Ann Math Artif Intell (2009) 56:23–42
DOI 10.1007/s10472-009-9141-x

Graphical reasoning in compact closed categories for quantum computation

Lucas Dixon · Ross Duncan

Published online: 2 July 2009

Abstract Compact closed categories provide a foundational formalism for a variety of important domains, including quantum computation. These categories have a natural visualisation as a form of graphs. We present a formalism for equational reasoning about such graphs and develop this into a generic proof system with a fixed logical kernel for reasoning about compact closed categories. A salient feature of our system is that it provides a formal and declarative account of derived results that can include 'ellipses'-style notation. We illustrate the framework by instantiating it for a graphical language of quantum computation and show how this can be used to perform symbolic computation.

Keywords Graph rewriting · Quantum computing · Categorical logic · Interactive theorem proving · Graphical calculi · Ellipses notation

Mathematics Subject Classifications (2000) 03G30 · 18C10 · 03G12 · 05C20 · 81P68

1 Introduction

Recent work in quantum computation has emphasised the use of graphical languages motivated by the underlying logical structure of quantum mechanics itself [1, 5, 7,

L. Dixon (✉)
University of Edinburgh, Edinburgh, UK
e-mail: l.dixon@ed.ac.uk

R. Duncan
University of Oxford, Oxford, UK
e-mail: ross.duncan@comlab.ox.ac.uk

8, 21]. These techniques have a number of advantages over the conventional matrix-based approach to quantum mechanics:

- The matrix representation is sensitive to the basis chosen for the underlying space; visual representation abstracts over the values in the matrices, removing irrelevant detail that is difficult or tedious for a human to interpret.
- Many properties have a natural graphical representation. For example, disjointness of subgraphs implies separability of quantum states.
- The algebra of graphs generalises to domains other than vector spaces: it provides a representation for compact closed categories [12].

A major problem with graphical calculi is the lack of machinery for automating their manipulation. The main contribution of this paper is a graph-based formalism that is suitable for representing and reasoning about compact closed categories with additional equational structure. This has a wide variety applications including reasoning about relations, stochastic processes, and synchronous processes [2, 8, 14]. In this paper we introduce the representation, develop it into a formal proof system, and highlight its application for symbolic reasoning about quantum computation.

We begin by presenting a graphical model of quantum computation, which displays the typical features of a graphical calculus. Quantum processes are represented by graphs built up from basic elements. We view vertices as operations which have types corresponding to their incident edges, and hence our notion of subgraph is different to that of standard texts on graph theory. A subgraph represents a "subprocess" so the types of the basic operations must be preserved, and thus we do not allow additional edges in a subgraph. Non-structural equivalences are captured by equations between graphs. An important result in this calculus is the Spider Theorem which takes the form of an equation between graphs involving informal ellipses notation (see Section 2).

The formalisation, in a graphical form, of rules containing ellipses notation and the corresponding reasoning with such rules requires an extension of the graphical calculus that eventually forms *graph patterns*. We develop this by first defining a formalism for graphs, their transformations, and an appropriate subgraph relation. We introduce a general form of graph combination, called *plugging*, which includes both parallel and sequential composition as special cases. Since redexes are preserved by plugging, this gives a compositional account of equational reasoning for compact closed categories. Our graph-based formalism is a faithful representation the of free compact closed category generated from its basic elements. We also introduce a general formalism for ellipses notation in graphs which forms *!-box graphs*. By combining *!-box graphs* with our compositional graph formalism, we provide a suitable representation for *graph patterns* that can formally represent and reason with rules derived from the Spider Theorem.

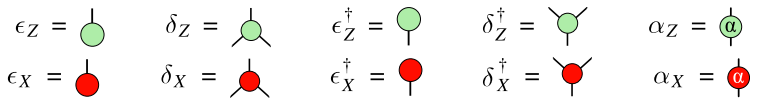
Using our graph-based formalism as the representational foundation, we develop a simple logical framework for manipulating models of compact closed categories. This has a suitable rewriting mechanism where the axioms of the underlying object-formalism are expressed as equations between graphs. We then present a short case study that illustrates the framework by instantiating it for the model of quantum computation introduced earlier. This shows how the framework can be used to symbolically perform simplifications of quantum programs as well as simulate computations.

2 Quantum computations as graphs

In this section we will describe a set of generators and equations used to reason about quantum computation, and show how some of its formal properties lead to particular issues for the development of reasoning machinery.

Initiated in [1], a substantial strand of work in quantum informatics has involved the development of high-level models of quantum processes based on compact closed categories. In these formalisms, quantum processes—such as quantum logic gates, or the measurement of a qubit—correspond to arrows in the category, while the different quantum data types, usually just arrays of qubits, are the objects.

A recent account of the graphical language is provided by Coecke and Duncan [6]. This is based on a graphical language for compact closed categories, as described in Section 5, augmented with equations to describe the behaviour of quantum systems. Edges represent qubits and, in particular, distinguished domain and codomain edges represent the inputs and outputs respectively of a quantum process.[1] Internally, several edges may represent the same physical qubit at different times. An edge may even represent a "virtual" qubit which stands for a correlation between different parts of the system. Nodes are shaded (coloured) with a lighter shade (green) and a darker one (red) to denote two families of operations on qubits, expressed graphically as the following generators:



where $\alpha \in [0, 2\pi)$. The δ_Z and ϵ_Z represent quantum operations which respectively copy and delete the eigenstates of the Pauli Z operator.[2] Notice that δ_Z has one edge in its domain for the qubit to be copied, and two edges in its codomain for the two copies it produces. Similarly, ϵ_Z has one qubit as input and no outputs. The adjoints $\delta_Z^\dagger$ and $\epsilon_Z^\dagger$ correspond to an operation known as *fusion*, and to the operation of preparing a fresh qubit in a certain state. The α_Z corresponds to phase shift of angle α in the Z direction. The family of maps indexed by X are defined in exactly the same way, but relative to the Pauli X operator. In addition, we have 

which represents a Hadamard gate.

The free compact closed category is then given by all graphs formed by composing and tensoring these basic graphs. All quantum operations may be defined by combining these simple operations—which are essentially classical—on two complementary observables.

We emphasise that this is a notation for representing quantum processes, not just quantum states. In this setting a state is simply a process with no inputs; that

[1] In this account, no interpretation of edge direction is needed as objects in the underlying categorical model are self dual.

[2] Uniform copying operations are forbidden by the no-cloning theorem [25], but such operations are possible if we demand only the eigenstates of some self-adjoint operator to be copied. Other states will not not copied. The same remarks hold true for erasing [15].

is, a graph with empty domain. Since our formalism is based on the underlying mathematical structure rather than any particular model of quantum computation, it is capable of representing quantum circuits, measurement-based quantum computations, as well as other models. Indeed, an important application of this work is to show that states or computations implemented differently are equivalent.

