THE UNIVERSITY OF CHICAGO

COMPOSITIONALITY AND CONNECTIVITY: DIAGRAMMATIC REASONING IN PROOF ASSISTANTS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
BENJAMIN CALDWELL

CHICAGO, ILLINOIS
AUGUST 2026

ORCID iD: https://orcid.org/0009-0000-6821-8282

To the absurd journey

So it goes.

– Kurt Vonnegut, *Slaughterhouse-Five*

– Me, *Commonly*

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank my advisors, Stuart Kurtz and Robert Rand, for their continued support in chasing down a question I found fascinating. I would also like to thank my committee for their feedback during the candidacy process and their role in helping shape this work. I would like to thank my collaborators, whether close or momentary, throughout the course of this PhD, including Adrian Lehmann, Bhakti Shah, David Spitz, Laura Zielinski, Evan Cook, and William Spencer. Finally I would like to thank my partner, Nioshi, who has been with me through the whole of my PhD. Nothing we do is done alone and I am eternally grateful for those I have been fortunate enough to travel with on this journey.

ABSTRACT

Process theories capture computation as a compositional structure. Similar to circuits, they care about representing processes as blocks which can be compose in sequence or in parallel. This makes them a valuable abstraction when it comes to reasoning about programs, particularly process theories such as the ZX-Calculus have found use in reasoning about quantum programs. The primary benefit of process theories for reasoning is their corresponding diagrammatic calculi. This makes them great for pen and paper reasoning, but leaves a gap in applying this style of reasoning within formal verification. Working on pen and paper avoids a large amount of the reasoning one has to do to convince a proof assistant of correctness. We explore how to implement diagrammatic calculi in the concrete case of the ZX-Calculus and also as an abstract structure. Initially we focus on them as purely compositional structures. This ends up being insufficient to capture the reasoning power of diagrammatic calculi. We can improve our ability to capture this reasoning by working with hypergraphs alongside the categorical structures used to represent process theories. This works for concretely sized diagrams, but is insufficient for parametrically sized diagrams. To accomplish this, we work with sized hypergraphs, which act as concretely defined hypergraphs with vertex labels representing the parametric size of each vertex. With this, we are able to compute directly on the non-parametric hypergraph portion of the sized hypergraph in order to reason diagrammatically within a proof assistant. This allows us to capture full diagrammatic reasoning within a proof assistant, as we can even handle variables, which are commonly written to represent a parametric number of wires.

CHAPTER 1

INTRODUCTION

Process theories serve as an abstraction of computational theory which prioritizes the *compositionality* of the process. Rather than focusing on reasoning over specific states and how they transform, process theories instead are given as operators composed in sequence or as parallel. Many systems can be modeled as process theories, including logic circuits, the ZX-calculus (Coecke and Duncan [2011]), and even linear algebra, by viewing matrices as transforming a vector space. From a category-theoretic perspective, process theories are instances of Symmetric Monoidal Categories (SMCs). Through this interpretation, process theories gain a natural diagrammatic language of "string diagrams."

Much like a flowchart, string diagrams represent the structural relationship between components of a process, usually interpreted as the flow of information. These diagrams can then be enhanced with an equational theory specifying processes which should be considered equivalent. Often, these equations arise from equalities of the semantics of a process theory, such as the matrix formalism of quantum circuits. A string diagram enhanced with these equalities is called a *diagrammatic calculus*. Diagrammatic calculi are practical tools for reasoning about complex systems using pen and paper. Applying a rule from the equational theory is as simple as replacing a subdiagram. This replacement preserves the denotational semantics for our given diagram. For example, a local optimization on a quantum circuit can be viewed as a subdiagram replacement, where a sequence of gates is replaced with a more optimal sequence.

When reasoning with diagrams, often the thing you are looking for is the connectivity between processes. In boolean circuits, two `NOT` gates will cancel if placed in sequence. Sequencing is connecting two gates together with a line. This style of pen and paper reasoning works well and allows for easy verification given a small set of rules. Pen and paper is not a proof assistant, however. Often times extra work must be done to convince the type

checker that two terms are equivalent. This style of reasoning is not one that comes naturally to proof assistants, which are mostly built over inductive terms. Instead, the natural data structure for this type of reasoning is given by a graph, a structure which can be difficult to formalize and give semantics to. It turns out by primarily working with hypergraphs with interfaces, there is a natural correspondence between compositional terms and graph terms which can be utilized to verify equivalence of two diagrams up to a natural denotational semantic.

This thesis is divided into two primary sections, compositionality and connectivity. In the compositionality section we will cover the *Vy*ZX project, which built and verified the ZX-Calculus within the Rocq proof assistant without using graphs. This led to terms being easy to express, but was not able to capture diagrammatic reasoning. Still it shows how to build compositional generators for a diagrammatic calculus. The connectivity section covers the TensorRocq project, which served as a generalization of the work done in *Vy*ZX while also extending it to work with graph structures. This allows for terms to be written as compositonal structures and rewriting to occur in a hidden graph layer which only cares about connections.

1.1 Related works

ZX-calculus tools

Quantomatic (Kissinger and Zamdzhiev [2015]) is a diagrammatic proof assistant specific to the ZX-calculus which allows for assisted ZX-calculus rewrites. Quantomatic uses a visual interface and allows finite diagrammatic rewrites to be performed using its UI. Quantomatic isn't verified with respect to underlying linear algebraic semantics, meaning it would have difficulty interfacing with a broader verified library. It is a useful interactive theorem prover for reasoning about ZX-diagrams, but it cannot verify programs that operate on ZX-diagrams.

This is a gap that *Vy*ZX aims to fill. Quantomatic, however, allows for reasoning through adjacency and hence avoids the structural overhead introduced by *Vy*ZX, leaving it a useful tool for research on theoretical ZX-calculus results.

Proof visualization

There have been efforts in developing primarily graphical proof assistants (Jörg et al. [1999]), as well as integrating visual components into primarily textual proof assistants (Ayers et al. [2021]). One recent development in this domain is the diagram editor Yade for Rocq proofs (Lafont [2023]). Yade deploys a bidirectional framework to allow for both construction of diagrammatic proofs from proof scripts, and generation of mechanical proofs from diagrams. The diagrammatic foundations of categorical semantics make it a perfect candidate for such a bidirectional tool. Proof assistants such as Lean (de Moura et al. [2015]) have even successfully integrated interactive, user-specified graphical components into the proof engineering workflow for users to define their own interfaces as they see fit (Nawrocki et al. [2023]). This approach is far more general than the one we took with ZXViz, but may provide a roadmap to introduce such a tool to other domains.

Verified quantum computing

Several attempts have been made to verify quantum computation within a proof assistant, beginning with Boender et al's Rocq library used to prove quantum teleportation (Boender et al. [2015]). Later, Rand et al. developed a Rocq library to verify programs written in the $\mathcal{Q}$wire quantum programming language; this library later became QuantumLib (Rand et al. [2018], Paykin et al. [2017], INQWIRE Developers [2022]). QuantumLib was used and expanded in the development of the sqir intermediate representation (Hietala et al. [2021a]), the voqc verified optimizing compiler (Hietala et al. [2021b]), verified quantum oracles (Li et al. [2022]), and a proof of Shor's algorithm (Peng et al. [2023]); it also underlies the present

work.

Other attempts to verify quantum computing and quantum protocols include QBRICKS (Chareton et al. [2021]), which verified key quantum algorithms using path sums in the Why3 prover, QHLProver (Liu et al. [2019]), which verified a quantum Hoare logic in Isabelle, and CoqQ (Zhou et al. [2023]), which was able to verify cutting-edge quantum algorithms using Rocq and the Mathematical Components library. These tools showcase the active effort to verify quantum computation. Given the relevance of ZX-calculus, we believe *Vy*ZX provides the foundation to fill an important gap in verified quantum software.

String diagrams for graphical verification

Concurrently with our own work, Castello et al. proposed *causal separation diagrams* (CSDs) for reasoning about parallel processes in a proof assistant, which they formalized in Agda (Castello et al. [2023]). A key difference between our work and their work is that their diagrams enforce a strict temporal ordering of operations: "Only Connectivity Matters" is antithetical to their application to logical clocks. The lack of OCM leads to restrictions on composition, which is an interesting aspect that many graphical reasoning systems without OCM properties share. Their work is very exciting as it implicitly deals with edge labeling, creating a path to reasoning about adjacency. We take the simultaneous development of *Vy*ZX and CSDs as a sign of burgeoning interest in process theories and diagrammatic reasoning in proof assistants.

The ViCAR project aims for generality by implementing typeclasses and using monoidal coherence to assist in rewriting diagrams (Shah et al. [2025]). It is embedded within Rocq and uses a technique called *foliation* to separate terms into a normal form where patterns can be more easily identified. It provides a basic visualization engine, but is unable to ignore associativity constraints in composing morphisms, so the visualization has to directly convey associativity information, unlike in string diagrams.

Chyp is a proof assistant for symmetric monoidal categories written in Python (Kissinger [2023]). It implements a cospan of hypergraphs data structure to rewrite symmetric monoidal category terms. Chyp has a focus on interactivity, as a standalone proof assistant that can be used to prove simple facts about SMCs defined by generators (Kissinger [2023]). Users can define generators and rules in order to do syntactic proof about any SMC they can write with a finite number of generators. This makes Chyp suitable for reasoning about things like quantum circuits, where the Clifford + T gate (Nielsen and Chuang [2010]) set can serve as finite generators.

Pous' String Diagrams project has a fully interactive string diagram editor which can extract the necessary steps for a Rocq proof (Pous [2026]). This project is situated at the intersection of the above works. Like ViCAR, it attempts to be generally applicable to Rocq projects with its companion Rocq library. In order to do interactive proof, String Diagrams for Rocq uses an external tool that can import Rocq terms and export Rocq proofs as a series of tactic calls.

All these works show a clear interest in using string diagrams for reasoning across various domains, yet they have a clear gap in either their implementation being outside of a proof assistant, or if they are within a proof assistant they require additional reasoning steps to manage associativity of terms. To better understand this problem, we first look at a compositional theory built within the Rocq proof assistant. Through it, we explore how to best implement a compositional theory and how we can abstract this implementation to capture true diagrammatic reasoning.

CHAPTER 2

COMPOSITIONALITY

2.1 Introduction

This chapter is about representing a ZX-diagram in terms of its compositional structure in a proof assistant, while regaining key rules that pertain to graphs, including "only connectivity matters". Since ZX-calculus is a *calculus* endowed with a range of semantics-preserving rewrite rules, we prove the soundness of these rules as well, allowing users to graphically transform one ZX-diagram into an equivalent one.

We use these insights to present a formally verified ZX-calculus library called *Vy*ZX, formalized in a paper by Lehmann et al. [2026] which this chapter is based upon. By verified, we mean that our transformations, particularly the complete equational theory for the ZX-calculus, have been proven sound with respect to the linear-algebraic semantics of ZX-diagrams.

In pursuit of practical reasoning about ZX-diagrams, *Vy*ZX presents several key contributions:

***Vy*ZX diagrams are inductively defined and parametric** (Section 2.4). ZX-diagrams can have a variable number of inputs, connections, and outputs. Practically, ZX-calculus rules are often written parametrically, to allow application in diverse contexts. *Vy*ZX's inductive structure allows for variables for all inputs, connections, and outputs, allowing us to state and prove rules in their most general form. As our structure is inductively defined, we are able to do inductive proof over the structure of diagrams in a natural way. We can also reason over the structure of diagrams with variable placeholders for arbitrary diagrams.

***Vy*ZX allows for graphical rewriting on top of the denotational semantics** (Section 2.7). *Vy*ZX allows for inductive reasoning about graphs via denotational semantics. Semantics are given in the form of `QuantumLib` matrices, but not all proofs require the user

to appeal to those semantics. This is because *VyZX* verifies a complete set of standard ZX-calculus rewrite rules, allowing the user to prove equivalences between ZX-diagrams purely diagrammatically.

We also provide a visualizer for graphical structures. The ZXViz rocq-lsp plugin integrates graphical proof states into the user's proof writing environment. This addresses a fundamental limitation of existing graphical libraries, including *VyZX*, that graphs are hard to represent and reason about textually. In *VyZX*, we frequently see examples with a deeply nested structure that obfuscates the ZX-diagrams they represent. A human-readable graphical visualization of the proof state makes *VyZX* proofs significantly more approachable.

***VyZX* verifies a complete set of rewrite rules** (Section 2.5). We show how to prove the complete equational theory of Jeandel et al. [2020] using *VyZX*. This allows us to move from structural proof to diagrammatic proofs, as a complete equational theory on diagrams obviates the need to reason about the denotation of ZX-diagrams.

***VyZX* verifies the universality of ZX-diagrams**. Extrapolating from a proof sketch by van de Wetering [2020], we show how to construct arbitrary scalars as ZX-diagrams. We then show how to graphically take the sum of two ZX-diagrams, following Jeandel et al. [2024]. Combining these two results gives us an elegant proof that we can represent any linear map as a ZX-diagram.

***VyZX* works with other semantic models** (Section 2.6). Other works in verified quantum computing, such as SQIR (Hietala et al. [2021a]) and VOQC (Hietala et al. [2021b]), use a matrix-based Rocq quantum computing library "`QuantumLib`" (INQWIRE Developers [2022]) to define their semantics. We use `QuantumLib` to establish interoperability with SQIR circuits and show how these tools can interact through quantum circuit ingestion to *VyZX* diagrams.

Although *VyZX* was developed for the ZX-calculus, the same principles apply to any symmetric monoidal category, which can represent a broad range of mathematical structures,

all of which can be visualized as graphs (Selinger [2010]).

2.2 Inductive Graphs and Category Theory

Reasoning about a graphical language like ZX inside a proof assistant is difficult. As in most graphs, our main concern is connectivity, but graphical languages have semantics, meaning that we need a way to construct diagrams that enforces a consistent order between the graph's nodes. To find a suitable inductive definition, we turn to the category that underpins the ZX-calculus and use categorical definitions to inspire our inductive constructors. This allows us to look at something more general purpose than just ZX-calculus, and discuss how we could graphically reason about any monoidal category. This section motivates our diagrammatic work in terms of categorical concepts. However, the underlying category theory isn't necessary to understand the paper as a whole.

Monoidal categories are useful mathematical objects when creating inductive graphs, as every monoidal category admits a graphical language (Selinger [2010]) while also being defined by a small collection of equations, which we will cover later in this section. To get started, we lay out Definition 1 for categories and see how the basic objects have graphical interpretations. We will call the graphical interpretations diagrams, and each categorical definition will have an associated diagram constructor. The following definitions are adapted from Joyal and Street [1993] and Selinger [2010].

Definition 1 (Category)**.** *A* ***category*** C *is*

1. *A collection of objects of* C *for which we write* A *in* C*,*
2. *A collection of arrows between objects in* C *written* $f : A \to B$*,*
3. *A composition operator for arrows* $\circ$ *such that if* $f : A \to B$ *and* $g : B \to D$ *are arrows of* C*,* $g \circ f : A \to D$ *is an arrow of* C*,*

4. *two operations domain and codomain such that for an arrow* $f : A \to B$, *domain* $f = A$ *and codomain* $f = B$, *and*

5. *for every object* A *of* C, *an identity arrow* $\mathtt{id}_A : A \to A$,

where the following equations are satisfied:

$$\mathtt{id}_B \circ f = f, \qquad f \circ \mathtt{id}_A = f, \qquad (h \circ g) \circ f = h \circ (g \circ f)$$

Definition 2 (Functor)**.** *A functor* F *between categories* C *and* D, *written* $F : C \to D$, *is really two operations, one on the objects of* C *and one on the morphisms of* C *mapping in such a way that for objects* $A, B \in C$ *and morphisms* $f : A \to B, g : B \to E$ *satisfying* $F(f) : F(A) \to F(B)$ *and* $F(g \circ f) = F(g) \circ F(f)$ *and* $F(id_A) = id_{F(A)}$.

