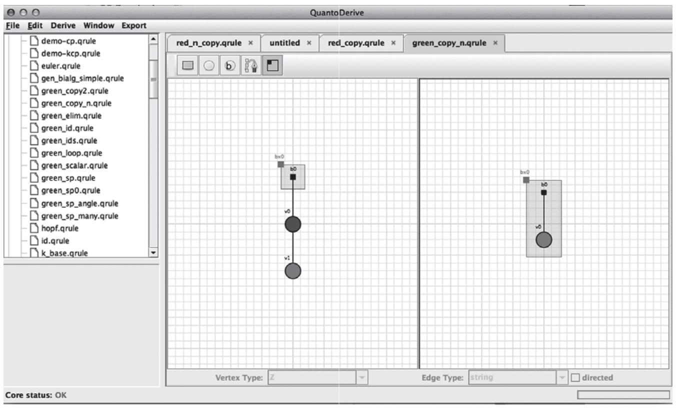
(which, incidentally is pronounced 'bang-box'). A diagram with !-boxes is interpreted as a family of diagrams obtained by copying the diagram inside the !-box n times, while retaining any wires into or out of the !-box:



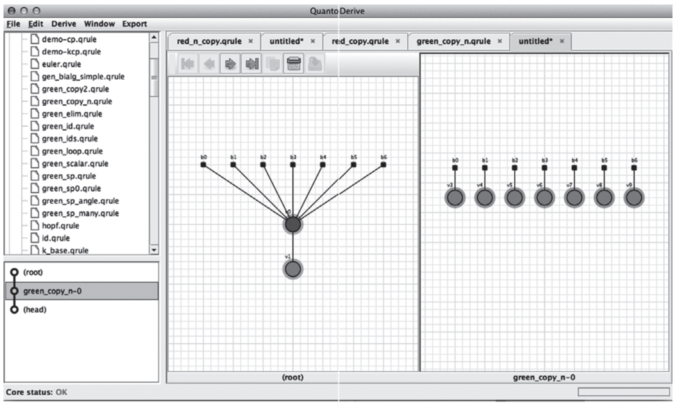
Hence an equation between diagrams with !-boxes gives an entire family of diagram equations, where each !-box on the LHS and its corresponding !-box on the RHS are copied n times:



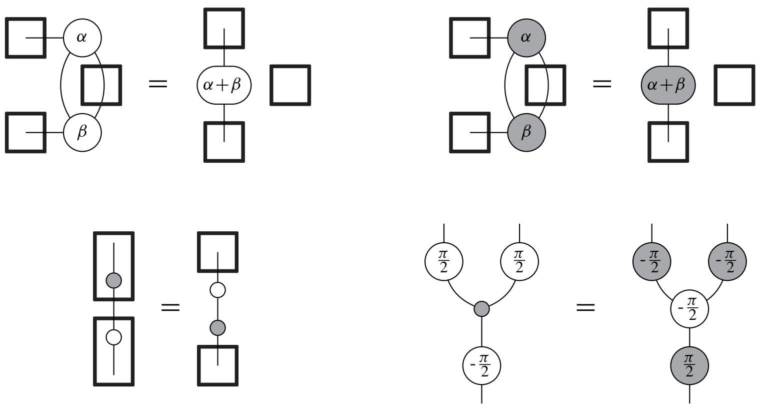
This !-box rules look like this in Quantomatic:



which can be applied to produce, e.g.:



Using !-box rules, we can express the whole ZX-calculus in Quantomatic as follows:



Remarkably, the fully general strong complementarity rule, which was a bit difficult to describe in words, is very easy to write down with !-boxes.

14.3 Synthesising Physical Theories

Can a machine come up with interesting new theorems about physics? More specifically, can it take the ingredients of a diagrammatic theory (e.g. spiders or antispiders) and start plugging them together to discover automatically how they interact? This is what the next feature of Quantomatic, called *conjecture synthesis*, is all about.

Rather than starting with a fixed set of diagram rewrite rules and trying to derive new rules, it starts with a concrete model and tries to discover which rules should hold. In other words, suppose we have a collection of generators, given concretely, e.g. as matrices:

$\leftrightarrow \begin{pmatrix}1&0&0&0\\0&0&0&1\end{pmatrix}$ $\leftrightarrow \begin{pmatrix}1\\1\end{pmatrix}$

$\leftrightarrow \begin{pmatrix}1&0\\0&0\\0&0\\0&1\end{pmatrix}$ $\leftrightarrow \begin{pmatrix}1&1\end{pmatrix}$