The beauty of graphical calculi for compact closed categories is that equations which hold for general algebraic reasons are absorbed into the notation. However in order to represent quantum computation, generic structure will not suffice: we need additional equations between graphs. In the system we present here, these describe the interaction between complementary observables and allow equivalent computations to be proved equivalent. The equations are discussed in detail in [6] and are presented here graphically in Fig. 1.

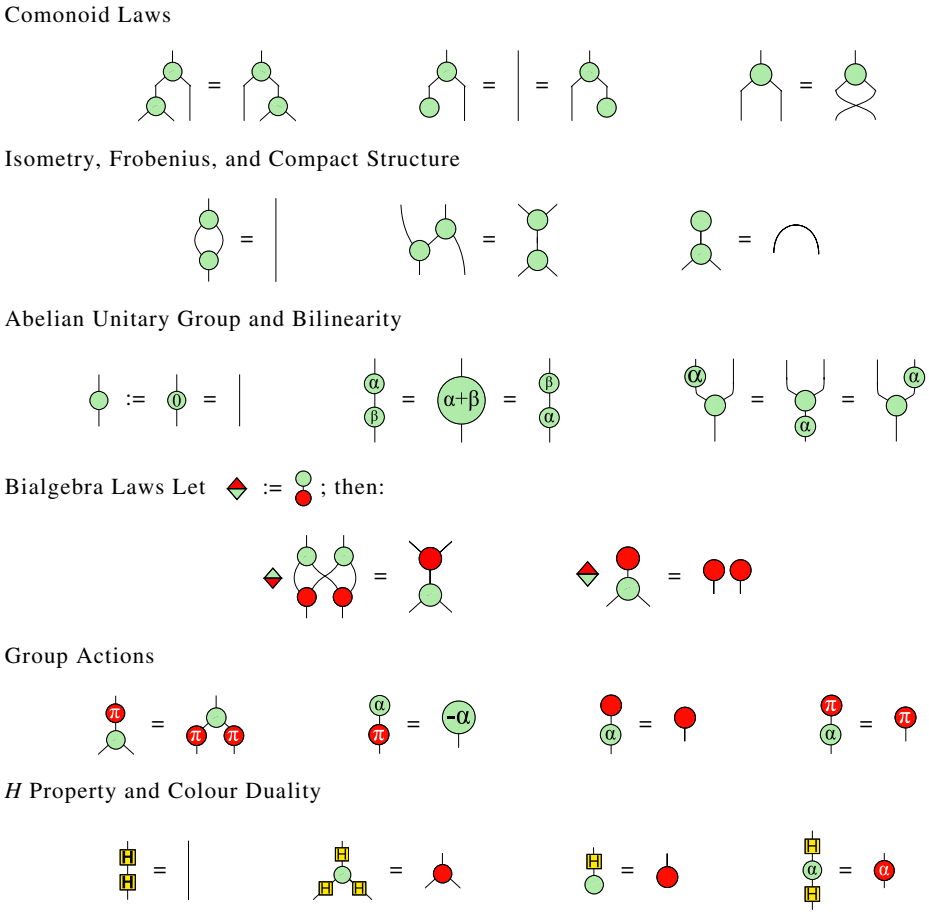


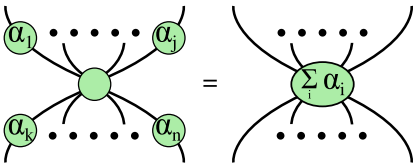
Fig. 1 Graphical equations for quantum systems. In addition, we have a "colour duality": each equation shown here gives rise to second, which is obtained by exchanging the two colours (shades). The colour duality is derivable from the equations involving H

The equations from Fig. 1 which involve only one colour allow the remarkable *spider theorem*, first noted in [8], to be proved:

Theorem 1 (Spider Theorem) *Let G be a connected graph generated from δ_Z, ϵ_Z, α_Z and their adjoints; then G is totally determined by the number of inputs, the number of outputs, and the sum modulo 2π of the αs which occur in it.*

Remark 1 The theorem holds also for the X family of operations.

Hence any connected subgraph involving nodes of only one colour may be collapsed to a single vertex, with a single value α, giving a "spider". Conversely, a spider may be arbitrarily divided into sub-spiders, provided the total in- and out-degree is preserved, along with the sum of the αs. Informally, this can be depicted graphically as the equation:



From this one can derive n-fold versions of many of the other equations.

Spiders offer a very intuitive way to manipulate graphs, and are far more compact and convenient in calculations than the graphs built up naively from the generators. However, no finite set of equations suffices to formalise spiders: we must move from finite graphs, where each vertex has bounded degree, and which are subject to a finite number of equations, to a system where nodes may have arbitrarily many edges, and there are infinitely many equations. The desire to retain intuitive reasoning methods for these infinite families of equations motivates the extension of graphs to *graph patterns*, the main subject developed in this paper.

3 Graphs

Definition 1 (Graph) A *directed graph*[3] consists of a 4-tuple (V, E, s, t) where V and E are sets, respectively of *vertices*[4] and *edges*, and s and t are maps which give the source and target vertices of a an edge respectively:

$$E \underset{t}{\overset{s}{\rightrightarrows}} V$$

We will assume throughout this paper that both V and E are finite.

[3]Equivalently: a directed graph is a functor G from $\bullet \rightrightarrows \bullet$ to **Set**; a graph morphism is then a natural transformation $f : G \Rightarrow H$.

[4]We will use the words "vertex" and "node" interchangeably.

Remark 2 Note that any number of edges are allowed between vertices, including from a vertex to itself.

Let $\mathrm{in}(v) := t^{-1}(v)$ and $\mathrm{out}(v) := s^{-1}(v)$ denote the *incoming* and *outgoing* edges at a vertex v. The *degree* of a vertex v is $\deg(v) := |\mathrm{in}(v)| + |\mathrm{out}(v)|$. To distinguish between elements of different graphs, we will use the subscript notation $G = (V_G, E_G, s_G, t_G)$.

We say that a vertex v is a *successor* of u if the there exists an edge e such that $s(e) = u$ and $t(e) = v$. A pair of vertices are *connected*, written $u \sim v$, if they lie in the reflexive, symmetric, transitive closure of the successor relation. The equivalence classes $V/\sim$ are the *connected components* of G. We write $|v|$ to denote the equivalence class containing the vertex v; we write $[v]$ to denote the subgraph determined by $|v|$.

Definition 2 (Graph morphism) Given graphs G and H, a *graph morphism* $f : G \to H$ consists of functions $f_E : E_G \to E_H$ and $f_V : V_G \to V_H$ such that:

$$s_H \circ f_E = f_V \circ s_G, \tag{1}$$

$$t_H \circ f_E = f_V \circ t_G. \tag{2}$$

These conditions ensure that the structure of the graph is preserved.

Definition 3 (Open graph, open graph morphism) An *open graph* $\Gamma = (G, \partial G)$ consists of a directed graph G, and a set of vertices $\partial G \subseteq V_G$, such that for each $v \in \partial G$ we have $\deg(v) = 1$. The set ∂G is called the *boundary* of Γ; those vertices in $V_G \setminus \partial G$ are called the *interior* of Γ, written $\mathrm{Int}\, G$.