Definition 3 (Natural Transformation)**.** *A natural transformation between two functors* $F, G : C \to D$, *written* $\tau : F \to G$, *is given by a family of morphisms* τ_A *for each object of* C *such that for every morphism* $f : A \to B$ *the following diagram commutes:*

$$\begin{array}{ccc} F\ A & \xrightarrow{F\ f} & F\ B \\ {\scriptstyle \tau_A}\downarrow & & \downarrow{\scriptstyle \tau_B} \\ G\ A & \xrightarrow[G\ f]{} & G\ B \end{array}$$

Expanding on our basic categorical definitions, we look at the definitions for *monoidal* (Definition 4), *symmetric* (Definition 6), and *autonomous* (Definition 21) to see their diagrammatic interpretations.

Definition 4 (Monoidal Category)**.** *A category* C *is* ***monoidal*** *if there exists:*

1. *A binary functor* $\otimes$ *on objects and a left and right identity object for* $\otimes$ *called* I.

2. *A binary functor* $\otimes$ *on arrows such that if* $f : A \to B$ *and* $g : C \to D$ *then* $f \otimes g : A \otimes C \to B \otimes D$.

3. *Natural isomorphisms* $\alpha_{A,B,C} : (A \otimes B) \otimes C \simeq A \otimes (B \otimes C)$, $\lambda_A : I \otimes A \simeq A$, $\rho_A : A \otimes I \simeq A$

Where α, λ, ρ *satisfy the following:*

1. $(f \otimes (g \otimes h)) \circ \alpha_{A,B,C} = \alpha_{A',B',C'} \circ ((f \otimes g) \otimes h)$,

2. $f \circ \lambda_A = \lambda_{A'} \circ (\mathtt{id}_I \otimes f)$,

3. $f \circ \rho_A = \rho_{A'} \circ (f \otimes \mathtt{id}_I)$,

With the following axioms also holding:

1. $\otimes$ *is a bifunctor, so* $\mathtt{id}_A \otimes \mathtt{id}_B = \mathtt{id}_{A \otimes B}$ *and* $(k \otimes h) \circ (g \otimes f) = (k \circ g) \otimes (h \circ f)$,

2. *The "pentagon" and "triangle" coherence axioms given in Figure 2.1 commute.*

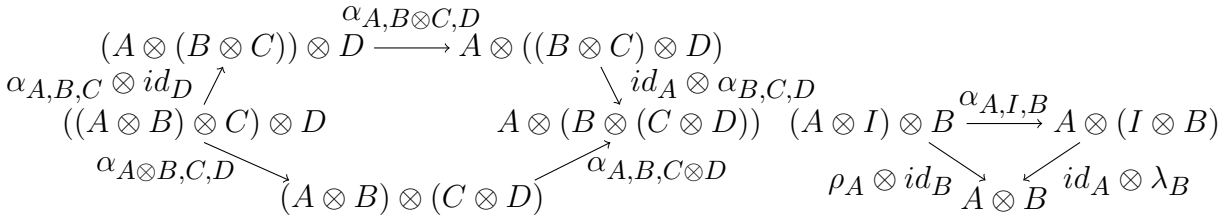


Figure 2.1: The pentagon and triangle coherence axioms.

Objects, arrows, and compositions have a natural interpretation as *string diagrams*, shown in Figure 2.2. The definition of a monoidal category adds a sense of parallelism to our category. This will allow us to take two objects or arrows and set them in parallel, or take two operations and perform them in parallel as well. The isomorphism $\alpha_{A,B,C}$ allows us to reassociate an object $(A \otimes B) \otimes C$ to $A \otimes (B \otimes C)$. We can take two arrows f and g and form a new arrow $f \otimes g$ or pass information along with $f \otimes \mathtt{id}_A$. The identity object I can

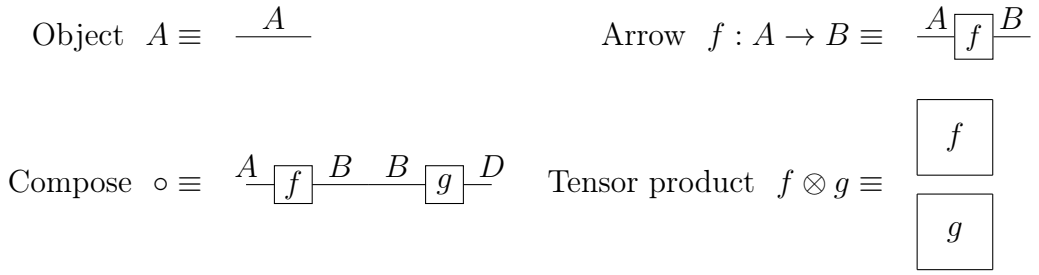


Figure 2.2: The diagrammatic construction for monoidal category definitions, where D_1 and D_2 are arbitrary diagrams.

be represented using the empty diagram, which is just blank space, allowing us to satisfy the equations λ_A and ρ_A in our diagram.

To define a symmetric category, we first define a braided category.

Definition 5 (Braided Category)**.** *A monoidal category C is **braided** if there exists a natural family of isomorphisms $c_{A,B} : A \otimes B \to B \otimes A$ such that the diagrams in Figure 2.3 commute. Intuitively we can understand the braiding isomorphisms $\beta_{A,B}$ and $\beta^{-1}_{A,B}$ as the two ways we*

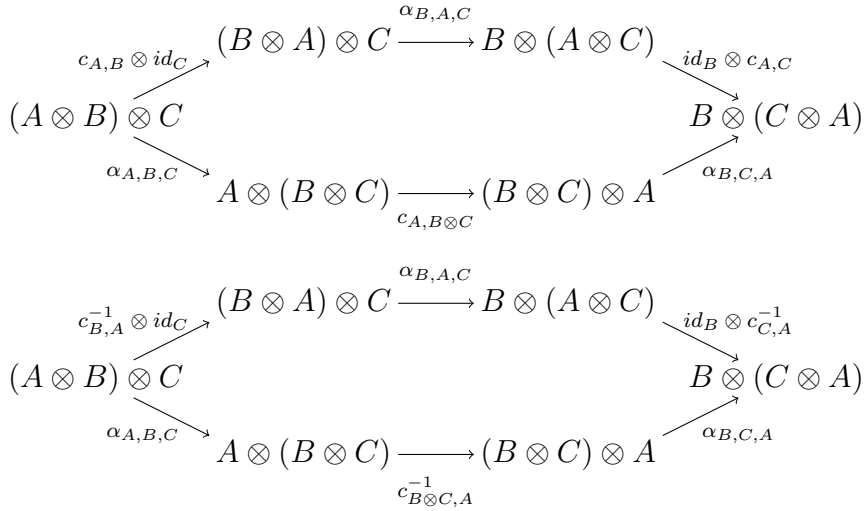


Figure 2.3: The Hexagon isomorphisms for braided categories

could lay pieces of string over one another. Either we can have string A go over string B or have string B go over string A.

Diagrammatically, this gives us two ways to wrap wires, picking one to be on top of the

other. Since we are only interested in symmetric monoidal categories for our purposes, we immediately extend the notion of a braided category.

Definition 6 (Symmetric Category)**.** *A braided category C is **symmetric** if $\beta_{A,B} \cong \beta^{-1}_{B,A}$.*

The symmetric braid can be interpreted as string diagrams as follows.

Symmetric braid $\beta_{A,B} \equiv$

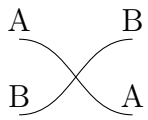


Figure 2.4: The diagrammatic interpretations for symmetric categories.

To start defining an autonomous category, we first define the dual of an object.

Definition 7 (Exact Pairing)**.** *In a monoidal category, an **exact pairing** between two objects A and B is given by a pair of morphisms $\nu : I \to B \otimes A$ and $\epsilon : A \otimes B \to I$ such that the following diagrams (written as if our category were strict, without loss of generality (Selinger [2010])), commute.*

$$\begin{array}{ccc} A & \xrightarrow{id_A \otimes \nu} & A \otimes B \otimes A \\ & \searrow^{id_A} & \downarrow{\scriptstyle \epsilon \otimes id_A} \\ & & A \end{array} \qquad \begin{array}{ccc} B & \xrightarrow{\nu \otimes id_B} & B \otimes A \otimes B \\ & \searrow^{id_B} & \downarrow{\scriptstyle id_B \otimes \epsilon} \\ & & B \end{array}$$

In an exact pairing, we say that A is the left dual of B and B the right dual of A and we call ν the unit and ϵ the co-unit. We can visualize these as follows.

Unit $\nu_A \equiv$ 

Co-unit $\epsilon_A \equiv$ 

Definition 8 (Autonomous Category)**.** *We say a monoidal category is **autonomous** if every object has both a left and a right dual (Definition 7).*

We can see the graphical definitions for a unit and co-unit in Definition 7. Given these definitions, we have a minimal collection of necessary objects to define our inductive data

structure. The simplest form of symmetric monoidal category we can define will be string diagrams with only the identity arrow, which have semantics that capture the connections from some input or output of a diagram to another input or output. This is the simplest form of a diagram we can create, and we can extend this idea later to build diagrams for the ZX-calculus. We assign these different names in Table 2.1 to match their diagrammatic interpretation more closely.

Categorical Concept	Inductive Constructor	Symbol
$\mathtt{id}_A$	Wire	–
I	Empty	$\emptyset$
$f \circ g$	Compose f g	$g \leftrightarrow f$
$\otimes$	Stack	$\updownarrow$
Symmetric braid	Swap 1 1	$\times$
Unit	Cap	$\subset$
Co-unit	Cup	$\supset$

Table 2.1: Translating categorical concepts to inductive constructors with their respective symbolic notation.

With all of our categorical concepts and constructors for diagrams laid out, we can use these exact constructors to define string diagrams. We define inductive constructors for string diagrams `SD` $(n\ m : \mathbb{N})$ with n inputs and m outputs as:

$$\frac{}{\texttt{Cup} : \texttt{SD}\ 0\ 2} \qquad \frac{}{\texttt{Cap} : \texttt{SD}\ 2\ 0} \qquad \frac{}{\texttt{Wire} : \texttt{SD}\ 1\ 1}$$

$$\frac{}{\texttt{Swap n m} : \texttt{SD}\ (\mathrm{n} + \mathrm{m})\ (\mathrm{m} + \mathrm{n})} \qquad \frac{}{\texttt{Empty} : \texttt{SD}\ 0\ 0}$$

$$\frac{\texttt{sd}_0 : \texttt{SD}\ \text{in mid} \qquad \texttt{sd}_1 : \texttt{SD}\ \text{mid out}}{\texttt{Compose}\ \texttt{sd}_0\ \texttt{sd}_1 : \texttt{SD}\ \text{in out}}$$

$$\frac{\texttt{sd}_0 : \texttt{SD}\ \text{in}_0\ \text{out}_0 \qquad \texttt{sd}_1 : \texttt{SD}\ \text{in}_1\ \text{out}_1}{\texttt{Stack}\ \texttt{sd}_0\ \texttt{sd}_1 : \texttt{SD}\ (\text{in}_0 + \text{in}_1)\ (\text{out}_0 + \text{out}_1)}$$

For visualizations of cap and cup, refer to Definition 7, for swap refer to Figure 2.4, for Wire, Compose, and Stack refer to Figure 2.2. We will refer to the diagrams generated by these constructors as the *block form* for string diagrams. They can be seen as blocks we can stack on top of one another or compose side by side if their dimensions are equal.

2.2.1 Connection Information in the Block Structure

Existing soundness and completeness results show that these graphical languages and their connection information properly reflect the categories they represent (Selinger [2010]). Completeness says that if two diagrams are equivalent up to some manipulation of the diagram (say twisting a wire around, moving a box, or stretching wires while preserving connections), then that manipulation can be explained as a consequence of the axioms of the category. In the other direction, soundness states that every axiom of a category holds in the diagrammatic language, up to some manipulation of the diagram. For example, take $\texttt{id}_A \circ f = f$. This is an axiom, but at a diagrammatic level it just looks like shortening the wire on the left side of the diagram. These two results together are known as coherence results for diagrammatic languages. All that remains for us is to take a look at the connection information of the different rules to justify why we call these diagrams "graphs". To do this, we can check the axioms given in the prior definitions Definition 1 through Definition 21 and observe they do not modify the connection information of diagrams. We take our block structure constructors from above and view them as string diagrams to do this. We then can observe that the three isomorphisms maintain connection information. From Figure 2.5 and Figure 2.6, we can see

the isomorphisms that move wires around do not modify the connection information of the diagram. The remaining isomorphisms λ_A and ρ_A do not change our diagram visually, as they add or remove the empty diagram which contains no connections. The same is true for $\alpha_{A,B,C}$, which only reassociates objects, while diagrams have no parentheses. This means the minimum collection of rules required for our string category will not modify any connection information, telling us we can use these constructors to describe graphs. We can specify our category's arrows to extend our inductive construction beyond a minimal collection of rules. When extending our string diagrams to define the ZX-calculus in Section 2.4, we have to add constructors to represent the Z and X spiders. Further, we will define an appropriate semantic interpretation for the ZX calculus.

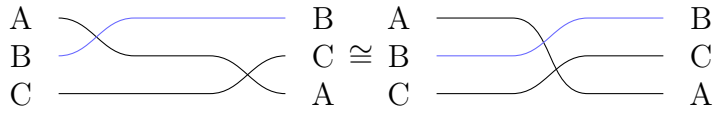


Figure 2.5: The connection information for the hexagon isomorphism; see Definition 5.

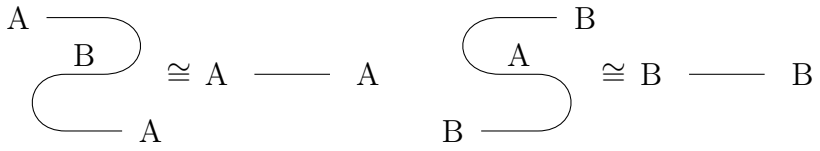
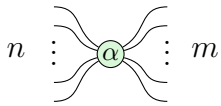
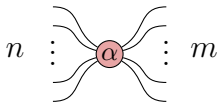


Figure 2.6: The connection information for the triangle isomorphisms; see Definition 7.



$|0\rangle^{\otimes m}\langle 0|^{\otimes n} + e^{i\alpha}|1\rangle^{\otimes m}\langle 1|^{\otimes n}$



$|+\rangle^{\otimes m}\langle +|^{\otimes n} + e^{i\alpha}|-\rangle^{\otimes m}\langle -|^{\otimes n}$

Figure 2.7: Z and X spiders with their standard bra-ket semantics.