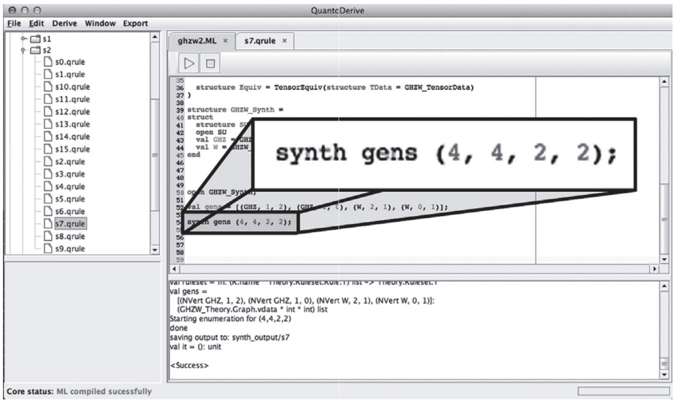
$\leftrightarrow \begin{pmatrix}0&1\\1&0\\1&0\\0&0\end{pmatrix}$ $\leftrightarrow \begin{pmatrix}0&1\end{pmatrix}$

$\leftrightarrow \begin{pmatrix}1&0&0&0\\0&1&1&0\end{pmatrix}$ $\leftrightarrow \begin{pmatrix}1\\0\end{pmatrix}$

but we have no idea what sorts of equations hold between diagrams of these generators. Are they spiders? Do they satisfy something like strong complementarity? Or are they something else altogether?

The way human scientists (e.g. us) would try to figure this out is simply to start plugging these things together and see which equations arise. This process is totally algorithmic, so in principal, a machine could do it. Furthermore, a machine can do it in a it would be much faster and more systematic way.

This is where conjecture synthesis comes in. This is a routine to effectively enumerate diagrams up to some given size while using the rules we discover along the way to eliminate redundancies and speed up the search for new, interesting rules. After setting up a list of generators (called '`gens`' below), this routine is invoked in Quantomatic with an incantation like this:



```
  structure Equiv = TensorEquiv(structure TData = GHZW_TensorData)
)

structure GHZW_Synth =
struct
  structure SU
  open SU
  val GHZ = GHZ
  val W = GHZW_
end



open GHZW_Synth;

val gens = [(GHZ, 1, 2), (GHZ, 1, 0), (W, 2, 1), (W, 0, 1)];

synth gens (4, 4, 2, 2);
```

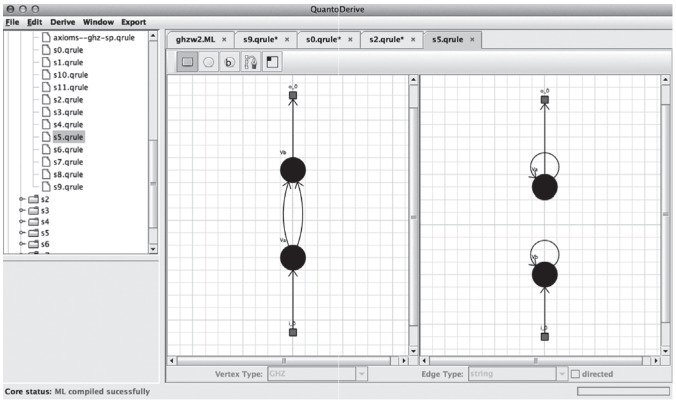
```
synth gens (4, 4, 2, 2);
```

```
val ruleset = fn: (R.name * Theory.Ruleset.Rule.T) list -> Theory.Ruleset.T
val gens =
  [(NVert GHZ, 1, 2), (NVert GHZ, 1, 0), (NVert W, 2, 1), (NVert W, 0, 1)]:
  (GHZW_Theory.Graph.vdata * int * int) list
Starting enumeration for (4,4,2,2)
done
saving output to: synth_output/s7
val it = (): unit

<Success>
```

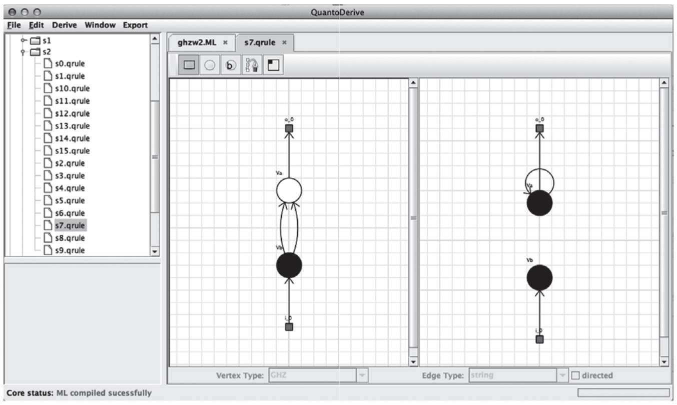
Core status: ML compiled sucessfully

which performs the synthesis up to size $(4, 4, 2, 2)$; that is, it will generate rules involving at most four generators, four wires between them, two inputs, and two outputs. With those parameters, it found about 170 rules. Amongst them are lots of versions on spider fusion, our friends the exploding antispiders (shown here in black):