Given open graphs $(G, \partial G)$ and $(H, \partial H)$ a graph morphism $f : G \to H$ defines a *morphism of open graphs* $f : (G, \partial G) \to (H, \partial H)$ if $f_V(v) \in \partial H \Rightarrow v \in \partial G$ for all v in V_G.

We will refer to an open graph $(G, \partial G)$ simply as G when it is unambiguous to do so.

Definition 4 (Strict map) Let $f : (G, \partial G) \to (H, \partial H)$ be an open graph morphism say that f is *strict* if $\forall e \in E_H$, if $s_H(e) \in f_V(\mathrm{Int}\, G)$ or $t_H(e) \in f_V(\mathrm{Int}\, G)$ then $\exists e' \in E_G$ such that $f_E(e') = e$.

Strictness ensures that there are no additional edges connected to vertices in the image of $\mathrm{Int}\, G$.

We emphasise two points about the distinction between interior and boundary nodes for open graphs. We view graphs as computational objects, built up by connecting smaller objects together; we view the interior vertices as computational primitives. Strict maps ensure that the interior structure—the types and connections of the vertices—is preserved. The boundary of an open graph defines the *interface* of the system; the boundary nodes indicate this interface, and do not carry computational meaning. Hence boundary nodes have degree one: they simply mark an edge where something may be connected. Morphisms of open graphs preserve this view by not allowing interior nodes to be mapped to the boundary.

We can also view graphs as topological spaces. In this case the boundary nodes can be seen as points which lie outside the space but are needed to define it, similar to the end points of an open interval. From this point of view, morphisms of open graphs are *continuous*. What then are the open sets of this space? Open subgraphs arise via two graph operations: removing connected components from the graph, and removing single points. We note that it suffices to consider removing points which lie on edges, since vertex removal can be simulated by disconnecting the vertex and then removing the resultant component. Since we are indifferent to which point on the edge is removed, we introduce the notion of *splitting an edge*. The intuition is that by removing a point from the middle of the edge e, we introduce two new boundary points.

Definition 5 (Splitting an edge) Let G be an open graph, and suppose $e \in E_G$; we define $G_{\ltimes e}$, the *splitting of G on e*, via the graph $G' = (V_G + \{e_1, e_2\}, (E_G \setminus \{e\}) + \{e_1, e_2\}, s', t')$, where e_1, e_2 do not occur in V_G or E_G, and s' and t' are defined such that

- $s'(e_1) = s_G(e), t'(e_1) = e_1$;
- $s'(e_2) = e_2, t'(e_2) = t_G(e)$;

and they otherwise agree with s_G and t_G respectively. Then $G_{\ltimes e} := (G', \partial G + \{e_1, e_2\})$.

We define a canonical morphism i embedding $G_{\ltimes e}$ back into G as follows:

- $i_V(e_1) = t_G(e_1)$; $i_V(e_2) = s_G(e_2)$; and $i_V(v) = v$ otherwise.
- $i_E(e_1) = i_E(e_2) = e$; and $i_E(e') = e'$ otherwise.

Clearly, i is injective on the portion of $G_{\ltimes e}$ excluding e_1 and e_2, and it is strict.

Definition 6 (Removing a component) Let $\Gamma = (G, \partial G)$ be an open graph, and suppose that $v \in V_G$. The graph obtained by removing the component $[v]$ is denoted $\Gamma - [v] := (G - [v],\ \partial G \setminus (|v| \cap \partial G)$ where the underlying graph is given by:

$$G - [v] = (V_G \setminus |v|\,,\quad E_G \setminus s_G^{-1}(|v|),\quad s_G|_{(E_G \setminus s_G^{-1}(|v|))},\quad t_G|_{(E_G \setminus s_G^{-1}(|v|))}\ ).$$

Writing $G + H$ for the disjoint union of open graphs, it is immediate that we have the isomorphism $G \cong [v] + (G - [v])$, and hence that the coproduct injection $\mathrm{in}_2 : (G - [v]) \hookrightarrow [v] + (G - [v])$ provides a canonical map back into the original graph. A further consequence is that every graph is equivalent to the disjoint union of its connected components.

It is easy to show that the operations of splitting edges and removing components generalise to sets of edges and vertices, and further that any sequence of such operations can be standardised so that all the splittings come first.

Definition 7 (Open subgraph) Let G be an open graph; then each pair (F, U) with $F \subseteq E_G$ and $U \subseteq V_G$ defines an *open subgraph* $G_{\ltimes F} - [U]$.

Every open subgraph of G has a canonical map embedding it back into G, constructed from the canonical embeddings at each step; it is strict, and injective everywhere except the new edges and boundary nodes introduced by splittings.

Definition 8 (Exact embedding) We call an open graph morphism $f : (G, \partial G) \hookrightarrow (H, \partial H)$ an *exact embedding* if:

1. f is strict;
2. f_E is injective;
3. f_V is injective; and,
4. $f_V(v) \in \partial H \Leftrightarrow v \in \partial G$, for all $v \in V_G$..

Definition 9 (Matching) We say that G *matches* H if there exists an open subgraph H' of H, and an exact embedding $e : G \hookrightarrow H'$. In this case we write $G \leq H$; we write $[\![G]\!]$ for the set of all graphs which G matches.

Proposition 1 *Let G, H, and K be open graphs. Then*

1. $G \leq G$;
2. $G \leq H$ *and* $H \leq K$; *then* $G \leq K$;
3. *If* $G \leq H$ *and* $H \leq G$ *then* $G \cong H$.
4. $G \leq H$ *iff* $[\![H]\!] \subseteq [\![G]\!]$.

Proof The first property follows from the fact that the identity map is an exact embedding; the second and fourth hold because exact embeddings are closed under composition. For the third property: since we can exactly embed G into a subgraph of H, and vice versa, we must have that these subgraphs are isomorphic to the original graphs; from here the isomorphism between G and H is easily constructed. □

4 Graphs with exterior nodes

We now present a generalisation of the open graphs described in the previous section. The purpose of this generalisation is to offer more precise control over matching: a graph G will match H when it can be exactly embedded *in a given configuration*.

Definition 10 (Extended open graph) An *extended open graph*, henceforth abbreviated *e-graph*, is pair (G, X) where G is a graph and $X \subseteq V_G$ is a distinguished set of vertices. The elements of X are called the *exterior nodes* of G; those vertices in $V_G \setminus X$ are called the *interior*.

An e-graph morphism $f : (G, X) \to (H, Y)$ is a graph morphism such that $f_V(v) \in Y$ implies $v \in X$ for all $v \in V_G$.

The exterior nodes of an e-graph generalise the boundary nodes of an open graph and are viewed in the same way: as points outside the graph. As well as marking the edge of the graph, exterior points also constrain how the edges incident at them may be embedded into a larger graph: they must meet at the same point. This will be made explicit below.