2.3 The ZX Calculus

In most existing quantum software, like IBM's Qiskit (Qiskit contributors [2023]) or Google's Cirq (Developers [2022]), quantum programs are written as *quantum circuits* (Deutsch [1989]), a quantum analog to classical circuits that generally features from three to over a dozen distinct basic gates.[1] By contrast, ZX-diagrams are expressible with only two kinds of nodes and a more flexible graphical structure. In addition to their comparative simplicity, ZX-diagrams have proven useful for creating quantum circuit optimizers (Kissinger and van de Wetering [2020], Cowtan et al. [2020], de Beaudrap et al. [2020]), simulating quantum circuits (Kissinger and van de Wetering [2022]), implementing error-correcting codes (de Beaudrap and Horsman [2020], Chancellor et al. [2018]), and reasoning about measurement-based quantum computing (Duncan and Perdrix [2010], McElvanney and Backens [2023]).

Fundamentally, ZX-diagrams are graphs with green and red nodes, called Z and X *spiders*, with n inputs and m outputs, along with a rotation angle $\alpha \in [0, 2\pi)$. Spiders without an explicit rotation parameter are considered to have a 0 rotation. The semantics of Z and X spiders are shown in Figure 2.7. Here, $|0\rangle^{\otimes n}$ with n repeated zeros represents (in Dirac's *bra-ket* notation) a 2^n-length basis vector with a 1 in the first position and zeros elsewhere, while $\langle 0|^{\otimes n}$ represents its transpose. Similarly, $|1\rangle^{\otimes n}$ is a 2^n-length vector with a 1 in the last position. The intuition behind these spiders is that they take in a quantum state, preserve the $|0\rangle^{\otimes n}$ vectors, and rotate the $|1\rangle^{\otimes n}$ vectors by α, postselecting on the two cases above. The red X spiders act equivalently, but using the X basis. From a linear algebraic standpoint, Z and X spiders correspond to complex-valued matrices. The number of inputs and outputs to a spider determines the dimensions of the matrix it represents. Combined, the Z and X spiders are sufficient to represent any quantum computation (see Section 2.3.3).

The *ZX-calculus* (Coecke and Duncan [2011], Coecke and Kissinger [2017]) uses ZX-

1. For an accessible and thorough introduction to quantum computing, we encourage the reader to consult the standard textbook in the area (Nielsen and Chuang [2010]).

diagrams with a set of rewrite rules to translate between equivalent quantum operations. We show a sample of common rewrite rules in Section 2.3.2. Note that instead of denoting ZX-diagrams as equal, we denote them as proportional ($\propto$), meaning they are equal up to a non-zero scalar factor. Considering equality up to scalar factors is a convention in ZX-calculus, as any non-zero scalar can be written as a ZX-diagram with no inputs or outputs.

This presentation of ZX-calculus draws upon the extensive survey by van de Wetering [2020]. We also recommend Coecke's introduction to the ZX-calculus (Coecke [2023]).

2.3.1 Meta-Rules

Colorswapping We define a color-swapped ZX-diagram as a ZX-diagram with the same structure but changing every spider from Z to X and X to Z while preserving the angle. It can be shown that if a rule can be applied to a ZX-diagram zx_1 transforming it into zx_2, then it can be applied to a color-swapped version of zx_1 transforming it into a color-swapped zx_2. With this in mind, we show any rule only for one color configuration, understanding that it applies to the color-swapped version.

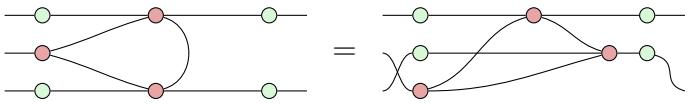


Figure 2.8: Two equivalent ZX-diagrams, where the right diagram is deformed. Connections and qubit order are maintained.

Only connectivity matters Perhaps the most important meta-rule in the ZX-calculus is that only connectivity matters (OCM). This means that wires can be arbitrarily deformed as long as the input and output order to the overall diagram is maintained. Note that this especially means that the overall diagram's matrix semantics maintain their dimensions while we allow diagram components to change their matrix dimensions. This rule is very powerful as it allows us to move diagrams into the most convenient form to apply rules to. Another

important corollary of OCM is that it doesn't matter whether a wire is an input or output to a node. We convert inputs to outputs (and vice versa) by moving the wires over the spider and adding a cup ($\subset$) or cap ($\supset$) as appropriate. From a quantum information perspective, this corresponds to exchanging inputs and outputs by adding appropriate Bell states or Bell measurements. We show an example of OCM deformations in Figure 2.8, where we can observe a Bell measurement in the left diagram becoming a simple wire between the two central red spiders. The full set of rules generating OCM are given in Figure 2.9.

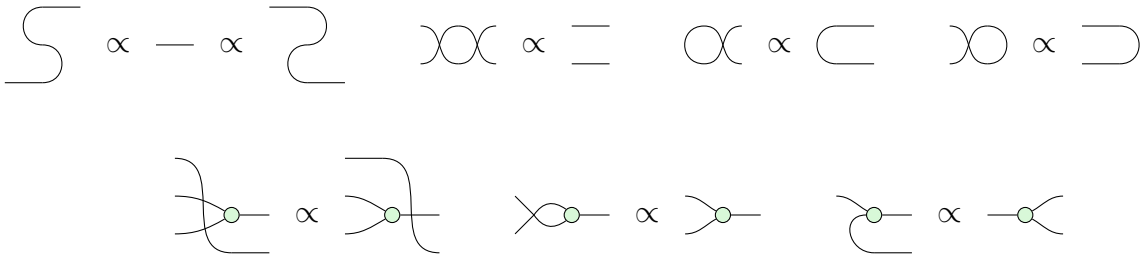
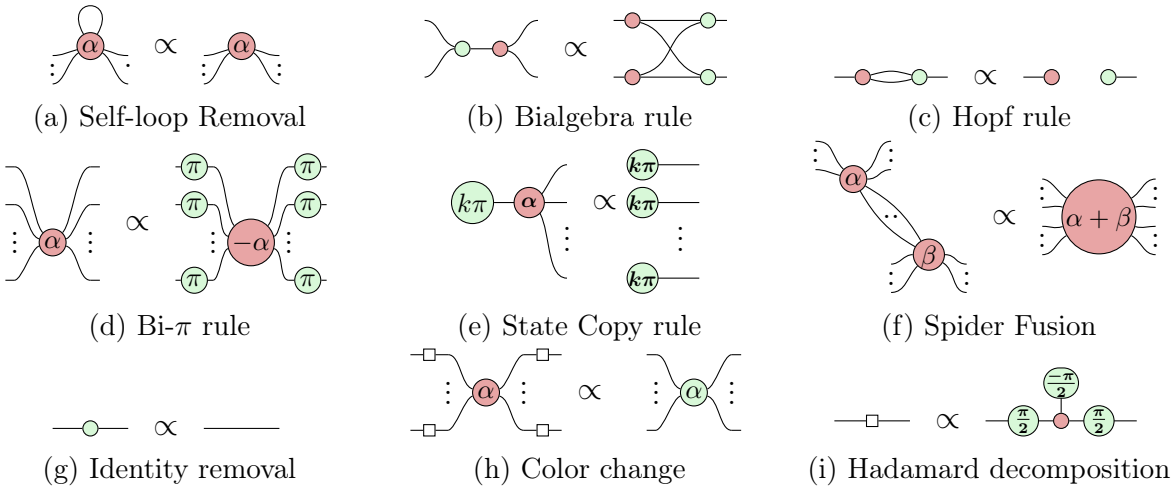


Figure 2.9: The Only Connectivity Matters rules [Jeandel et al., 2018a, Section 2.2]

2.3.2 Rewrite Rules



(a) Self-loop Removal (b) Bialgebra rule (c) Hopf rule

(d) Bi-π rule (e) State Copy rule (f) Spider Fusion

(g) Identity removal (h) Color change (i) Hadamard decomposition

Figure 2.10: Some rules of the ZX-calculus, where $\alpha, \beta \in \mathbb{R}, k \in \mathbb{N}$. Together, they form a (non-minimal) complete set of rules for the Clifford fragment of ZX-diagrams (Jeandel et al. [2020]).

The ZX-calculus comprises *rewrite rules* that can be used to transform diagrams by replacing a certain subdiagram with another. A rewrite rule is called *sound* if the matrix semantics of the diagram before the application of the rule are the same as those after the application of the rule. Whenever we refer to a rewrite rule, we implicitly mean a rewrite rule which is sound, i.e. one which preserves the interpretation of our diagrams as linear maps. Figure 2.10 lists certain rewrite rules of the ZX-calculus which are commonly used. For each of these rules, there is also a color-swapped version of the rule which changes green spiders to red spiders and red spiders to green spiders. Note that some of these, such as the Hopf rule (Figure 2.10c), are usually called derived rules to distinguish them from some set of axioms. In *Vy*ZX, all rules are proven sound directly without appeal to axioms, so we make no such distinction.

One of the most critical ZX-calculus rules is that spiders connected by an arbitrary (non-zero) number of edges can be fused into a single node with the angles added together. This rule is shown in Figure 2.10f. Applying this rule twice to Figure 2.8 would yield a diagram with a single red spider. The reverse is also true: any spider can be split such that the two new spiders add up to the original angle. A corollary of this is that spiders with 0 angle can be split off any input or output, so long as they have the color of the original spider. Further, with the spider fusion rule, we can manipulate the number of inputs and outputs for all other rules by adding a fusible node to said input or output.

Another simplification is that we can remove self-loops, as shown in Figure 2.10a (van de Wetering [2020]). Intuitively, this rule says that a node cannot provide more information about itself. The remaining rules are fairly self-explanatory, for an intuitive description of their behavior, see van de Wetering [2020].

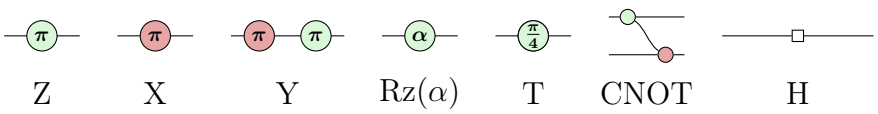


Figure 2.11: Common gates represented in the ZX-calculus.

2.3.3 Representing Unitary Gates

We translate some standard unitary gates from the quantum circuit model into ZX-calculus in Figure 2.11. Z, T, and Rz(α) represent Z-axis rotations by the given angles, while X and Y rotate qubits by π around the X and Y axes. The H (or Hadamard) gate switches between the Z and X bases; following van de Wetering [2020] it can be decomposed into a sequence of red and green nodes. Finally, CNOT applies an X rotation (or NOT) on the second qubit contingent on the first being $|1\rangle$. This set of unitary gates corresponds to the RzQ gate set (Nam et al. [2018]), a universal gate set for quantum circuits (i.e. one capable of implementing any quantum computation).

Having translations from gates into ZX allows us to convert quantum circuits to diagrams. Once we translate circuits into diagrams, we can use the ZX-calculus rules without being bound by the rigid structure of the circuit model. While several equational theories have recently been proposed for quantum circuits, these are generally more complicated than those of the ZX-calculus (Clément et al. [2023, 2024]).

Due to its broad applicability to quantum computing, including circuit optimizations (Kissinger and van de Wetering [2020], Cowtan et al. [2020]) and error correction (de Beaudrap and Horsman [2020], Chancellor et al. [2018]), as well as its array of extensions (such as the ZH (Backens and Kissinger [2019]) and ZW (Hadzihasanovic [2017] calculi)), the ZX-calculus is a prime target for formal verification. However, achieving this in practice requires representing these graphical structures inductively, which will prove challenging.

2.4 Inductive ZX-diagrams

Given the ZX-calculus is a symmetric monoidal category (the interested reader may find more information about the category theoretical foundations in Section 2.2) generated by the Z and X *spiders*, we can naturally translate the ZX-calculus into an inductive datatype for Rocq, as shown in Figure 2.12. Note that for convenience we add a box (□), which corresponds to

the conventional Hadamard box in ZX-calculus, which is technically not a core part of the ZX-calculus. We define additional constructs for convenience. Relevant constructs include `n_stack1` : $\forall$ `n` : `nat`, `ZX 1 1` $\rightarrow$ `ZX n n`, written `n` $\uparrow$ `zx`, which stacks a ZX diagram with a single input and output n times upwards. We also create a shorthand for n stacked wires called `n_wire n`.

To assign meaning to our syntactic constructs, we construct a semantic evaluation function $[\![.]\!]$: `ZX n m` $\rightarrow$ `Matrix` $2^m \times 2^n$; a ZX-diagram with n inputs and m outputs semantically evaluates to a complex-valued matrix of size 2^m by 2^n. In practice, all our semantics are built using QuantumLib's (INQWIRE Developers [2022]) matrices and complex numbers.

We implement $[\![.]\!]$ as shown in Figure 2.13, with the minor difference that instead of generating the matrices for Z spiders through the composition of vectors, we directly build the resulting matrix. The semantic definition of the X spider is also constructed using the equation seen in Figure 2.10h. Some constructs such as the wire ($-$), Hadamard box ($\square$), cup ($\subset$), and cap ($\supset$), could be equivalently created out of various compositions of spiders; however, given that they will appear frequently in definitions and lemmas, it is useful to have them as their own constructors.

We define the equivalence relation *proportionality* to state that two diagrams are equal up to a scalar factor:

$$\forall(\mathtt{zx_0}, \mathtt{zx_1} : \mathtt{ZX\ n\ m}), \mathtt{zx_0} \propto \mathtt{zx_1} \coloneqq \exists c \in \mathbb{C}, [\![\mathtt{zx_0}]\!] = c \cdot [\![\mathtt{zx_1}]\!] \wedge c \neq 0$$

$$\frac{\mathtt{in\ out} : \mathbb{N} \qquad \alpha : \mathbb{R}}{\mathtt{Z\ in\ out}\ \alpha : \mathtt{ZX\ in\ out}} \qquad \frac{}{\supset : \mathtt{ZX\ 0\ 2}} \qquad \frac{}{\subset : \mathtt{ZX\ 2\ 0}} \qquad \frac{\mathtt{in\ out} : \mathbb{N} \qquad \alpha : \mathbb{R}}{\mathtt{X\ in\ out}\ \alpha : \mathtt{ZX\ in\ out}}$$

$$\frac{}{- : \mathtt{ZX\ 1\ 1}} \qquad \frac{}{\square : \mathtt{ZX\ 1\ 1}} \qquad \frac{}{\times : \mathtt{ZX\ 2\ 2}} \qquad \frac{}{\emptyset : \mathtt{ZX\ 0\ 0}}$$

$$\frac{\mathtt{zx_0} : \mathtt{ZX\ in\ mid} \qquad \mathtt{zx_1} : \mathtt{ZX\ mid\ out}}{\mathtt{zx_0} \leftrightarrow \mathtt{zx_1} : \mathtt{ZX\ in\ out}} \qquad \frac{\mathtt{zx_0} : \mathtt{ZX}\ \mathrm{in}_0\ \mathrm{out}_0 \qquad \mathtt{zx_1} : \mathtt{ZX}\ \mathrm{in}_1\ \mathrm{out}_1}{\mathtt{zx_0} \updownarrow \mathtt{zx_1} : \mathtt{ZX}\ (\mathrm{in}_0 + \mathrm{in}_1)\ (\mathrm{out}_0 + \mathrm{out}_1)}$$

Figure 2.12: The inductive constructors for block representation ZX-diagrams

We show within Rocq that this is an equivalence relation and that `Stack` ($\updownarrow$) and `Compose` ($\leftrightarrow$) are *parametric morphisms* (Sozeau [2023]), meaning that we can safely rewrite using proportionality within ZX-diagrams. Using the definition of proportionality, we can then prove facts about the ZX-calculus.