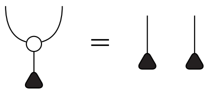
Core status: ML compiled sucessfully

and also a variation with the GHZ-spider on top:

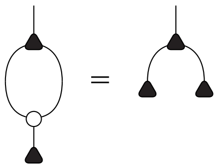


Vertex Type: GHZ
Edge Type: string
Core status: ML compiled sucessfully

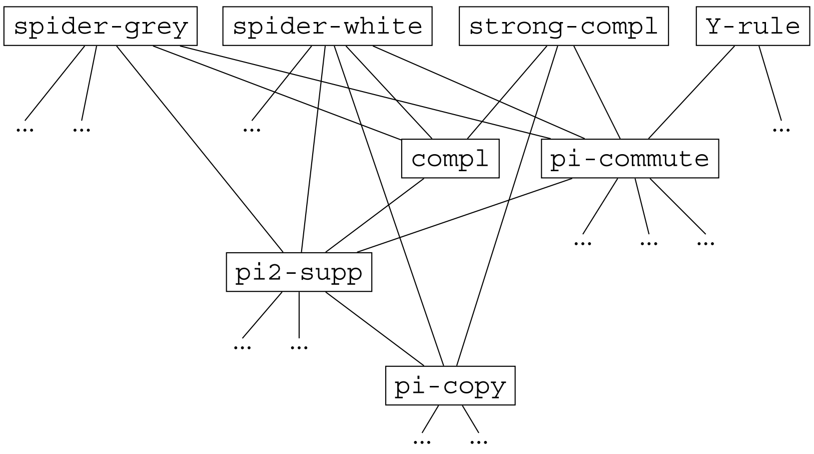
Of course, 170 unrelated rules is not a graphical calculus. This is the point where things start to get really interesting, as we try to discover relationships between these rules. Which rules are the most important? Which give the most ‘proving power’? The search procedure is designed to eliminate rules that are ‘obviously’ provable from others; e.g. if we know this rule:



conjecture synthesis is smart enough not to bother checking whether this rule holds:



But what about the less obvious stuff? For this, one can use automated proof strategies such as the ones we described in Section 14.1 to try to build up a dependency relation between rules, i.e. a graph of which rules can be proven using others. For instance, the ZX-calculus would provide something like this:



from which one could very quickly discover what are the most interesting basic rules and non-trivial theorems that exist in this theory.

This is very much ongoing work, and many aspects about what makes such a procedure effective are still unclear. The high-water mark is of course discovering theorems that are interesting in their own right, quite apart from how they were found. Of course, this has the unfortunate side effect of putting us mere human scientists out of work!

14.4 Historical Notes and References

The theory behind Quantomatic Kissinger and Zamdzhiev (2015) was developed by Dixon and Kissinger (2013), following from earlier results by Dixon and Duncan (2010) and Dixon et al. (2010). !-boxes were first proposed in Dixon and Duncan (2009) and formalised in Kissinger et al. (2014).

One of the first interactive theorem provers was Stanford LCF, developed by Robin Milner (1972) and named after the logic of computable functions by Dana Scott (1993). This was succeeded by Edinburgh LCF (Gordon et al., 1979), which originated the *LCF paradigm* of a core logical 'kernel' driven by various (semi-)automated 'tactics'. A short history can be found in Gordon (2000). Other notable provers to adopt this paradigm are Isabelle (Paulson et al., 1986) and Coq (Coqand et al., 1984). The latter has gained significant attention lately due to its use in homotopy type theory and the univalent foundation of mathematics (Shulman et al., 2013).

In recent years, several enormous proofs were fully formalised in theorem provers. The FlysPecK project, led by Hales et al. (2015), succeeded in giving a formal proof of the Kepler conjecture in 2015. A formal proof of the four-colour theorem was given by Gonthier (2008) and the Feit–Thompson theorem by Gonthier et al. (2013).

The method for synthesising diagrammatic theories was proposed by Kissinger (2012b), based on techniques introduced by Johansson et al. (2011). The latter has been used to (among other things!) automatically prove little theorems that can be bought as gifts (Bundy et al., n.d.).

The second example in Section 14.1 is equivalent to rule (C14) from (Selinger, 2015), one of the 15 rules used to construct a complete calculus for Clifford circuits.