Definition 11 (Splitting a vertex) Let (G, X) be an e-graph with $x \in V_G$; we define a new e-graph $G_{\ltimes x}$ by *splitting the vertex* x as $G_{\ltimes x} := (G', (X \setminus \{x\}) + \mathrm{in}(x) + \mathrm{out}(x))$ where $G' := ((X \setminus \{x\}) + \mathrm{in}(x) + \mathrm{out}(x)), E_G, s', t')$ and

$$s'(e) = e \quad \text{if } e \in \mathrm{out}(x),$$
$$t'(e) = e \quad \text{if } e \in \mathrm{in}(x),$$
$$s'(e) = s_G(e), \quad t'(e) = t_G(e) \text{ otherwise.}$$

We can define a canonical map $i : G_{\ltimes x} \to G$ by $i_E = \mathrm{id}$, and $i_V(v) = x$ if $v \in \mathrm{in}(x) + \mathrm{out}(x)$ and $i_V(v) = v$ otherwise. Evidently, the splitting operation can be lifted to sets of vertices, so we may write $G_{\ltimes U}$ when $U \subseteq V_G$. We define a relation $\heartsuit$ over the vertices of $G_{\ltimes U}$ by $v_1 \heartsuit v_2$ iff $i_V(v_1) = i_V(v_2)$.

Definition 12 (Relaxation of an extended graph) Let (G, X) be an e-graph; define its *relaxation*, $\mathrm{relax}(G) := G_{\ltimes X}$.

Essentially, $\mathrm{relax}(G)$ is the closest approximation of G as an open graph. Note that if G is an open graph itself—i.e., all its exterior points are of degree one—then $\mathrm{relax}(G) \cong G$.

Definition 13 (Matching an extended graph) We say that (G, X) matches (H, Y) when there exists H', an open subgraph of $\mathrm{relax}(H)$, and an exact embedding $f : \mathrm{relax}(G) \to H'$ such that if $v \heartsuit u$ in $\mathrm{relax}(G)$ then $f(v) \heartsuit f(u)$ in H'. In this case we write $G \leq_e H$. As before we define $[\![G]\!]_e := \{H | G \leq_e H\}$. Matching is illustrated graphically in Fig. 2.

Proposition 2 $G \leq_e H \Leftrightarrow [\![G]\!]_e \supseteq [\![H]\!]_e$.

4.1 Composing graphs

We now introduce a general method for composing graphs which we call *plugging*; it works equally well for graphs, open graphs, and e-graphs. We will give here the definition for the case of e-graphs, but the reader will have no difficulty in modifying the definitions for the other cases.

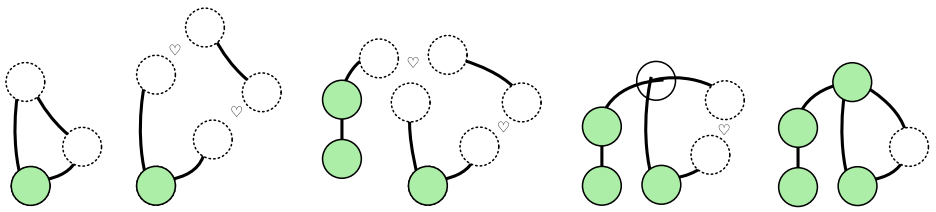


Fig. 2 An illustrative example showing the steps involved in the e-graph matching $G \leq_e H$. This involves operations on G and H, and finally finding an exact embedding between $relax(G)$ and H'

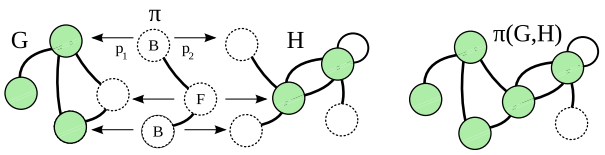


Fig. 3 The plugging of G and H via the two-sided e-graph π with embeddings p_1 and p_2

Let (G, X) be an e-graph, and suppose that we have a partition of its vertices $V_G = F + B$ into a *front set* and a *back set*; in this case call (G, X, F, B) a *two-sided e-graph*.

Definition 14 (Plugging) Let (π, V_π, F, B) be two-sided e-graph, with a pair of embeddings $p_1 : (\pi, V_\pi) \to (G, X)$ and $p_2 : (\pi, V_\pi) \to (H, Y)$ such that $p_1(F) \subseteq X$ and $p_2(B) \subseteq Y$. Then we define the *plugging*, $\pi^{p_1}_{p_2}(G, H)$, via the pushout:

$$\begin{array}{ccc} \pi & \xrightarrow{p_1} & G \\ {\scriptstyle p_2}\downarrow & & \downarrow \\ H & \longrightarrow & \pi^{p_1}_{p_2}(G, H) \end{array}$$

The result of the plugging is the minimal graph matched by both G and H such the two copies of π are identified. We will simply write $\pi(G, H)$ for the plugging, taking the embeddings p_1 and p_2 as given. An example illustrating plugging is given in Fig. 3.

Proposition 3 *Let π, G, and H be as above, and let K be some e-graph; then*

- $\pi(G, H) \cong \pi(H, G)$;
- $G \leq_e \pi(G, H)$ *and* $H \leq_e \pi(G, H)$;
- $K \leq_e G$ *implies* $K \leq_e \pi(G, H)$;

5 Compact closed categories

Definition 15 (Compact closed category) A strict symmetric monoidal category [3] is called compact closed [12] when each object A has a chosen dual object A^*, and morphisms

$$d_A : I \to A^* \otimes A \qquad e_A : A \otimes A^* \to I$$

where I is the tensor identity of the compact closed category, such that

$$A \cong A \otimes I \xrightarrow{\mathrm{id}_A \otimes d_A} A \otimes A^* \otimes A \xrightarrow{e_A \otimes \mathrm{id}_A} I \otimes A \cong A = \mathrm{id}_A \tag{3}$$

$$A^* \cong I \otimes A^* \xrightarrow{d_A \otimes \mathrm{id}_{A^*}} A^* \otimes A \otimes A^* \xrightarrow{\mathrm{id}_{A^*} \otimes e_A} A^* \otimes I \cong A^* = \mathrm{id}_{A^*} \tag{4}$$

Every arrow $f : A \to B$ in a compact closed category $\mathcal{C}$ has a *name* and *coname*:

$$\ulcorner f \urcorner : I \to A^* \otimes B, \qquad \llcorner f \lrcorner : A \otimes B^* \to I,$$