This proportionality judgment is sufficient for most of our use-cases, such as for optimizing unitary circuits (Kissinger and van de Wetering [2020], see also Section 2.6). For some applications, however, such as reasoning about probabilistic quantum circuits, it is important to track the values of the associated constants. To facilitate those applications, we define refinements of implicit proportionality ($\propto$) with explicit proportionality constants, and prove each lemma with the strictest possible relation. We define explicit proportionality ($\propto$`[c]`) denoting proportionality by c, which must be nonzero:

$$\forall(\mathtt{zx_0}, \mathtt{zx_1} : \mathtt{ZX\ n\ m})\ (\mathtt{c} : C), \mathtt{zx_0} \propto \mathtt{[c]}\ \ \mathtt{zx_1} \coloneqq [\![\mathtt{zx_0}]\!] = c \cdot [\![\mathtt{zx_1}]\!] \wedge c \neq 0$$

Using this relation, the explicit constant is captured. We also define a stricter version $\propto$`=`, which captures semantic equality, i.e., $\propto$`[1]`:

$$\forall(\mathtt{zx_0}, \mathtt{zx_1} : \mathtt{ZX\ n\ m}), \mathtt{zx_0} \propto\!= \mathtt{zx_1} \coloneqq [\![\mathtt{zx_0}]\!] = [\![\mathtt{zx_1}]\!]$$

In practice, most results hold up to strict semantic equality. Moreover, we can add a scalar gadget to diagrams to convert statements of $\propto$`[c]` to statements of $\propto$`=` following

$$\begin{aligned}
[\![\mathtt{0}]\!] = I_{1\times 1} \qquad [\![-]\!] = I_{2\times 2} \qquad [\![\sqcap]\!] &= H \qquad [\![\subset]\!] = [1,0,0,1]^T \\
[\![\times]\!] = |00\rangle\langle 00| + |11\rangle\langle 11| + |01\rangle\langle 10| + |10\rangle\langle 01| \qquad [\![\supset]\!] &= [1,0,0,1] \\
[\![\mathtt{Z\ n\ m}\ \alpha]\!] = |0\rangle^{\otimes m}\langle 0|^{\otimes n} + e^{i\alpha}|1\rangle^{\otimes m}\langle 1|^{\otimes n} \qquad [\![\mathtt{X\ n\ m}\ \alpha]\!] &= H^{\otimes m} \times [\![\mathtt{Z\ n\ m}\ \alpha]\!] \times H^{\otimes n} \\
[\![\mathtt{zx_0} \leftrightarrow \mathtt{zx_1}]\!] = [\![\mathtt{zx_1}]\!] \times [\![\mathtt{zx_0}]\!] \qquad [\![\mathtt{zx_0} \updownarrow \mathtt{zx_1}]\!] &= [\![\mathtt{zx_0}]\!] \otimes [\![\mathtt{zx_1}]\!]
\end{aligned}$$

Figure 2.13: *Vy*ZX semantics

the technique in van de Wetering ([van de Wetering, 2020, Section 3.4]). Concretely, the notation `to_gadget` uses tactics to convert a lemma ending in a statement $\texttt{zx}_0 \propto \texttt{[c]}\ \texttt{zx}_1$ to a lemma ending $\texttt{zx}_0 \propto= \texttt{zx\_of\_const c} \updownarrow \texttt{zx}_1$, where `zx_of_const c` is a gadget with semantics `[c] : Matrix 1 1`, a 1 by 1 matrix whose only element is c. This allows us to reason with only $\propto$= when we want to prove strict equality of ZX-diagrams up to their semantic interpretations. By using the subrelation feature of Rocq's setoid rewriting, the stricter proportionality relations $\propto$= and $\propto$[c] can be directly rewritten in weaker contexts, allowing us to consistently state lemmas in their strictest forms without loss of functionality. For simplicity, we state lemmas in this paper using $\propto$, while in the development they are stated in their stricter forms, either $\propto$= or $\propto$[c].

In addition to reasoning about proportionality, we often have to reason about the composition of ZX-diagrams. Here, a common challenge in dependently typed programming shows up, as we require precise equality of dimensions across proportionality and the composition constructor. While the proof of the associativity of `Compose` is trivial, we encounter an issue with `Stack`. Consider the diagrams $\texttt{zx}_0 : \texttt{ZX n m}, \texttt{zx}_1 : \texttt{ZX n' m'}, \texttt{zx}_2 : \texttt{ZX n'' m''}$. Here the diagrams $(\texttt{zx}_0 \updownarrow \texttt{zx}_1) \updownarrow \texttt{zx}_2$ and $\texttt{zx}_0 \updownarrow (\texttt{zx}_1 \updownarrow \texttt{zx}_2)$ have the incompatible types `ZX (n+n')+n'' (m+m')+m''` and `ZX n+(n'+n'') m+(m'+m'')`, respectively, as we see from the typing rules in Figure 2.12. To bridge this gap, we define a `cast` function with the following type:

```
cast (n m:nat) {n' m':nat} (pfn: n=n') (pfm: m=m') (zx: ZX n' m') : ZX n m.
```

`cast` uses a proof that $n = n'$ and $m = m'$ to change the explicit dimensions of a ZX-diagram. Casting functions have been used in similar contexts, like the Lean matrix library mathlib3's `matrix.reindex` (mathlib Community [2020]) or Rocq vector library's `cast` (Rocq Development Team [2025]). Note that Rocq can infer n' and m' of `cast`. Casts introduce more implicit structure into our already highly structured diagrams. For example, in proving associativity, we can decide which side of the proportionality judgment is cast to the other's dimensions. In the implementation, we define numerous lemmas to move casts through our

structure under type restrictions appropriately. In practice, it turns out that the added structure and required moves are the only proof overhead caused. Potential obligations from creating/manipulating casts can commonly be resolved using a linear arithmetic solver such as Rocq's `lia` (Besson [2007]).

With proof irrelevance for natural number equality (see Hedberg's theorem (Hedberg [1998])), we then in theory can rewrite `cast`s without looking at their specific proof terms, as proofs of natural number equality are interchangeable. Unfortunately, in their current states, Rocq's `rewrite` and `setoid_rewrite` tactics cannot perform such rewrites, even if the proofs are provably irrelevant. However, *Vy*ZX handles most straightforward cases of cast automatically, as we discuss in Section 2.8.2.

We show the Rocq code for stack associativity as an example illustrating the use of `cast` in Listing 2.1. Here, `cast` serves to unify the dimensions of the proportionality statement. Notice that while we could provide an exact proof to cast (`Nat.add_assoc`), we instead keep the proof parameterized. This circumvents the limitations of the Rocq rewrite engine with respect to proof irrelevance by proving the lemma for an arbitrary proof, allowing it to be rewritten both forward and backward in any context. If used in forward rewrites, having parameterized proofs creates additional obligations, which can either be immediately resolved with automation or deferred to the end, where all remaining obligations generated by cast operations can be resolved using a linear arithmetic solver. Since our lemmas are parametric over equality proofs, we can backwards rewrite with any arbitrary proof.

For key results, it is prudent to restate the theorem with explicit rather than parametric proof lemmas, exhibiting a witness for the `cast` equalities. These explicit lemmas can be trivially solved using automation that uses the parametric proof with an arithmetic solver. For example, let us look at Listing 2.1. We can define an explicit version shown in Listing 2.2, which also demonstrates the tactic to solve these kinds of problems. Note that the statements in Listings 2.1 and 2.2 are almost identical, except that the latter states the existence of

proofs for `cast`. Therefore, our approach allows us to provide a lightweight, Rocq rewrite engine compatible, encapsulation of proof irrelevance at the cost of automatically resolvable goals.

2.5 Verifying a Complete Equational Theory

To encode the ZX-calculus, we need both diagrams and rules. Section 2.4 described how we can encode any ZX-diagram in Rocq. Here, we show how to encode the canonical set of rules from Section 2.3 using the visual representation generated by ZXViz.

Only connectivity matters

The idea behind "only connectivity matters" is that we can arbitrarily deform a diagram without changing its semantics. The ability to deform diagrams is nicely generated by a small set of axioms ([Jeandel et al., 2018a, Section 2.2]). Of the "only connectivity matters" rules given by Jeandel et al., one significant rule allows the free conversion between inputs and outputs. We provide wrapping lemmas like the one shown in Figure 2.15a (as a string diagram in Figure 2.9) to make this possible within our block structure. Likewise, the yanking equations (see Figure 2.15b and Figure 2.9) allow us to remove redundant cups and caps from our diagrams. By proving the lemmas given by Jeandel et al., we encapsulate "only connectivity matters" directly rather than assuming it by using a datatype (such as an adjacency matrix) that only explicitly encodes connection information.

```
Lemma stack_assoc : ∀ {n0 n1 n2 m0 m1 m2 : nat}
(zx0 : ZX n0 m0) (zx1 : ZX n1 m1) (zx2 : ZX n2 m2) prfn prfm,
(zx0 ↕ zx1) ↕ zx2 ∝ cast _ _ prfn prfm (zx0 ↕ (zx1 ↕ zx2)).
```

Listing 2.1: Example of the use of `cast`: Proving the associativity of stack. Found in `src/CoreRules/StackRules.v`

Spider fusion

Spider fusion is perhaps the most interesting and important rule in the ZX-calculus. We implement it in *Vy*ZX by explicitly encoding the number of inputs and outputs of each spider and the connections between the two spiders. Notice that the dimensions in Figure 2.14a across the cast are `top` + (1 + `mid`) + `bot` and `top` + (1 + `mid` + `bot`) respectively. To unify this discrepancy, we must cast (see Section 2.4) one side of the composition. This rule also highlights that we can express *Vy*ZX rules in multiple ways. For example, we assume that the left node in the fusion expression is above the wires and the right node is below the wires. There are, however, many combinations of different locations for the rules. Further, we could cast either the left or right side.

Self-loop removal

Figure 2.14b shows self-loop removal in *Vy*ZX. We express self-loop removal with the restriction that the self-loop has come from a given spider's top inputs/outputs. However, using swaps constructors, we can swap any output of a spider to become the uppermost. We can then remove swaps fully connected to a spider using other lemmas.

```
Global Tactic Notation "crush_explicit_cast" constr(lemma) :=
    unshelve (do 2 eexists; apply lemma); lia.

Lemma stack_assoc_non_param :  ∀ {n0 n1 n2 m0 m1 m2}
(zx0 : ZX n0 m0) (zx1 : ZX n1 m1) (zx2 : ZX n2 m2), ∃ prfn prfm,
(zx0 ↕ zx1) ↕ zx2 ∝ cast _ _ prfn prfm (zx0 ↕ (zx1 ↕ zx2)).
Proof.
    crush_explicit_cast stack_assoc.
Qed.
```

Listing 2.2: Example of the explicit `cast`: Proving the associativity of stack. Not found in repository.

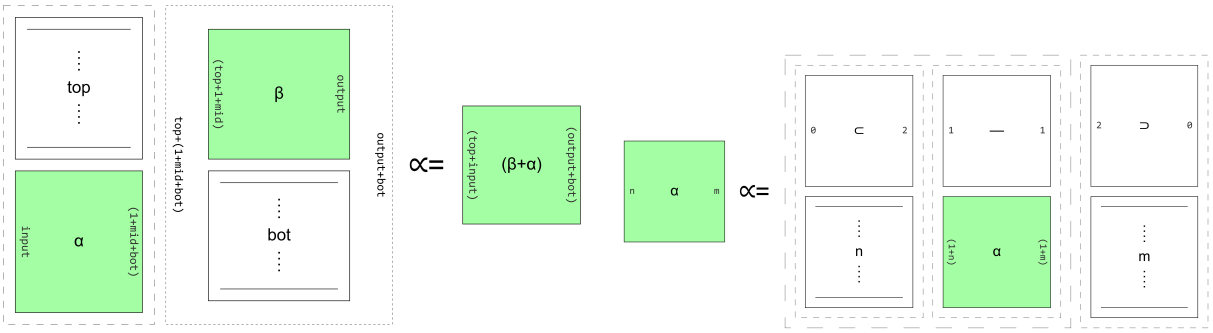
Other rules

The *Hopf rule*, *Bi-algebra rule* and *state-copy-rule* can be directly stated in *Vy*ZX cast-free, as shown in Figures A.1d and A.2a

Transpose & Adjoint

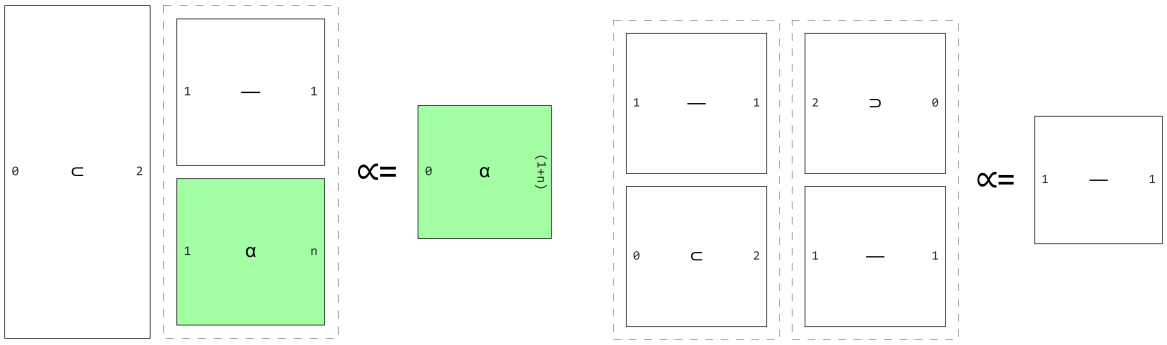
As we can see, *Vy*ZX rules are often restricted because they require certain relative positioning. However, using *Vy*ZX, we can rearrange the structure if necessary or prove lemmas with different positioning of components (e.g., for Figure 2.14a we also have a version where we have the left green spider on the bottom and the right green spider on the top). We can automate these changes using the transpose, adjoint, and color swap operations.

All told, VyZX implements Jeandel et al. [2018b] complete set of rewrite rules for the ZX-calculus. This allows us to prove two diagrams equivalent without having to rely on the matrix semantics or any matrix-level manipulations. The full set of rewrite rules is given in ZXViz form in Chapter A.



(a) Spider fusion in *Vy*ZX (b) Self-loop removal in *Vy*ZX

Figure 2.14: Select rules from *Vy*ZX



(a) The wrap over top left lemma used to express only connectivity matters

(b) A yanking lemma used to simplify connectivity information.

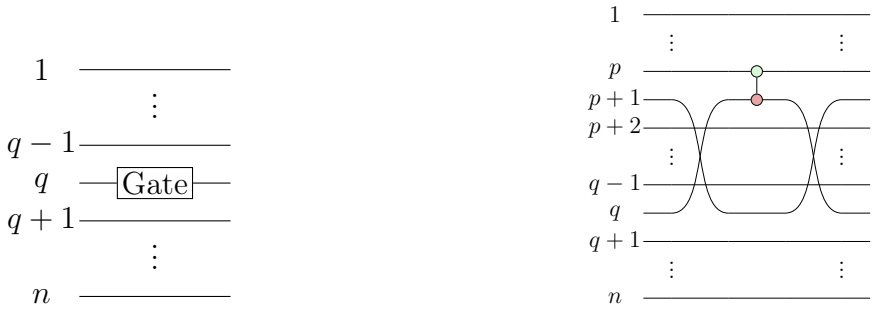
Figure 2.15: Visualizing two "only connectivity matters" lemmas in *VyZX*

2.6 Reasoning about Circuits with *VyZX*

Ingesting Quantum Circuits

Quantum programs are usually represented using the circuit model (Deutsch [1989], Nielsen and Chuang [2010]). SQIR (Hietala et al. [2021a]) embeds quantum circuits in Rocq, with `QuantumLib` matrix semantics (INQWIRE Developers [2022]). Fundamentally, its representation of circuits is an inductive structure that allows circuit components with n qubits (i.e., inputs and outputs) to be created. Quantum circuits in general work on a qubit-based model, circuits and all sub-circuits that are horizontally composed are constant size and induce square matrices. Circuits comprise a composition of components of size n. Each component either has no gate, a one-qubit gate operating on qubit $q < n$, or a two-qubit gate operating on qubits $p, q < n$ where $p \neq q$. We immediately see that this model differs substantially from *VyZX*'s model: diagrams have n inputs and m outputs for any n or m. Moreover, SQIR circuit components explicitly specify gate locations instead of stacking diagrams.