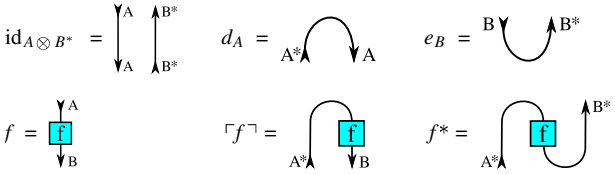


Fig. 4 Compact closed structure as graphs

which are constructed as $\ulcorner f \urcorner = (\mathrm{id}_{A^*} \otimes f) \circ d_A$ and $\llcorner f \lrcorner = e_B \circ (f \otimes \mathrm{id}_{B^*})$. Hence there are natural isomorphisms $\mathcal{C}(A, B) \cong \mathcal{C}(I, A^* \otimes B) \cong \mathcal{C}(A \otimes B^*, I)$ making $\mathcal{C}$ monoidally closed.[5] Furthermore, f has a dual, $f^* : B^* \to A^*$, defined by

$$f^* = (\mathrm{id}_{A^*} \otimes e_B) \circ (\mathrm{id}_{A^*} \otimes f \otimes \mathrm{id}_{B^*}) \circ (d_A \otimes \mathrm{id}_{B^*})$$

By virtue of equations (3) and (4), $f^{**} = f$. Thus $(\cdot)^*$ lifts to an involutive functor $\mathcal{C}^{\mathrm{op}} \to \mathcal{C}$, making $\mathcal{C}$ equivalent to its opposite.

5.1 Graph representations for compact closed categories

Open graphs with certain additional structure give a representation for compact closed categories; we now give an overview of this construction. The details omitted here can be found in [9]. Pictorial representations are in Fig. 4. We make the convention that the domain of an arrow is at the top of the picture, and its codomain is at the bottom.

A *concrete graph* Γ is 5-tuple $(G, \mathrm{dom}\,\Gamma, \mathrm{cod}\,\Gamma, <_{\mathrm{in}(\cdot)}, <_{\mathrm{out}(\cdot)})$ where:

- $G = (V, E, s, t)$ is a graph;
- $\mathrm{dom}\,\Gamma$ and $\mathrm{cod}\,\Gamma$ are totally ordered disjoint sets of degree one vertices of G. Therefore the union of these sets is the *boundary* of the open graph $(G, \mathrm{dom}\,\Gamma + \mathrm{cod}\,\Gamma)$;
- $<_{\mathrm{in}(\cdot)}$ is a family of maps, indexed by V such that $<_{\mathrm{in}(v)} : \mathrm{in}(v) \xrightarrow{\cong} \mathbb{N}_k$ where $k = |\mathrm{in}(v)|$.
- $<_{\mathrm{out}(\cdot)}$ is a family of maps, indexed by V such that $<_{\mathrm{out}(v)} : \mathrm{out}(v) \xrightarrow{\cong} \mathbb{N}_{k'}$ where $k' = |\mathrm{out}(v)|$.

Since the sets $\mathrm{dom}\,\Gamma$ and $\mathrm{cod}\,\Gamma$ consist of vertices of degree one, we can assign a polarity to each one: $v \mapsto +$ if the edge incident at v is an incoming edge; $v \mapsto -$ otherwise. Hence $\mathrm{cod}\,\Gamma$ and $\mathrm{dom}\,\Gamma$ are *ordered signed sets*. Given any ordered signed set S we write S^* for the same ordered set with the opposite signing. Given two such sets we can define their disjoint union $R + S$ as the disjoint union of the underlying sets, inheriting the signing and the order from R and S, with the convention that $r < s$ for all $r \in R, s \in S$.

Proposition 4 *Concrete graphs form a compact closed category whose objects are ordered signed sets and whose arrows $f : A \to B$ are concrete graphs with* $\mathrm{cod}\, f = B$ *and* $\mathrm{dom}\, f = A^*$.

[5] Compact closed categories are models of multiplicative linear logic where $A \multimap B$ is defined as $A^{\perp} \otimes B$.

For each ordered signed set A, the identity map id_A has $\mathrm{dom}\,\mathrm{id}_A = A^*$ and $\mathrm{cod}\,\mathrm{id}_A = A$; its underlying graph has $E = A$ and $V = A^* + A$ with $t(a) = a$ and $s(a) = a^*$. Given a pair of concrete graphs $f : A \to B$ and $g : B \to C$ their composition $g \circ f : A \to C$ is constructed by merging the two graphs, erasing the vertices of $\mathrm{cod}\, f$ and $\mathrm{dom}\, g$ (called the *boundary vertices*), and identifying the edges previously incident at the deleted vertices. (Due to the opposite polarity of the domain and codomain the edges have compatible direction.) The tensor product on objects A, B is simply $A + B$; given $f : A \to B$, $g : C \to D$, the graph of $f \otimes g$ is the disjoint union of the graphs of f and g. The unit for the tensor is the empty set. The morphisms $d_A : I \to A^* \otimes A$, $e_A : A \otimes A^* \to I$ have the same underlying graph as id_A, but $\mathrm{dom}\, d = \emptyset$, $\mathrm{cod}\, d = A^* + A$, $\mathrm{dom}\, e = A + A^*$ and $\mathrm{cod}\, e = \emptyset$.

Remark 3 Although we have not written it explicitly, both composition and tensor can both be expressed as plugging. The tensor is the plugging along the empty graph, while composition is plugging along an identity graph. In fact, one can define another compact closed category whose objects are two-sided graphs and whose arrows are e-graphs; sadly, space does not allow it to be described here.

This category captures exactly the axioms for compact closed structure, in the sense that any freely generated compact closed category can be represented by concrete graphs. We will consider a collection of basic terms[6] F whose types are vectors of some set of basic types T. Then:

Definition 16 (Labelling) A T, F-*labelling* θ for a concrete graph Γ is a pair of maps $\theta_T : E \to T$ and $\theta_F : (V - \mathrm{cod}\,\Gamma - \mathrm{dom}\,\Gamma) \to F$ such that for each vertex v, if $\mathrm{in}(v) = \langle a_1, \ldots, a_n \rangle$ and $\mathrm{out}(v) = \langle b_1, \ldots, b_m \rangle$ then

$$\theta v : \langle \theta a_1, \ldots, \theta a_n \rangle \to \langle \theta b_1, \ldots, \theta b_m \rangle$$

We say a concrete graph Γ is T, F-*labellable* if there exists an T, F-labelling for it; and if θ is a labelling for Γ, then the pair (Γ, θ) is called a T, F-*labelled graph*.

The T, F-labelled graphs form a compact closed category in the same way as the concrete graphs, subject to the further restriction that arrows are composable only when their labellings agree.

Theorem 2 *Let $\mathcal{C}$ be a compact closed category, freely generated by some set of arrows F and ground types T; then $\mathcal{C}$ is equivalent to the category of T, F-labelled graphs.*

Given a compact closed category $\mathcal{C}$ generated by some basic set of operations, the arrows of $\mathcal{C}$ have a canonical representation as labelled graphs. A consequence of the theorem is then that two arrows are equal by the equations of the compact closed structure if and only if their graph representations are equal.

As a final remark before moving on, note that the external structure of a vertex in a concrete graph is essentially the same as that of a complete graph; hence one can consistently view subgraphs as vertices, and abstract over the their internal structure.