Ingesting circuits is the process of converting SQIR circuits into *VyZX* diagrams. In the following, we describe our Rocq formalization of SQIR circuit ingestion in *VyZX*. Performing this transformation for a circuit with n qubits, we build components of type `ZX n n` for each



(a) Ingestion of a one qubit gate acting on qubit q into a ZX-diagram

(b) Constructing CNOT gate on qubits p and q (where $p < q$) for a circuit with n qubits)

Figure 2.16: Construction of SQIR operations in *VyZX*

gate application and then compose them appropriately.

WLOG, we can assume that 1-qubit gates are Hadamard, Pauli X, or Rz gates with rotation α, and 2-qubit gates are CNOT gates between two qubits[2]. For the case of 1-qubit gates, we define constructions `ZX_H` : `ZX` 1 1, `ZX_X` : `ZX` 1 1, and `ZX_Rz` : $\mathbb{R} \to$ `ZX` 1 1 that translate gates to the corresponding ZX-diagrams as shown in Section 2.3.3. To include explicit circuit size, we define padding functions, to add an arbitrary number of wires to the bottom or top of a diagram. We then use these functions to define a function that produces diagrams of the specified height. For a gate operating on qubit q, we pad our `ZX` 1 1 circuit with $q - 1$ wires above and $n - q$ wires below. Figure 2.16a shows the result of said transformation. To translate CNOT gates, we need to be more thoughtful. Since we can't easily connect two arbitrary qubits in *VyZX*, we first consider CNOTs on adjacent qubits. We see that in such situations, we can use the approach for single-qubit gates to correctly place the CNOT using a slight modification of the construction shown in Figure 2.16a. To allow for CNOT gates that do not operate on adjacent qubits, we swap the qubit with the higher index to be next to the qubit with the lower index. Notice, this is not trivially accomplished with the basic swap operations, as they merely swap adjacent qubits. Unlike the adjacent

2. We can make this assumption as VOQC can verifiably translate any SQIR circuit into the gate set that only contains these terms. This gate set is commonly referred to as the RzQ gate set (Hietala et al. [2021b], Nam et al. [2018]).

swaps, these swaps are not basic constructs of our representation; we need to define them. An arbitrary swap swaps the first and n^{th} qubits, requires chaining of $2n-1$ basic swaps[3]. Using arbitrary swaps and shifts, we can now interpret any wire crossing. Hence, in block representation, we can construct a CNOT acting on two qubits p, q by swapping one qubit next to the other qubit, applying the CNOT, and swapping back, as shown in Figure 2.16b. We can also construct single qubit gates as previously described. Composition of SQIR terms is represented by composition in our block representation. We use proof automation to convert any proposition about circuit equality into an equivalent proposition about ZX-diagrams.

Showing this translation highlights that our syntactic representation of graphical diagrams can interoperate with different notions of semantics. This is a common challenge, especially in systems relying on axiomatic semantics, since two systems will likely not share the same ground truth. By using `QuantumLib`, *Vy*ZX can be used for proof along with other libraries using `QuantumLib`, bridging the common pitfall of having different verification models that only exist in isolation.

Peephole optimizations

Using circuit ingestion, we demonstrate the capabilities of *Vy*ZX by verifying peephole optimizations developed by Nam et al. [2018] which are previously verified in the VOQC optimizer (Hietala et al. [2021b]).

We use automation to convert each circuit into a ZX-diagram and then proceed to prove equivalences using *Vy*ZX's rules. In *Vy*ZX, we are often able to clearly see the equivalences based on the few ZX rules that exist and have an immediate proof path we can follow. Due to *Vy*ZX's approach, we are able to use the fact that some of these rules are corollaries of each other, as they are equivalent save for the fact that they are transposes, complex conjugates, or color swapped versions of each other. This is something that is easy to

3. The construction works by moving the qubit 1 down to n and then moving qubit $n-1$ up to qubit 1.

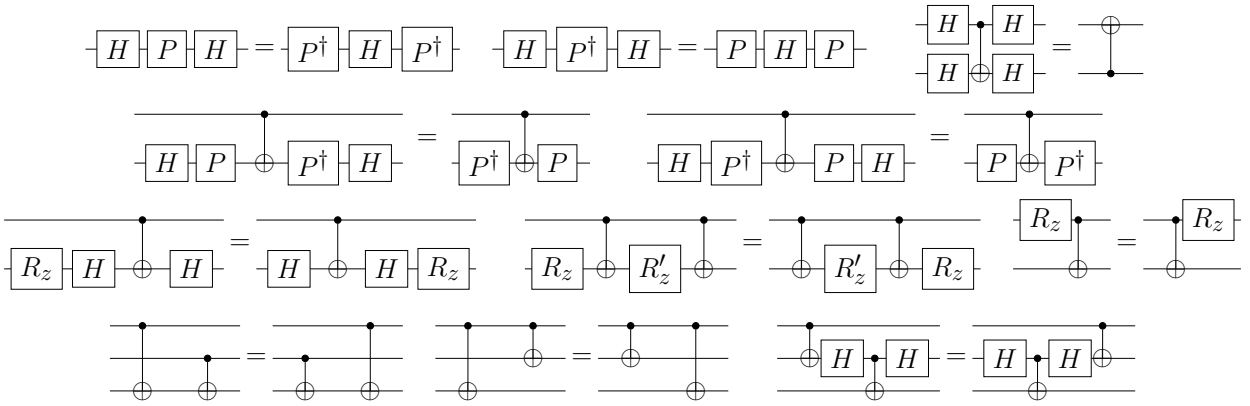


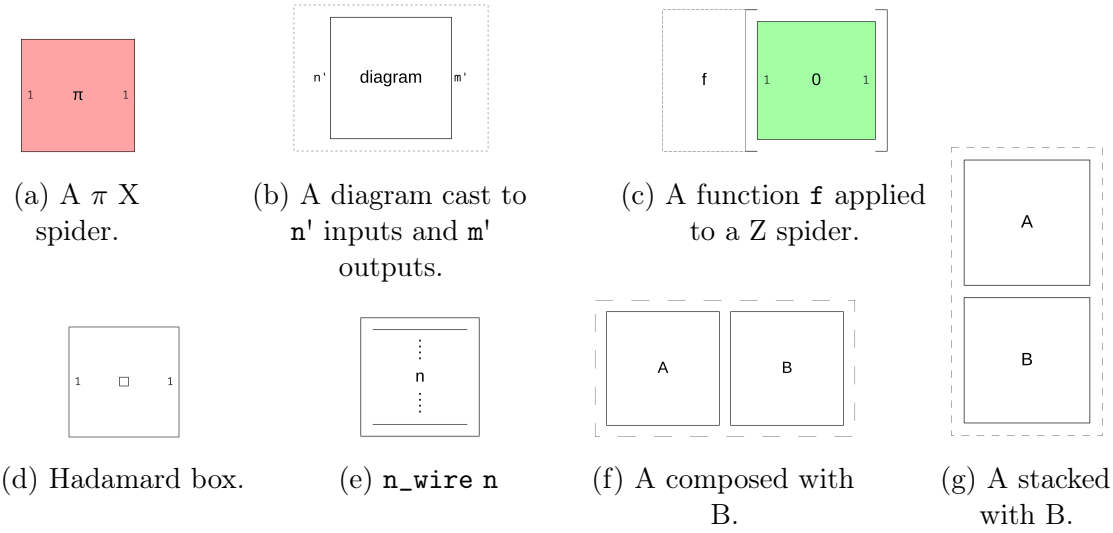
Figure 2.17: Quantum circuit peephole optimizations from figures 4 and 5 in Nam et al. [2018]

see diagrammatically (using ZXViz), but less obvious in circuit representation or textual representation of ZX-diagrams.

By implementing these optimizations, we show the flexibility *Vy*ZX for reasoning about quantum systems and set the foundations for building a verified optimizer in *Vy*ZX.

2.7 ZXViz

With all the structure we introduce for ZX-diagrams in *Vy*ZX, our textual, inductive representation of ZX-diagrams can be deeply nested and tricky to parse. To address this, we exploit the inherently visual nature of our diagrams to provide a human-readable diagrammatic representation using ZXViz. The block structure is a key feature that distinguishes ZXViz diagrams from standard visualizations of ZX-diagrams. ZX-diagrams are built around the principle of connectivity, which is useful for pen and paper reasoning but obfuscates details about associativity at the semantic level. *Vy*ZX proofs manipulate the associativity and subdiagrams of the block form rather than just the connectivity of the graphical form. Furthermore, parametric properties such as the number of inputs and outputs for casts must be kept intact in the visualizations, something the standard representation does not make room for.



(a) A π X spider.

(b) A diagram cast to `n'` inputs and `m'` outputs.

(c) A function `f` applied to a Z spider.

(d) Hadamard box.

(e) `n_wire n`

(f) A composed with B.

(g) A stacked with B.

Figure 2.18: ZXViz visualization of *Vy*ZX constructors and functions, as in Section 2.4.

Figure 2.18 shows how we visualize constructs from Section 2.5. To avoid ambiguity related to associativity information, we also enclose the stacked, composed, or cast diagrams within a dashed boundary. The styles for the boundaries are slightly varied by construct to improve readability when multiple dashed lines are side-by-side.

*Vy*ZX diagrams are frequently more complex than just the single elements explored above. In **??**, we show a graphical proof that uses ZX-calculus rewriting and ZXViz to show that the standard circuit for constructing a Bell pair is equivalent to a cup.

Visualization makes it much simpler to interpret the structure of terms. The visualization clarifies how the subterms fit into the larger associativity structure. We discuss how this aids proof engineering practice in Section 2.8.3. As terms get more complex, the visualizations significantly enhance the user's experience with parsing the proof goal. Using color for the `Z` and `X` spiders gives us another layer of visual communication that text cannot provide. Associativity can be challenging to understand from the term string, but the visual element makes it easy to parse. Additionally, it is often hard to tell whether subterms are adjacent (up to associativity) in a term string, particularly when they appear in nested horizontal and vertical compositions. In contrast, the visual representation clearly presents the positional

relationship between subterms, while also helping guide the user in reassociating the goal to directly expose adjacent terms.

First, the input *Vy*ZX proof state is lexed and parsed. We explicitly keep all notations and syntactic sugar during this process to stay faithful to the user's textual goal state. Next, the parsed term and all nested sub-terms are assigned graphical objects, which ZXViz places on an HTML canvas according to a user-provided scale. The rendering happens in a two-pass system. First, each element is assigned a base position and size based on stack and composition information. In the event of a mismatch in the sizes of composed elements, the second pass takes note of this discrepancy and increases the size of the smaller element. An example of this can be seen in Figure 2.19b. In this example, we initially determine the native size of all elements. All elements except the rightmost spider also keep this size through the second pass, as they are not stacked or composed with any element that has a larger minimum size. The rightmost spider, however, is composed with the stack of two base-size elements, so the second pass increases its vertical size to make it visually coherent. We handle text by appropriately scaling and wrapping it when necessary. The term is then rendered using the determined layout, with additional visual components including the formatted text and color. Finally, this canvas is displayed as a webview in the VSCode user interface and rendered as a panel alongside the code and goals.

2.8 Graphical proof in *Vy*ZX

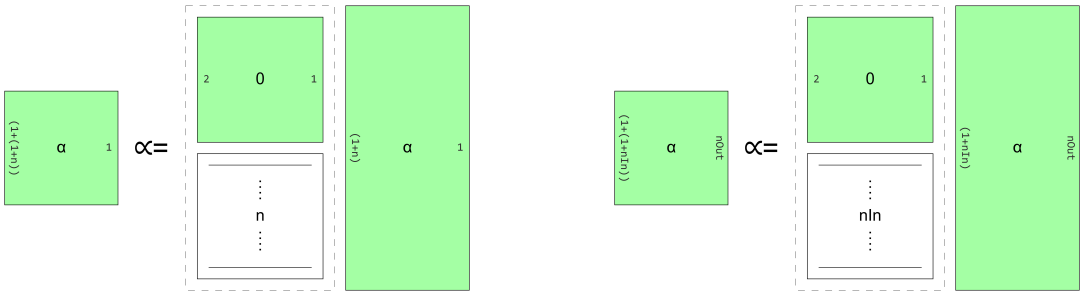
Every formal verification project must eventually confront proof engineering challenges. In *Vy*ZX, it is important to concisely express proofs about the ZX-calculus. To this end, we prove basic facts about the graphical language through the underlying linear algebraic semantics. These basic facts allow us to derive additional lemmas without appealing to linear algebra. Proving facts using only graphical reasoning allows for more understandable proofs. This approach is integral to the school of *Quantum Picturalism*(Coecke [2010], Coecke and

Kissinger [2017]), the philosophy that underlies the ZX-calculus and quantum process calculi in general.

The constructors within *Vy*ZX represent graphical objects textually. This textual representation can be unintuitive. We must consider how to make it approachable. Using ZXViz, we visualize the proof goal's structure to comprehend the proof state and effectively identify rewrite strategies.

2.8.1 Induction

A central proof tactic we want to bring to graphical calculi is induction, allowing us to prove statements for parametric numbers of inputs or outputs of a given node. To accomplish this for the most basic case, a single Z spider, we must reduce the number of input or outputs from that spider by 1. This technique relies on our splitting lemmas, which are themselves proved inductively. We discuss how we built induction capabilities throughout *Vy*ZX to apply inductive reasoning in practice.



(a) `Lemma grow_Z_left_2_1` (b) `Lemma grow_Z_top_left`

Figure 2.19: `grow_Z_top_left` is built by induction using the base lemma `grow_Z_left_2_1` which is proven directly through matrix semantics.

Induction on the size of a spider relies on splitting out a two-connection spider. If we only split a single connection off, we would not reduce the size of the original spider. By splitting two off, we reduce the size of the output, making it something we can apply our inductive hypothesis to. This splitting is shown by the lemma in Figure 2.19a. We prove this lemma

directly via the underlying semantics by using mathematical properties of composition and stacking. Using this lemma, we then prove the lemma shown in Figure 2.19b. This lemma allows us to reason inductively over spider sizes and hence allows us to prove parametric rules, such as the wrapping rules shown in Section 2.5. In **??** we show an example of a practical application of inductive proof in *Vy*ZX.

2.8.2 Proof automation

Previously, we primarily talked about lemmas that act on Z spiders; however, from the ZX-calculus, we know that every valid statement has a valid dual where all spider colors are swapped. We encapsulate this through a function that creates the dual diagrams, denoted by $\odot$, and we prove that `zx0` $\propto$ `zx1` implies $\odot$ `zx0` $\propto$ $\odot$ `zx1`. Similarly, we create a function to transpose diagrams and prove a corresponding lemma.

Using Rocq's proof automation features, specifically `autorewrite` and `Ltac`, we provide tactics that automatically prove the color-swapped or transposed versions of a lemma. For example, we prove the dual of `Hopf_rule_Z_X` in Listing 2.3, and we show the diagram in Figure 2.20. The `colorswap_of` tactic takes the color-swap of both sides of the provided statement, simplifies trivial components, and uses other proven identities of color-swap to simplify further. We automate proofs about the transposed versions of diagrams using a similar tactic. We can drastically reduce proof overhead and increase proof maintainability using these two automation tactics.

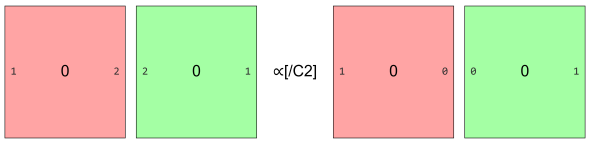


Figure 2.20: Color-swapping the hopf rule, as defined in Figure 2.10c, using automation

```
Theorem Hopf_rule_X_Z : X 1 2 0 ↔ Z 2 1 0 ∝[/C₂] X 1 0 0 ↔ Z 0 1 0.
Proof. colorswap_of Hopf_rule_Z_X. Qed.
```

Listing 2.3: Example of proving the dual of a lemma. See Figure 2.20 for visualization. In file `src/DiagramRules/Bialgebra.v`

Along with these proof automation options, we also have a variety of tactics to automatically deal with structural trivialities, such as superfluous stacks of empty diagrams or compositions with `n_wire`s. With this automation, we can deal with most such structures arising from rewrites in practice. Without the strategies mentioned above, the size of proofs would grow significantly and make them harder to comprehend. These observations confirm previous findings of domain-specific optimizations being a valuable aide to proof engineering (Ringer et al. [2020]).

We also have automation in place to simplify casts. Besides eliminating trivial casts, the automation attempts to merge multiple casts into a single cast by lifting them to the outermost part of the relevant structure. With these tools, we have been able to remove most casts within our diagrams automatically. In cases where our cast automation does not work immediately, we have found that after performing a suitable rewrite within the cast, the automation works successfully.

2.8.3 Using visual proof

*Vy*ZX aims to prove graphical statements. We represent such statements through a syntax tree, as described in Section 2.5, along with a provided visualization, as described in Section 2.7. Our visualization aims to convey the core information that *Vy*ZX diagrams capture: its inductive elements and the overall structure. Specifically, we choose not to use the conventional visualization form of ZX-diagrams, which focuses on connectivity without conveying the integral structural information we need. The integral structural information comes in the form of swaps, caps, and cups, which tell us how wires must bend and cross to make the diagram connect. This explicit information is not present when you only focus on connectivity.

ZXViz is integrated with the rocq-lsp (Developers [2023]) language server to render diagrams of proof goals in the current state automatically. When the proof state in focus changes, the rocq-lsp extension sends updated goals to ZXViz. ZXViz then sends the updated

diagrams to VSCode, which will display them to the proof engineer. This simple workflow augments the proof engineer's preexisting workflow by providing a digestible representation of *Vy*ZX diagrams.

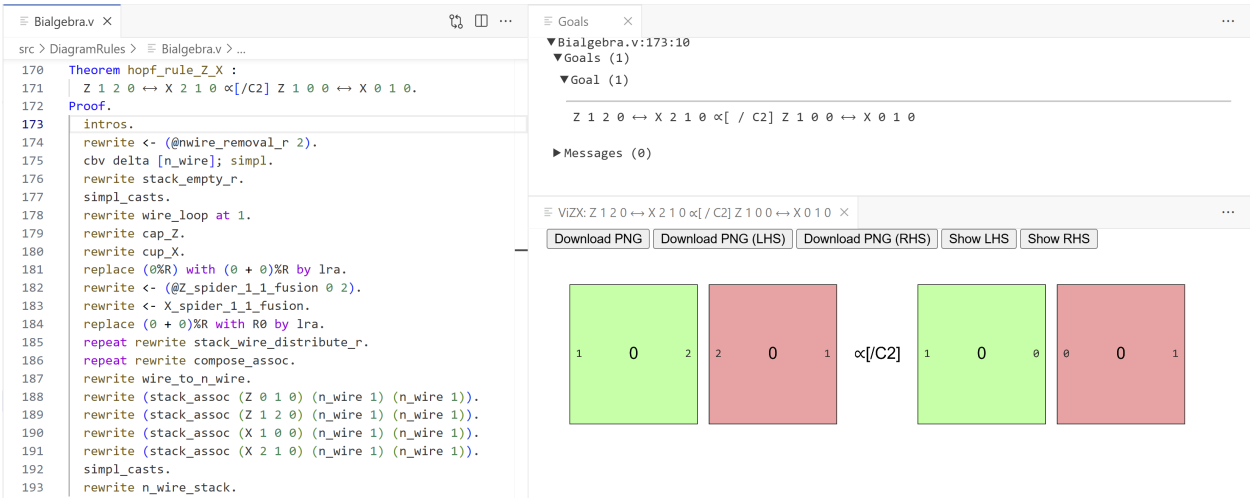


Figure 2.21: ZXViz integrated with the user's proof writing environment.

While simple terms are easy to read using just the textual rendering, more complex diagrams benefit from the visualization. In our work on developing completeness, we found the (C) rule for completeness (Jeandel et al. [2020]) (Figure 2.23) to be difficult to work with textually. It was hard to understand the positional relationship between subdiagrams and how to make progress simplifying them. As this is a semantic proof, it was important to control the computation of matrices carefully to avoid large terms that make the proof very slow. The visualization allowed us to identify repeated structures and easily isolate elements whose linear algebraic representations could be computed explicitly. For example, the Z-spider above a wire composed with an X-spider with phase 0 or π appears four times across the two sides, so replacing these structures with their corresponding linear maps simplified the goal substantially. By repeatedly simplifying subdiagrams, which we could identify using the visualizer, we were able to reduce both structures to a reasonable algebraic form that could be automatically solved.

```
Z 1 2 0 ↔ (Z 0 1 β ↕ - ↔ X 2 1 PI ↕ (Z 0 1 α ↕ - ↔ X 2 1 0))
↔ (Z 1 2 β ↕ Z 1 2 α ↔ (- ↕ X 2 1 γ ↕ -))
↔ (- ↕ Z 1 2 0 ↔ (X 2 1 (- γ) ↕ -) ↕ -)
∝= Z 1 2 0 ↔ (Z 0 1 α ↕ - ↔ X 2 1 0 ↕ (Z 0 1 β ↕ - ↔ X 2 1 PI))
   ↔ (Z 1 2 α ↕ Z 1 2 β ↔ (- ↕ X 2 1 (- γ) ↕ -))
   ↔ (- ↕ (Z 1 2 0 ↕ - ↔ (- ↕ X 2 1 γ)))
```

Listing 2.4: The textual form of the (C) rule for completeness as it appears in the proof environment, visualized in Figure 2.22. Found in file `src/DiagramRules/Completeness.v`

2.9 Conclusion

We presented *Vy*ZX, a formally verified ZX-calculus library. We defined ZX-diagrams not in terms of connectivity, but in terms of structure, so we could define semantics in terms of complex-valued matrices. This structure allowed us to prove facts inductively about ZX-diagrams, which then enabled us to write fully diagrammatic proofs while retaining the guarantees arising from our matrix-level ground truth. Using the inductive structure presented challenges, namely dealing with additional structure and not having access to connection information directly. We presented proof strategies, automation techniques, and a visualizer to deal with these challenges. These allowed us to prove a complete ZX-calculus equational theory, which will help verify the broad range of existing ZX-calculus tools and those yet to be developed. We hope that *Vy*ZX will be used to develop trusted quantum software and the techniques we developed here will prove useful beyond the quantum domain.

Figure 2.22: The completeness rule (C), seen textually in Listing 2.4.

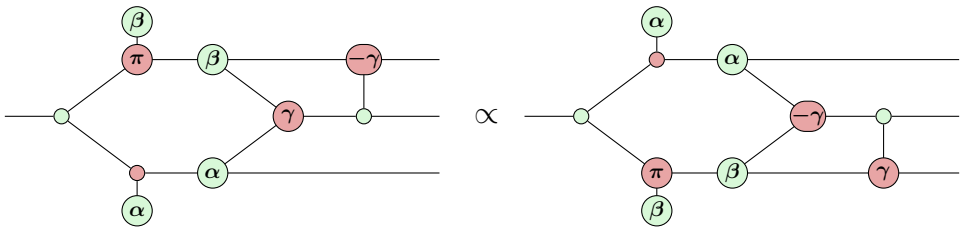


Figure 2.23: The completeness rule (C) as a ZX-diagram.

CHAPTER 3

CONNECTIVITY

3.1 Introduction

In the prior chapter we viewed a specific example of a diagrammatic calculus, the ZX-Calculus, as a purely compositonal structure. While this was sufficient for managing to verify the correctness of the calculus itself and even apply it to some small quantum circuit optimizations the compositional structure led to a long proof process that primarily involved reasoning about associativity of terms. Ultimately if we want to truly capture diagrammatic reasoning, it is insufficient to work exclusively with compositional structures. Compositional structures are still very useful. In particular, they allow us to express terms naturally within our proof assistant. In comparison, working with a graph would be harder to express in a proof assistant but easier to express on pen and paper. We also are able to perform induction in a very natural way on compositional terms where we may struggle if we were working with graphs.

In this chapter, we present TensorRocq, a Rocq tool for verified diagrammatic reasoning about SMCs with respect to tensor semantics. This implementation showcases a number of novel features:

- We automatically resolve equalities derivable from the axioms of SMCs, i.e. only connectivity matters, allowing us to ignore associativity within SMC terms.
- We provide a tactic for diagrammatic rewriting within SMCs, automatically finding and replacing subdiagrams matching proven equalities up to associativity.
- We create a framework for easily defining new SMC theories in the style of Chyp, which are given semantic meaning
- We provide a unifying framework to verify the relationship between SMCs and hypergraphs via tensor semantics.

- We provide an extensible system of typeclasses allowing TensorRocq to be integrated into existing projects by instantiating SMC theories.
- We provide the flexibility to work with existing Rocq libraries without requiring libraries to change their existing definitions or statements of lemmas.

By using the rewrite engine of TensorRocq, paper-style diagrammatic proofs can be directly translated into formal proofs with associativity elided automatically. Focusing on the meaningful rewrites over the noise of associativity makes proofs more readable, because they are no longer dominated by syntactic manipulation. It also makes proofs more resilient to small changes in definitions or statement, as terms are considered up to full SMC equivalence.

3.2 Background

3.2.1 Tensors

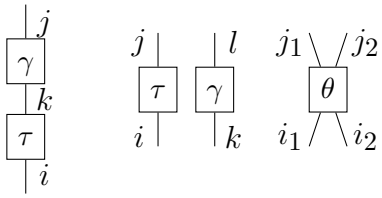


Figure 3.1: A visualization of tensor terms, including contraction, product, and tensors with multiple inputs. Following the standard convention, diagrams should be read from bottom to top.

Tensors are a popular abstraction in classical and quantum physics, where a tensor is given as a set of indexed numbers over a field. The following definitions related to tensors draw on Kissinger [2014], which will we discuss in the context of proof assistants.

Definition 9 (Concrete Tensor)**.** *A tensor τ over a field $\mathbb{F}$ with n inputs and m outputs with domain T gives a value*

$$\tau_{i_1 \ldots i_n}^{o_1 \ldots o_m} \in \mathbb{F}$$

for all choices of $i_1, \ldots, i_n, o_1, \ldots, o_m \in I$*, for some finite indexing set* I*. This can be equivalently represented as* τ_i^o *where* i *and* o *are vectors of lengths* n *and* m *respectively.*

The simplest example of a tensor would be the delta tensor. The delta tensor, often called the Kronecker delta, is 1 when its inputs and outputs are precisely equal and 0 otherwise. As a process, it can be thought of as an identity operator, as it does not change its outputs. As matrices, the identity matrix is the Kronecker delta.

Definition 10 (Delta tensor)**.** *For an arity* n *and field* $\mathbb{F}$*, the delta tensor* δ *is defined so* δ_i^o *is* 1 *if* $i = o$ *and* 0 *otherwise.*

We then also define a process called tensor contraction. This allows us to give some notion of process flow for tensors. When connecting an output to an input, contraction acts as a sequencing operator, moving outputs from one tensor to inputs of another. When connecting inputs between two tensors, it restricts those inputs to always be equal. The same is true for outputs. The notion of requiring inputs or outputs to always be equal is a strange idea for processes, but our goal is to use tensors as a semantic domain, so this extra expressivity is not a problem.

Definition 11 (Tensor Contraction)**.** *The contraction of two tensors* $\sum_{k_1 \in I} \tau_i^k \gamma_k^j$ *is taken by summing over the indexing set* I*.*

If two tensors are interpreted as modeling the behavior of some processes, contraction models the behavior of the composition of these processes. For example, the delta tensor models the identity process, whose output is precisely its input, and this is indeed an identity for contraction. Tensors also offer a way to reason about parallel composition by taking the product of two tensors in the corresponding field.

Definition 12 (Tensor Product)**.** *The product of two tensors* τ_i^j *and* γ_k^j *is given by* $(\tau\gamma)_{ik}^{jl} = \tau_i^j * \gamma_k^l$*, where* $*$ *is multiplication in the underlying field.*

These tensors should be thought of as black-box operations. Many examples exist, notably vector spaces over a field $\mathbb{F}$ can be represented as tensors. The tensor product is then the Kronecker product on matrices, while composition is matrix multiplication. Tensors are a useful tool for reasoning diagrammatically. As tensors have a natural interpretation as traced-symmetric monoidal categories, they have a natural diagrammatic interpretation (Kissinger [2014]).

There is also a dual syntactic notion which avoids writing out tensor contraction explicitly, and says a tensor contraction occurs when two variables are shared. It is common in physics and is known as the *Einstein summation convention*. Definition 11 could then be written as $\tau_i^k \gamma_k^j = \sum_{k \in I} \tau_i^k \gamma_k^j$. We also can visualize these terms as in Figure 3.1.

Tensors are particularly useful for verification. They are a heavily abstracted form of linear algebra which has a natural way to express both composition and connectivity through the same operator. This means they can connect to the other parts of our rewrite engine, hypergraphs and APROPs, which we will see in Section 3.2.2 and Section 3.2.3. Tensors have also been shown to have a natural interpretation as monoidal categories, further cementing their significance in the intersection between hypergraphs and APROPs (Kissinger [2014]).

3.2.2 Hypergraphs

Our definitions for hypergraphs build on those of Bonchi et al. [2022a], extending them with an interpretation of hypergraphs as tensors. A hypergraph is an extension of a graph where an edge can be incident to multiple vertices. In graphs, vertices are usually thought of as "objects" that edges can connect. In hypergraphs, it is common to flip that hierarchy, instead thinking of *edges* as primary, with vertices describing how edges are connected. As such, we often wish to equip edges with some data, which can be used to interpret a hypergraph in a concrete way.