[6] See [9] for a more thorough description of the nature of the terms.

6 !-boxes

To support reasoning with spiders we introduce the operation *!-boxing* (pronounced bang-boxing), on graph representations. Given a graph representation, this introduces a new notation, that of outlining a set of nodes (!-boxing them). We then introduce matching which formalises the idea that a !-box graph can have an arbitrary number of copies of the !-boxed nodes where every copy connects in the same way to the nodes outside the !-box.

Definition 17 (!-box graph) A *!-box graph* is a pair $(G, \mathcal{B}_G)$ where G is a graph and $\mathcal{B}_G$ is a set of disjoint subsets of V_G, called the *!-boxes* of G.[7]

Definition 18 (!-box matching) We say that $(G, \mathcal{B}_G)$ *matches* $(H, \mathcal{B}_H)$, written $(G, \mathcal{B}_G) \leq_! (H, \mathcal{C})$, when $(H, \mathcal{B}_H)$ can be obtained from $(G, \mathcal{B}_G)$ by the following operations on graphs, performed in order:

copy(C,$(G, \mathcal{B}_G)$): C is a set of functions where each member, c, is mapping from $\mathcal{B}_G$ to natural numbers. Each bang box, b, is copied $c(b)$ times. Any edges between a node, n, inside a !-box b, and a node, m, outside it, get copied so that there is a new edge from m to the new copy of n. When $c(b) = 0$, we call it *killing* as all nodes in the !-box get removed with any incident edges.

drop(K,$(G, \mathcal{B}_G)$): removes the subset, K of the !-boxes, but leaves their contents in the graph.

merging(M,$(G, \mathcal{B}_G)$): given a set of disjoint subsets of unconnected !-boxes, M, merging simply unions the members of each subset of !-boxes into a single !-box.

An illustration of matching with these operations is given in Fig. 5.

Proposition 5 *!-box matching is partial order.*

Reflexivity comes from the trivial matching (no copying, no dropping and no merging). Transitivity can be proved by constructing combined matching from two existing matches: killing a !-box that was constructed from a copy simply avoids copying the !-box in the first place, copying after merging simply involves additional copying beforehand and merging at the end. Antisymmetry can be proved by constructing from arbitrary matches $G \leq_! H$ and $H \leq_! G$ the trivial matching. The construction involves cancelling extra copying with killing until there is no copying.

We give a formal semantics to !-box graphs in terms of a set of graphs in the underlying representation. In particular, we denote the interpretation of a !-box graph $(G, \mathcal{B})$ by $[\![(G, \mathcal{B})]\!]_!$ and say that its members are instances.

Definition 19 (!-box semantics) $[\![(G, \mathcal{B})]\!]_!$ is the set of graphs matched by the !-box graph that have no !-boxes: $[\![(G, \mathcal{B})]\!]_! = \{H \mid (G, \mathcal{B}) \leq_! (H, \emptyset)\}$

[7] More expressive notions of !-boxes with nested and overlapping node sets provide interesting additional expressivity, but are not required for the system we formalise here.

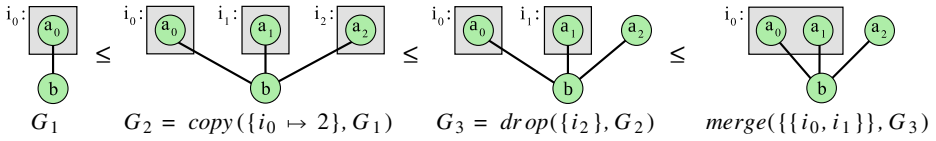


Fig. 5 An illustration of !-box graph matching using the !-box operations. This involves first copying !-box i_0 twice, then dropping i_3, and finally merging i_0 and i_1

Observe that every instance of a !-box graph can be defined by pairing each !-box with the natural number that defines how many copies are made of it. Thus $[\![G]\!]_!$ is isomorphic to the set of k-tuples of natural numbers, where k is the number of !-boxes. The need for the !-box matching operation, rather than using a direct k-tuple interpretation, is to allow matching between !-box graphs, and thus to provide a mechanism for derived rules.

Proposition 6 !-Matching respects !-box semantics*:* $G \leq_! H \Leftrightarrow [\![G]\!]_! \supseteq [\![H]\!]_!$. *The proof is a simple consequence from the* $\leq_!$ *being a partial order and the definition of* $[\![G]\!]_!$ *being a subset of the graphs that match* G.

Because !-box graphs correspond to a countably infinite number of concrete graphs, matching cannot be implemented by simply unfolding all interpretations. We now prove that matching is still decidable.

Theorem 3 *!-box graph Matching is decidable.*

The key observation is that a graph, G, will never match a graph with fewer nodes except by killing. Thus the copying(and killing) operations on G can be bounded by the number of nodes in the graph it is being matched against. While this gives a generate and test style algorithm, it is not efficient. The intuition for an efficient algorithm is to search through G incrementally increasing the matched part.

7 Reasoning with graph patterns

The representation formed by adding !-boxes to e-graphs, which we call *graph patterns*, allows us to express, in a finite way, certain infinite families of equations between e-graphs. In particular, the Spider Theorem can now be represented as shown in Fig. 6. We now define graph patterns and then describe how they can be used to develop a formal system for reasoning about compact closed categories.

Definition 20 (Graph pattern) A graph pattern is a !-boxed e-graph, i.e., a pair $(G, \mathcal{B}_G)$, where G is an e-graph and $\mathcal{B}_G$ is a set of !-boxes for G.

Definition 21 (Graph pattern interpretation) A graph pattern G represents the set of open concrete graphs: $[\![G]\!]_p = \bigcup\{[\![G']\!]_e \,.\, G' \in [\![G]\!]_!\}$

This definition allows us to lift matching and plugging from e-graphs through !-boxes to develop analogous definitions for graph patterns.

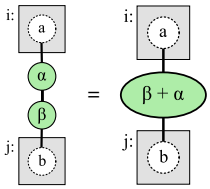


Fig. 6 The Spider Theorem, from Section 2, expressed formally using graph patterns. The !-boxes are named *i* and *j*. The variable nodes are white and named *a* and *b*. The non-variable node data (the angle) is written inside the node when non-zero

Definition 22 (Graph pattern matching) Given graph patterns G and H we define the matching relation $G \leq_p H$ as

$$G \leq_p H = \exists G'.\, G \leq_! G' \wedge G' \leq_e H$$

Plugging together of two graph patterns is an extension of e-graph plugging where !-box membership is respected. It restricts identification of nodes in a plugging to cases when their !-boxes are also be identified.

Definition 23 (Graph pattern plugging) Let $(G, \mathcal{B}_G)$ and $(H, \mathcal{B}_H)$ be graph patterns; then given a plugging $\pi_q^p(G, H)$, we have a valid graph pattern $(\pi_q^p(G, H), \mathcal{B}_G \cup \mathcal{B}_H)$ if and only if $\forall b \in \mathcal{B}_G$ if $\exists v \in V_\pi$ such that $p(v) \in b$ then $\forall w \in b\,, \exists v' \in V_\pi$ such that $p(v') = w$ and $\exists b' \in \mathcal{B}_H$ such that $q(v') \in b'$.

These definitions allow the properties of plugging for e-graphs to lift naturally to graph patterns.

The language of graph patterns forms a *meta-level* framework for reasoning about compact closed categories. The meta-level provides generic machinery to manipulate graphs and derive new rules. Following the terminology of logical frameworks, such as Isabelle [16], we call specification of additional structure, beyond the meta-level, the *object-level*. In our setting, this involves providing a set of equations between graphs. These equations are the axioms for the object system. For example, in Section 8 we define an object level theory for reasoning about quantum computation based on the graphical calculus introduced in Section 2. In addition to the axioms, the object level can also provides an appropriate matching or unification operation for data in the nodes and edges.

We now describe the meta-level framework, noting the conditions for a rule to be valid, and prove the system's adequacy. The resulting system forms the basis for an interactive proof assistant that supports reasoning in compact closed categories.

7.1 Equational rules

In our framework, the axioms defined by an object-level model, as well as derived rules, are pairs of graph patterns. The elements of the pair represent the left and right hand sides of an equation. Rules are declarative in that they denote a set of equations between the underlying formalism of concrete graphs.

The intuitive idea of substitution with a rule is to replace a subgraph that matches the left hand side with the rule's right hand side. However, not all pairs of graphs make a valid rule with respect to the underlying semantics. For an equation to be well defined with respect to the compact closed structure it must not be possible to change the type (the boundary nodes in the domain and co-domain) of an concrete graph graph by rewriting. Mapping this restriction back to rules on graph patterns results in the following conditions:

- There must be a isomorphism between exterior nodes in the left and right hand sides.
- Rules must also define a partial mapping between !-boxes on the left and right hand sides. The intuition for this mapping is that the unfolding used when matching a !-box on the left, is applied to the mapped !-box on the right before replacement.
- When an exterior node appears within a !-box on one side of a rule, it must also appear under a mapped !-box on the other side.

For notational convenience, we annotate !-boxes and exterior nodes in a graph with unique names. For example, see Fig. 6 which shows the Spider Theorem, where the mapping between !-boxes is represented by !-boxes having the same name. Similarly, the isomorphism between exterior nodes is captured by the set of exterior node names being equal.

7.2 Meta-level logic and derived rules

Having defined what makes a valid rule, we now present the meta-logic of the framework. This is quite simple as it only involves dealing with equations:

$$\frac{A = B \in \Gamma}{\Gamma \vdash A = B}\text{trivial} \qquad \frac{}{\Gamma \vdash A = A}\text{refl} \qquad \frac{\Gamma \vdash A = B}{\Gamma \vdash B = A}\text{sym}$$

$$\frac{\Gamma \vdash A = B \quad \Gamma \vdash C = D}{\Gamma \vdash (C = D[A/B])\theta}\text{subst} \qquad \frac{\Gamma \vdash A = B \quad \Gamma \vdash C = D}{\Gamma \vdash \pi(A, C) = \pi(B, D)}\text{plug}$$

where Γ is the set of object-level axioms, $D[A/B]$ is the graph D with a matching of A replaced by B, and θ is a matching or unification result defined by object level matching for the node and edge data.

For the reflexivity rule (*refl*), we assume that A is a well-formed pattern graph. This rule allows a new graph to be introduced. The *plug* rule allows graphs to be put together to form larger graphs by plugging in an analogous way to composition in functional programming. By then applying the *subst* rule, intermediate results are derived which can themselves be used to rewrite other rules and conjectures. Given that the axioms in Γ meet the validity conditions described earlier, the rules all preserve the validity of equations and thus the system as a whole ensures only valid rules are derived.

Given an object-level formalism, a set of equations can be applied automatically to simplify a graph or simulate computation. For such rewriting to terminate, a suitable left-to-right ordering on rules needs to be observed, such as a decrease in the size of

the graph. An initial study into this issue has been investigated by Kissinger [13]. An example of simulating a quantum computation is given in Section 8.

7.3 Lifting axioms and adequacy

The axioms of an object formalism come from the semantics of the underlying system. For instance, the equations given in Fig. 1 can be proved by matrix calculations in the underlying model. When such rules are expressed as graph patterns, we replace the concrete representation's boundary nodes with exterior nodes. This operation is called *lifting*. When a rule contains exterior nodes, the equation on graph patterns corresponds to an infinite family of equations between concrete graphs. Thus we might worry that the lifted equations express too much: they may allow rewrites which are not true. We call the property that the lifted representation is a conservative extension of the initial theory *adequacy*. For models of compact closed categories, the proof of adequacy is quite simple: given an equation between concrete graphs, $G = H$, we observe that every instance of the lifted equation has a subgraph matching the original equation such that the instance can be derived by plugging.

8 A case study in quantum computation

The model of quantum computation introduced in Section 2 provides an object formalism for our meta-level framework. In particular, the object level axioms come from lifting the equations in Fig. 1 and from the formalisation of the Spider Theorem in Fig. 6. Our model of quantum computation requires no data for the edges. The nodes on the other hand are either H (a Hadamard gate) with no additional data, or a Pauli operator which has an *angle* and a *colour*. The colour is darker red for the X basis operations and a lighter green for those the Z basis. For their part, angles are expressed as rational numbers which correspond to the coefficient of π in the underlying matrix.

To allow composition of rules to compute the resulting angles we give the X and Z nodes an *angle expression*. When a node is within a !-box, the expression is a single *angle-variable* which gets instantiated to a new angle-variable in each of the unfoldings of the !-box. When a node is not within a !-box, the angle-expression is a mapping from a set of angle-variables to the corresponding rational coefficient. When an angle-expression contains an angle-variable within a !-box, this is interpreted as a sum of the variables that result from its unfolding.

This rather simple expression language has a normal form by ordering the angle-variable by name. Matching then results in angle-variables being instantiated and the expressions in all affected nodes are then (re)normalised. An additional implementation detail must also be observed for the substitution rule: it must ensure that angle-variables in the rule being applied are distinct from those in the expression being rewritten.