Definition 13 (Labeled Hypergraph)**.** *A labeled hypergraph is an ordered pair containing a*

set of vertices $\mathcal{V} = \{v_0 \dots v_i\}$ *and edges* $\mathcal{E} = (l_0, e_0) \dots (l_j, e_j)$ *where* $e_k \subseteq V$ *for all* k *and* l_k *is the label of the hyperedge.*

With this definition, a graph is an instance of a hypergraph where each e_k has size 2. Labeled hypergraphs have a natural interpretation as tensors if the labels are associated to tensors, with vertices describing what indices to sum over in the resulting tensor. To make this translation concrete, it is easiest to work with *directed* hypergraphs, whose edges' vertex sets are split into inputs and outputs, as tensors have designated inputs and outputs.

Definition 14 (Labeled Directed Hypergraph)**.** *A typed directed hypergraph is an ordered pair containing a set of vertices* $\mathcal{V} = \{v_0 \dots v_i\}$ *and edges* $\mathcal{E} = (l_0, el_0, er_0) \dots (l_j, el_j, er_j)$ *where* $el_k, er_k \subseteq V$ *for all* k *and* l_k *is the label of the hyperedge.* e_l *and* e_r *are lists of vertices rather than sets, fixing the order in which they are incident to the hyperedge.* e_l *is the left edge list, or inputs, and* e_r *is the right edge list, or outputs, of the edge.*

Hypergraphs with Interfaces

Hypergraphs are not sufficient to model process theories, as we often think of a process as having inputs and outputs. To extend our notion of hypergraphs, we add the notion of an interface. Interfaces act as the inputs and outputs of the process represented by the hypergraph as a whole. Chyp refers to these interfaces by their categorical name of *cospans* and Bonchi et al. [2022a] give their categorical interpretation. We use the term interfaces to emphasize their interpretation over their categorical meaning.

Definition 15 (Interface)**.** *An interface for a hypergraph* H *is a pair of left and right ordered lists of vertices* (V_L, V_R) *where* V_L *are considered the "inputs" to the graph and* V_R *are considered the outputs. If* $v \in V_L$ *we have a single input into the vertex* v*. The order of the inputs and outputs matters, and we will take a top-down approach to ordering edges from our interfaces.*

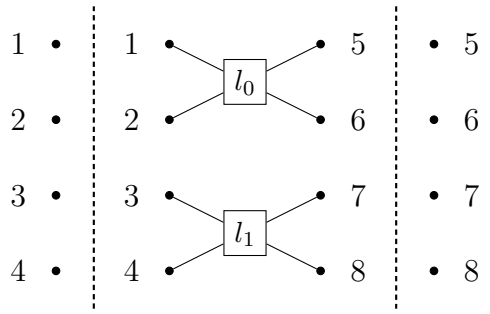


Figure 3.2: A labeled directed hypergraph with interfaces on the left and right. The interfaces fix the order on the inputs to hyperedges, which may differ from the order in their adjacent hyperedges.

We will often write our hypergraph with interfaces as an ordered triple given by (V_L, H, V_R) where the inputs are on the left, outputs on the right, and the graph in the middle. With these new interfaces, we need an updated definition for degree which respects the idea that vertices can have connections to the interfaces as well.

Definition 16 (Tensors of Labeled Hypergraphs with Interfaces)**.** *Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a labeled directed hypergraph with interfaces V_L and V_R, and for each edge $e_k = (l_k, el_k, er_k)$ let a tensor $\tau(e_k)$ be given with $|el_k|$ inputs and $|er_k|$ outputs, over a field $\mathbb{F}$ and finite indexing set A. Then, the tensor semantics of $\mathcal{H}$ is*

$$\tau(\mathcal{H})_i^o = \sum_{f:V\to A} \delta_i^{f(V_L)} \cdot \delta_{f(V_R)}^o \cdot \prod_{(l_k,v,w)\in\mathcal{E}} \tau(e_k)_{f(v)}^{f(w)}.$$

The key feature of tensors and hypergraphs is their emphasis on *connectivity* of tensors. The benefit of using the two in combination is that they have complementary characteristics. Tensors are abstract and not syntactic, but have very clear semantic meaning as functions to fields, while hypergraphs are concrete and syntactic objects but lack the same immediate semantic interpretation. Combining the two allows syntactic reasoning about hypergraphs to prove algebraic facts, as is detailed in (Kissinger [2014]).

The reverse direction is true as well: every tensor expression (meaning a tensor built up

with product and contraction from some set of known, abstract "black-box" tensors) can be represented by a labeled directed hypergraph by interpreting each input, output, and contraction as a vertex, and each abstract tensor as an edge from its inputs to its outputs. The power of this translation is that labeled directed hypergraphs which are isomorphic have equal semantics as tensor expressions. This allows algebraic relations to be proven by graphical reasoning. To show this, we build sequential composition and parallel products of hypergraphs to mirror the contraction and product of their tensor semantics.

For simplicity, composition is defined only for hypergraphs with interfaces which have equally sized interior interfaces, and with only these vertices in common. This can always be ensured by relabeling one of the hypergraphs.

Definition 17 (Vertices)**.** *The set of vertices $V(\mathcal{H})$ of a hypergraph with interfaces $\mathcal{H} = (V_L, H, V_R)$, where $H = (V, E)$, is the set $V_L \cup V_R \cup V$.*

Definition 18 (Composition of Interfaces)**.** *Given two hypergraphs with interfaces $\mathcal{H} = (H_L, H, H_R)$ and $\mathcal{J} = (J_L, J, J_R)$, where $H_R = J_L = V(\mathcal{H}) \cap V(\mathcal{J})$, we define the composition $\mathcal{H}; \mathcal{J} = (H_L, H, H_R); (J_L, J, J_R)$ of these hypergraphs to be $(H_L, H \sqcup J, J_R)$. Here, $H \sqcup J$ stands for the hypergraph whose vertices are the union of the vertices of H and J and whose edges are the disjoint union of the edges of H and J.*

This definition of composition joins hypergraphs along a common interface, and corresponds directly to the contraction of the corresponding tensor semantics.

Lemma 1. *Let $\mathcal{H} = (H_L, H, H_R)$ and $\mathcal{J} = (J_L, J, J_R)$ be hyperedges with interfaces such that $H_R = J_L = V(\mathcal{H}) \cap V(\mathcal{J})$. For each edge $e = (l, el, er)$ belonging to either H or J, let a tensor $\tau(e)$ be given with $|el|$ inputs and $|er|$ outputs, over a field $\mathbb{F}$ and finite indexing set A. Let $n = |H_L|$, $m = |H_R| = |J_L|$, and $o = |J_R|$. Then*

$$\tau(\mathcal{H}; \mathcal{J})_i^k = \sum_j \tau(\mathcal{H})_i^j \tau(\mathcal{J})_j^k.$$

Proof. Let $H = (V, E)$ and $J = (U, F)$, so $V \cap U = H_L = H_R$. By definition, the tensor semantics of the composition is

$$\tau(\mathcal{H}; \mathcal{J})_i^k = \sum_{f:V\cup U\to A} \delta_i^{f(H_L)} \cdot \delta^k_{f(J_R)} \cdot \prod_{(l_k,v,w)\in E\sqcup F} \tau(e_k)^{f(w)}_{f(v)}$$

The contraction of the respective tensor semantics is

$$\tau(\mathcal{H})_i^j \tau(\mathcal{J})_j^k = \sum_{f_V:V\to A} \delta_i^{f_V(J_L)} \cdot \delta^j_{f_V(J_R)} \cdot \prod_{(l_k,v,w)\in E} \tau(e_k)^{f_V(w)}_{f(v)} \cdot \sum_{f_U:U\to A} \delta_j^{f_U(H_L)} \cdot \delta^k_{f_U(H_R)} \cdot \prod_{(l_k,v,w)\in E} \tau(e_k)^{f_U(w)}_{f(v)} \tag{3.1}$$

Observe that the functions f_V and f_U are each required to take values $j_1, \ldots, j_m$ on $H_R = J_L$ (rather, if they do not, the summand is zero because of the delta tensors). Moreover, recall that V and U are disjoint outside $H_R = J_L$; so, we can join these summations to run over $f : V \cup U \to A$ substituting for f_V and f_U, removing the delta tensor terms involving H_R and J_L. Then, observing that

$$\prod_{(l_k,(v_1,\ldots,v_n),(w_1,\ldots,w_m))\in E\sqcup F} \tau(e_k)^{f(w_1),\ldots,f(w_m)}_{f(v_1),\ldots,f(v_n)} = \prod_{(l_k,(v_1,\ldots,v_n),(w_1,\ldots,w_m))\in E} \tau(e_k)^{f(w_1),\ldots,f(w_m)}_{f(v_1),\ldots,f(v_n)} \cdot \prod_{(l_k,(v_1,\ldots,v_n),(w_1,\ldots,w_m))\in F} \tau(e_k)^{f(w_1),\ldots,f(w_m)}_{f(v_1),\ldots,f(v_n)}, \tag{3.2}$$

we arrive at (3.1). □

Definition 19 (Product of Interfaces)**.** *Given two disjoint hypergraphs H, and J, with disjoint interfaces H_L, H_R and J_L, J_R we define the parallel product, or stack, $(H_L, H, H_R)*(J_L, J, J_R)$ to be $(H_L + J_L, H \cup J, H_R + J_R)$.*

An analogous statement to that of Lemma 1 holds relating the tensor semantics of

the product of hypergraphs with interfaces to the product of the tensor semantics of the hypergraphs.

Lemma 2. *Let $\mathcal{H}$ and $\mathcal{J}$ be disjoint hyperedges with disjoint interfaces. Then*

$$\tau(\mathcal{H} * \mathcal{J})_{ij}^{kl} = \tau(\mathcal{H})_i^k \tau(\mathcal{J})_j^l$$

Proof. Let $H = (V, E)$ with interfaces V_L, V_R and $J = (U, F)$ with interfaces U_L, U_R be disjoint hypergraphs and interfaces. Then $\tau(\mathcal{H} * \mathcal{J})_{ij}^{kl}$ is precisely given by

$$\tau(\mathcal{H} * \mathcal{J})_{ij}^{kl} = \sum_{f:V\cup U\to A} \delta_{ij}^{f(I_i)f(I_j)} \cdot \delta_{f(O_k)f(O_l)}^{kl} \cdot \prod_{(l_k,v,w)\in E\cup F} \tau(e_k)_{f(v)}^{f(w)}.$$

We use the fact that $\delta_{ij}^{kl} = \delta_i^k \delta_k^l$ as well as the fact that V, U are disjoint, we can split the function f into its restriction over V and U and split the sum. Similarly we can also split the product as E, U are disjoint, giving us the term

$$\sum_{f:V\to A} \delta_i^{f(I_i)} \cdot \delta_{f(O_k)}^k \cdot \prod_{(l_k,v,w)\in E} \tau(e_k)_{f(v)}^{f(w)} \cdot \sum_{f:U\to A} \delta_j^{f(I_j)} \cdot \delta_{f(O_l)}^l \cdot \prod_{(l_k,v,w)\in F} \tau(e_k)_{f(v)}^{f(w)}.$$

which is precisely

$$\tau(\mathcal{H}) \cdot \tau(\mathcal{J}).$$

□

Rewriting Hypergraphs

Rewriting at the level of hypergraphs is well explored. The general context is that there is a known rewrite rule $T \equiv S$ at the level of hypergraphs, and we want to substitute T for S in a larger hypergraph H containing T as a subgraph. The standard method to accomplish this is to represent G as a *double pushout*, in which it is separated into three parts, the middle

one being isomorphic to S. Then, we can directly substitute S in place of T.

Lemma 3 (Decomposition)**.** *For a hypergraph H with interfaces V_L and V_R and a given subhypergraph T of H, we can form the decomposition of H into C_L and C_R such that there exist V_i, V_j, and V_k such that $H = (V_L, C_L, V_k + V_i); ((V_k, \emptyset, V_k) * (V_i, T, V_j)); (V_k + V_j, C_R, V_R)$*

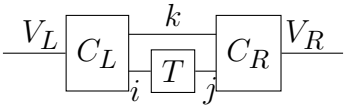


Figure 3.3: The decomposition given by Lemma 3.

Lemma 4 (Double Pushout Rewrite)**.** *For a hypergraph H with interfaces V_L and V_R and a rewrite rule $T = T'$ where T is a subgraph of H, we can perform the decomposition procedure as in Lemma 3, replacing the term T with T'.*

Both Lemma 3 and Lemma 4 have been well explored and proved through the language of cospans of hypergraphs (Bonchi et al. [2022a]). Their inclusion here serves to illustrate the process undertaken by TensorRocq.

3.2.3 PROPs

Diagrammatic reasoning and symmetric monoidal categories (SMCs) are closely tied together, with each SMC coming equipped with a diagrammatic language. In general, SMCs are systems with objects and transformations between them called arrows. We can take tensor products over objects and arrows in order to perform actions in parallel, and we can sequence arrows together to perform multiple actions in a row. For a clear definition of SMCs and their corresponding diagrammatic languages, we recommend the Selinger [2010] survey of the topic.

Definition 20 (PROPs)**.** *A PROP (short for PROducts and Permutations) is a symmetric monoidal category where the objects are natural numbers and the tensor product on objects*

is addition. Specifically we have a collection of arrows labeled as $f : n \to m$ *where* n *is the domain and* m *the codomain of* f*, a braid operator* $\beta_m^n : n \otimes m \to m \otimes n$*, and composition* $f; g : n \to o$ *for arrows* $f : n \to m$ *and* $g : m \to o$*.*

PROPs are useful for tracking generic amounts of nonspecific information, making them perfect for process theories. The object n for some natural number n could be used to represent n booleans and the arrows could be boolean operators such as *AND*, *OR*, and *XOR*. In general we can use them to reason about any kind of information flowing through a system, not just boolean circuit logic. Occasionally we want to put restrictions on two objects being equal in some sense. One way to do this is through the use of Autonomous PROPs or APROPs.

Definition 21 (Autonomous PROP)**.** *Autonomous PROPs are PROPs enhanced with two families morphisms called the unit* (η_n) *and counit* (ϵ_n) *given by* $\eta_n : 0 \to n + n$ *and* $\epsilon_n : n + n \to 0$ *such that* $\eta_n \otimes id_n; id_n \otimes \epsilon_n = id_n$ *and* $id_n \otimes \eta_n; \epsilon_n \otimes id_n = id_n$*.*

What the η and ϵ operators actually stand for becomes more clear in practice when we turn them into tensors or hypergraphs. As a hypergraph with interfaces, η_n as a tensor is a collection of delta tensors (Definition 10) which force the top n outputs to be equal to the bottom n outputs in order. The same is true of ϵ_n, only for inputs. This allows for representing connectivity information which does not respect the directionality of edges. Though these terms are not present in every SMC, they are used extensively in theories that include them. By expressing them syntactically, we can reason about them directly at the level of hypergraph isomorphism, rather than treating them as generators and having to use extra rules.

APROPs are the primary syntactic interface for working with TensorRocq. All theories are stated as APROPs and then translated into hypergraphs to perform rewrites before being extracted back to APROP terms. With that in mind, we present the details of TensorRocq's implementation before proceeding to give examples of its use.