The quantum Fourier transform is among the most important quantum algorithms, forming an essential part of Shor's algorithm [23], famous for providing polynomial factoring. In our graph pattern calculus this circuit becomes the top-left graph in Fig. 7. This figure shows how computation can be symbolically performed

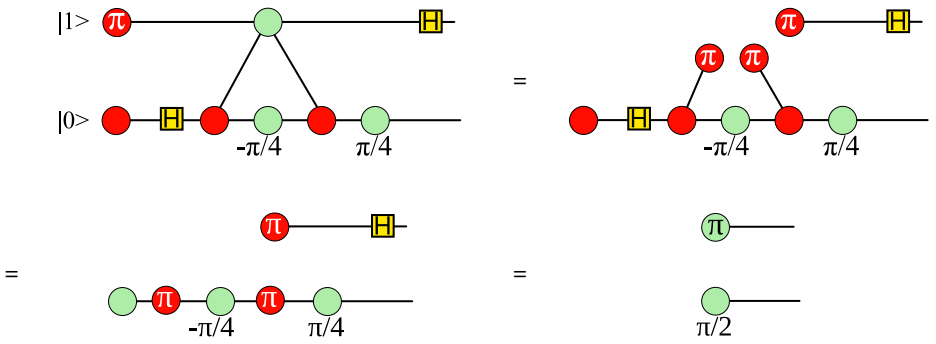


Fig. 7 An example computation of the quantum fourier transform with inputs 1 and 0, performed symbolically by rewriting

by rewriting with the lifted equations from Fig. 1 and the graph pattern version of the Spider Theorem.

9 Related work

There are several foundational approaches to graph transformation, including algebraic approaches [10], node-label controlled [11], matrix based [24], and programmed graph replacement [20]. These provide general ways of understanding graph transformations which can then be implemented to provide machinery for a specific application. However, systems based on these theories do not provide machinery for the semantics of compact closed categories. The distinctive feature of our form of graph rewriting is that the graphs capture the structural properties of compact closed categories and this is respected by rewriting: rewriting is compositional and preserves the type of the rewritten subgraph. This allows us to define a plugging operation over which rewriting distributes.

Bundy and Richardson have described an account of ellipses notation for lists [4]. Various authors have also considered ellipses representations for matrices [17, 22], and more recently, Prince, Ghani and McBride have developed a general formalism for ellipses notation using Containers [18]. Providing machinery for rewriting of graphs with ellipses notation, which is needed to represent the Spider Theorem, is a novel contribution of our approach to graph rewriting.

We note that our graphical notation has little connection to *graph states* as used in various approaches to measurement-based quantum computation [19]. In that approach the graph structure is used to provide a description of the entanglement in a state: it does not provide a complete description of a computation.

10 Conclusions and further work

We extended the representation of compact closed categories as graphs to provide a more expressive account of the interface offered by an open graph. This representation enjoys a plugging operation that has sequential and parallel composition

as special cases. We also described a formalism for ellipses notations on graphs and showed that matching is decidable. These representations, together, provide a rich language of *graph patterns*. This language then forms the foundation for a simple meta-logic for reasoning about models of compact closed categories.

We use the graphical language to extend existing graphical calculi for quantum computation. In particular, informal reasoning with graphical equations that contain ellipses notation, such as the Spider Theorem, now have a formal graphical representation. We illustrate this by showing how computation can be performed by symbolic graphical rewriting.

We are left with several exciting avenues for further research. These include considering confluence results for sets of rewrite rules, complexity results for matching, increasing the expressiveness of the representation for graph-patterns, and finding a complete set of rewrite rules for graphical models of quantum computation. Finally, details of an implementation can be found online at http://dream.inf.ed.ac.uk/projects/quantomatic.

Acknowledgement This work was funded by EPSRC grants EPE/005713/1 and EP/E045006/1.

References

1. Abramsky, S., Coecke, B.: A categorical semantics of quantum protocols. In: LICS 2004, pp. 415–425. IEEE Computer Society, Los Alamitos (2004)
2. Abramsky, S., Gay, S., Nagarajan, R.: Interaction categories and the foundations of typed concurrent programming. In: Broy, M. (ed.) Proceedings of the 1994 Marktoberdorf Summer School on Deductive Program Design, pp. 35–113. Springer, New York (1996)
3. Asperti, A., Longo, G.: Categories, Types and Structures. MIT, Cambridge (1991)
4. Bundy, A., Richardson, J.: Proofs about lists using ellipsis. In: Proc. of the 6th LPAR. LNAI, vol. 1705, pp. 1–12. Springer, New York (1999)
5. Coecke, B.: Kindergarten Quantum Mechanics. Lecture Notes (2005)
6. Coecke, B., Duncan, R.: Interacting quantum observables. In: ICALP 2008. LNCS (2008)
7. Coecke, B., Paquette, E.O.: POVMs and Naimark's theorem without sums. In: Proc. of the 4th International Workshop on Quantum Programming Languages (2006)
8. Coecke, B., Pavlovic, D.: Quantum measurements without sums. In: The Mathematics of Quantum Computation and Technology, CRC Applied Mathematics & Nonlinear Science. Taylor and Francis, London (2007)
9. Duncan, R.: Types for quantum computation. Ph.D. thesis, Oxford University (2006)
10. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. EATCS Series). Springer, New York (2006)
11. Janssens, D., Rozenberg, G.: Graph grammars with node-label controlled rewriting and embedding. In: Proc. of the 2nd International Workshop on Graph-Grammars and Their Application to Computer Science, pp. 186–205. Springer, New York (1983)
12. Kelly, G.M., Laplaza, M.L.: Coherence for compact closed categories. J. Pure Appl. Algebra **19**, 193–213 (1980)
13. Kissinger, A.: Graph rewrite systems for complementary classical structures in y-symmetric monoidal categories. Master's thesis, University of Oxford (2008)
14. Kock, J.: Frobenius Algebras and 2-D Topological Quantum Field Theories. Cambridge University Press, Cambridge (2003)
15. Pati, A.K., Braunstein, S.L.: Impossibility of deleting an unknown quantum state. Nature **404**, 164–165 (2000)
16. Paulson, L.C.: Isabelle: A Generic Theorem Prover. Springer, New York (1994)
17. Pollet, M., Kerber, M.: Intuitive and formal representations: the case of matrices. In: MKM'04. LNCS, vol. 3119, pp. 317–331. Springer, New York (2004)
18. Prince, R., Ghani, N., McBride, C.: Proving properties about lists using containers. In: FLOPS. LNCS, vol. 4989, pp. 97–112. Springer, New York (2008)

19. Raussendorf, R., Briegel, H.J.: A one-way quantum computer. Phys. Rev. Lett. **86**, 5188–5191 (2001)
20. Schfürr, A.: Programmed Graph Replacement Systems, pp. 479–546. World Scientific, River Edge (1997)
21. Selinger, P.: Dagger compact closed categories and completely positive maps. In: Proc. of the 3rd International Workshop on Quantum Programming Languages (2005)
22. Sexton, A.P., Sorge, V.: Semantic analysis of matrix structures. In: ICDAR '05: Proceedings of the Eighth International Conference on Document Analysis and Recognition, pp. 1141–1145. IEEE Computer Society, Washington, DC (2005)
23. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Sci. Statist. Comput. **26**(5), 1484–1509 (1997)
24. Velasco, P.P.P., de Lara, J.: Matrix approach to graph transformation: matching and sequences. In: ICGT. LNCS, vol. 4178, pp. 122–137. Springer, New York (2006)
25. Wootters, W., Zurek, W.: A single quantum cannot be cloned. Nature **299**, 802–803 (1982)