3.3 Implementation

3.3.1 Tensors

The efficacy of tensors as a semantic backing is their *compositionality* and their representation of *connectivity*. Tensors naturally represent sequential and parallel compositions by contraction and product, respectively, so they can encode the horizontal and vertical compositions of a symmetric monoidal category. This is particularly useful because they encode the complex notion of "only connectivity matters" for string diagrams as simple algebraic identities, dramatically simplifying reasoning. This simple equational theory makes reasoning about general tensors significantly simpler than reasoning about more complicated structures like SMC terms, particularly with the automatic solvers over (semi)rings provided by Rocq (Gregoire and Mahboubi [2005]).

```
Definition Tensor (R : Type) (n m : nat) (A : Type) := vec A n → vec A m → R.
Definition DimensionlessTensor (R : Type) (A : Type) := ∀ n m, Tensor R n m A.
```

Listing 3.1: Rocq definition of tensors

To define tensors, we chose to use a shallow embedding using Rocq's functions directly, presented in Listing 3.1. Because a tensor can be any function, we cannot perform any computation on its structure, but we obtain a sufficiently general definition to apply it to a variety of theories. To expose the compositional nature of tensors, we explicitly expose their sizes. This ensures that contractions are well-typed, without having to provide any additional well-formedness conditions. We found it necessary to work with "dimensionless" tensors parametric over size in certain contexts, such as the semantics of hypergraphs where edges may have any number of inputs or outputs. Tensors with fixed dimension can be naturally lifted to dimensionless tensors by taking them to be zero on improperly-sized arguments. This means if we want to restrict the dimensions of a specific tensor or hyperedge then we can, while still working with it as a dimensionless tensor. This is good, because dimensionless

tensors greatly simplify the implementation of our hypergraph.

In practice we require the base type R to have an associated semiring structure, comprising associative, commutative, and distributive addition and multiplication operations, all with respect to some equivalence relation on R. The type A is generally required to be *summable*, meaning there is a designated list of elements of A which can be summed over to perform a contraction. When these structures are present on R and A, we can define the product and contraction of tensors as usual.

3.3.2 Hypergraphs

To implement hypergraphs, we have to make some modifications to the usual mathematical definition to get things to work well in a proof assistant. The standard definition involves a set of edges and a set of vertices, and requires that edges reference only vertices in that designated set. In a proof assistant, sets of general objects are not well-behaved (without assuming axioms, it is necessary to consider sets up to some equivalence relation). Moreover, requiring edges' input and output vertices to be contained in a given set would entail carrying around a proof of that fact, either within the definition itself or as an additional predicate that the graph is well-formed. Instead, we can ensure that this condition is met by computing the set of vertices incident to any edge (including input and output edges for hypergraphs with interfaces). This alone would be insufficient as hypergraphs can have isolated vertices. We store an additional field of `hypervertices` which is used to record extra vertices that may not be mentioned in the edges. Finally, we replace the set of edges with a finitely-supported map from positives to edges, as this makes it easier to reason about functions that combine multiple hypergraphs. Edges have labels of a type `T` that is equipped with some equivalence relation, using the Rocq-std++ library's (Krebbers and Jung) `Equiv` typeclass, which we will here denote by $\equiv_T$. This equivalence relation makes it possible to reason about more general theories in which generators may be equivalent without being literally equal (for example, if

the generators are parameterized by Rocq's rational numbers). We present our definition in Listing 3.2.

```
Record HyperGraph {T} := mk_hg {
  (* The edges of the hypergraph *)
  hyperedges : Pmap (T * list positive * list positive);
  (* Additional vertices of the hypergraph, which are often
    disjoint from the referenced vertices of [hyperedges]
    (in practice, we only care about the subset of [hypervertices]
    not referenced in [hyperedges], but do not enforce disjointness) *)
  hypervertices : Pset;
}.
```

Listing 3.2: Hypergraph built with `Pset` and `Pmap` from stdpp

```
Record CospanHyperGraph {T : Type} {n m : nat} := mk_cohg {
  hedges : HyperGraph T;
  inputs : vec positive n;
  outputs : vec positive m;
}.
```

Listing 3.3: Hypergraph with an interface (cospan) with `n` inputs and `m` outputs

We implement our hypergraphs as a record containing the hyperedges and hypervertices of the graph, as described above. We use the implementations of extensional maps and sets of positives from stdpp (Krebbers and Jung), giving us access to the standard operations. Hypergraphs with interfaces are defined as hypergraphs along with vectors specifying which vertices are the inputs and outputs. We use explicit input and output sizes so that Rocq's typechecking can ensure that compositions are properly sized.

To give tensor semantics to a hypergraph with interfaces, we require that the type `T` be interpretable as dimensionless tensors, as recorded by the `TensorLike` typeclass. This records

that each element t of T corresponds to a tensor, and in such a way that equivalent elements of T (by $\equiv_T$) are mapped to equivalent (dimensionless) tensors. We use this representation rather than taking the edge labels to be `DimensionlessTensor`s so that our algorithms can compare edge labels effectively, as case analysis does not work on `DimensionlessTensor`s. This is essential for reflective isomorphism testing, which underpins our automation.

```
Class TensorLike (R A T : Type) {Equiv T} := {
  interpretTensor (t : T) : DimensionlessTensor R A;
  interpretTensorProper :: Proper (equiv ==>  equiv) interpretTensor
}.
```

Listing 3.4: TensorLike typeclass modified from the original for readability

TensorLike allows us to interpret our type T into DimensionlessTensors, giving us a way to turn labels into processes. The Proper part here tells us that this process preserves equivalence. These tensors give semantic interpretation to hypergraphs against which we can verify the correctness of our operations and relations. Equivalence of tensor semantics is the ultimate notion of hypergraph equivalence that we consider, but the power of hypergraphs is that their *syntactic* equivalence, isomorphism, can be effectively computed without explicitly translating them to tensors. With our modified definition of hypergraphs with interfaces, the definition of isomorphism is slightly modified to accommodate the small difference in the edge map, vertex set, and relation on labels. Specifically, the injective functions f_v and f_e witness an isomorphism between the hypergraphs with interfaces G and H if:

a. The interface (inputs and outputs) of H is the application of f_v to the interface of G

b. There is an edge (t, v, w) indexed by k in G if and only if there is an edge (t', v', w') indexed by $f_e(k)$ such that $t \equiv_T t'$, $v' = f_v(v)$, and $w' = f_v(w)$

c. The (computed) vertex set `vertices H` is the image of `vertices G` by f_v.

We prove the induced relation of isomorphism is an equivalence relation on hypergraphs with interfaces, and moreover that isomorphic graphs have equivalent tensor semantics. This means semantic equalities can be proven by checking the syntactic condition of hypergraph isomorphism.

Because we have defined hypergraphs computationally, we can implement a reflective partial decision procedure for isomorphism checking. This takes the form of an algorithm for finding isomorphisms along with a proof that this algorithm is correct, meaning that isomorphism checking reduces to simple computation with Rocq's reduction engines (Grégoire and Leroy [2002]). Graph isomorphism can be computationally expensive to compute, but in this context it proves tractable. Because the hypergraphs have directed, labeled hyperedges, as well as interfaces, there is very little ambiguity as to what map could induce an isomorphism. In practice, we find that the graph isomorphism test scales well to moderately sized goals, but can become slow on very large graphs. Empirically, it runs in less than a second on all the non-synthetic goals we encountered in our examples, which had sizes up to several dozen edges.

For an SMC term built up from generators by composition and stacking, we can represent the term by a hypergraph with interface, assuming we can represent the generators. To use this representation in a proof assistant, we must verify the correctness of this translation; in TensorRocq this is with respect to the underlying tensor semantics. For an SMC with semantics in tensors over some semiring, we can express that a hypergraph with interface represents a given SMC term by saying they have the same semantics as tensors. This allows us to prove results in existing developments that admit tensor semantics, rather than just abstractly reasoning about graphs.

3.3.3 PROP implementation

To reason about SMC terms via computational reflection, we need a syntactic representation for those terms. Because we consider only categories with tensor semantics, which have natural number dimension, we can restrict our focus to SMCs over natural numbers, giving us PROPs as discussed previously. However, we extend the usual notion of PROPs with cap ($\cap$)and cup ($\cup$) constructors, which connect two outputs and inputs, respectively. These allow us to reason about connectivity in a wider variety of domains. For instance, this means the equations given in Definition 21 reduce to isomorphisms of hypergraphs, and therefore do not need to be explicitly encoded. Moreover, cap and cup can be implemented as tensors, so we do not need an explicit cap and cup in the SMC under consideration. Our SMC rewriting tactic does not produce caps and cups, so it can be used in an SMC without them.

```
Inductive APROP {T : Type} : nat → nat → Type :=
  | Aid n : APROP n n
  | Aswap n m : APROP (n + m) (m + n)
  | Acup n : APROP 0 (n + n)
  | Acap n : APROP (n + n) 0
  | Acompose {n m o} (ap1 : APROP n m) (ap2 : APROP m o) : APROP n o
  | Astack {n1 m1 n2 m2} (ap1 : APROP n1 m1) (ap2 : APROP n2 m2) :
    APROP (n1 + n2) (m1 + m2)
  | Agen (t : T) n m : APROP n m. (* Generators of the category, encoded using T *)
```

APROPs have natural semantics as tensors, assuming a `TensorLike` instance is present for `T`. They can also be converted to hypergraphs with interfaces by direct computation, so we can use the fast isomorphism testing function on hypergraphs to prove equivalences of APROPs simply by evaluating the test on the hypergraph semantics of the APROP terms.

```
Fixpoint APROP_graph_semantics {T n m} (ap : APROP T n m) : CospanHyperGraph T n m
    :=
```

```
match ap with
| Aid n ⇒ id_graph n
| Aswap n m ⇒ swap_graph n m
| Acup n ⇒ cup_graph n
| Acap n ⇒ cap_graph n
| Acompose ap1 ap2 ⇒
    compose_graphs (APROP_graph_semantics ap1) (APROP_graph_semantics ap2)
| Astack ap1 ap2 ⇒
    stack_graphs (APROP_graph_semantics ap1) (APROP_graph_semantics ap2)
| Agen t n m ⇒ graph_of_tensor t n m
end.
```

To perform hypergraph rewriting on APROP terms, we need not only to convert APROPs to hypergraphs, but also to convert hypergraphs back to APROPs. However, not every hypergraph represents a valid APROP term: for example, a non-monogamous hypergraph cannot be represented by an SMC (Bonchi et al. [2022a]). As such, a function from hypergraphs to APROP terms is necessary partial. We implement such a partial function which attempts to convert a hypergraph into an SMC term by proceeding "left-to-right", i.e. from inputs to outputs, extracting hyperedges by horizontal composition. In the case of terms arising from an SMC, the resulting hypergraph is acyclic monogamous, and indeed any acyclic monogamous hypergraph can be represented as an SMC term, hence an APROP term. Because double-pushout rewriting preserves acyclic monogamous hypergraphs, every hypergraph our tactic generates will be representable as an APROP term (Bonchi et al. [2022a]).

3.3.4 Rewriting

We implement rewriting modulo SMC structure based on the double-pushout rewriting described in Bonchi et al. [2022a]. In broad strokes, given a term H and some known

equivalence $L \equiv R$, to rewrite L into R within H we must decompose H into a form in which L appears directly as a subterm. For acyclic monogamous cospans of hypergraphs, this decomposition (the pushout complement) can be computed by separating H into edges with paths into L and edges with paths out of L.

We have several tactics for rewriting, depending on the specific context in which the rewriting occurs, but they have the same structure. For simplicity, we will focus on the tactic which operates on APROPs directly; the tactics which apply to other theories differ only in adding additional steps on either end to first convert the goal into a statement about APROPs and at the end convert the result back to the target theory. As such, the illustrative example is to suppose we have APROP terms H, L, and R, and there is a proof of $L \equiv R$ relative to the APROP tensor semantics. First, we convert the target term H and both sides of the equivalence $L \equiv R$ to hypergraphs G_H, G_L, and G_R, by direct computation. We call an unverified matching function to try to find a subgraph of G_H isomorphic to G_L; this function is a slight modification of the isomorphism test to remove the condition the mapping be surjective. Given a match is found, we call the `decompose` function to split G_H into a composition $G_{C_1}; I \otimes G_{L'}; G_{C_2}$. If matching and decomposition were successful, the hypergraph $G_{L'}$ should be isomorphic to G_L, and thus G_H should be isomorphic to $G_{C_1}; I \otimes G_L; G_{C_2}$. The tactic then attempts to convert G_{C_1} and G_{C_2} to APROP terms C_1 and C_2, which will always be possible when the original terms are SMC terms. Therefore, we should have $H \equiv C_1; I \otimes L; C_2$, which the tactic proves using an isomorphism test. Finally, we can replace L with R because composition and stacking preserve tensor semantics.

The correctness of this rewrite depends only on isomorphism testing and composition, with no assumption that the matching, decomposition, and term extraction from hypergraphs to APROPs are correct. These functions are would be difficult to verify, and more importantly, omitting a verification requirement allows us to optimize performance and result quality. This creates a separation of concerns: the rewrite engine can be implemented and improved

quickly, with isomorphism testing ensuring correctness throughout. However, requiring this additional check does incur a performance penalty. Experimentally, the isomorphism check appears to take up to a third of the time of a rewrite, especially on larger goals. If matching, decomposition, and term extraction were verified, that check could be avoided, speeding up the tactic.

Allowing unverified rewriting engines also expands the scope of applications. As we assume nothing about the matching algorithm, another tool could perform the matching, or even selection of lemma to rewrite, with the assurance that any rewrite performed is correct.

3.4 Examples

3.4.1 Signature Reasoning

There are two ways to use TensorRocq, *theories* and *instantiations*. Building a theory allows you to give a collection of generators and rewrite rules as an APROP which can then be used to build terms and perform rewriting. We call this axiomatized reasoning since it relates to small set of axioms rather than the underlying structures (this is not to be confused with axiomatization in Rocq). Instead, we define a structure called the Signature which allows the user to provide generators, an equivalence relation, and rewrite rules.

```
Structure Signature {A : Type} {SA : Summable A, EqA : EqDecision A} := {
  #[canonical=yes] gens : Type;
  #[canonical=no]  gens_equiv :: Equiv gens;
  #[canonical=no]  gens_equivalence :: @Equivalence gens equiv;
  #[canonical=no]  rules : ∀ {n m}, relation (APROP gens n m);
}.
```

We reason about these rewrites using the same tensor semantics as the rest of the paper. However, the signature structure does not require us to be given a ring. Instead we generate

an appropriate polynomial ring to use our rewriting system. None of this is exposed to the user, and they are able to define a Signature and use our rewrite system immediately. To show this, we will take a look at a simple Frobenius algebra. A Frobenius algebra is an example of a PROP given by four generators

$$m : 2 \to 1, \quad u : 0 \to 1, \quad n : 1 \to 2, \quad v : 1 \to 0.$$

satisfying a small collection of rewrite rules given by

$$(m * id); m = (id * m); m, \quad (u * id); m = id, \quad (id * u); m = id,$$

$$n; (n * id) = n; (id * n), \quad n; (v * id) = id, \quad n; (id * v) = id,$$

$$(n * id); (id * m) = (id * n); (m * id).$$

The first three rules show m and u form a monoid, the second three show n and v form a comonoid. The final rule shows how the monoid given by m and u interacts with the monoid given by n and v. There are two equivalent formulations of this condition.

$$n * id; id * m = m; n \quad id * n; m * id = m; n$$

If m and n are both commutative, then these rules imply our previous rule. However if we have a non-commutative Frobenius algebra we end up needing both. We will now show how to define this Frobenius algebra in TensorRocq and derive the two other forms of the Frobenius condition.

First, we must define a relation with respect to which we will give our rules, as we are not working with a denotational semantic in this implementation. Instead, denotational semantics are generated for us from our given signature. We give one that simply states our

terms share the same dimension, but any relation would do. We also provide notations for the generators beforehand to make writing the signature simpler.

```
Notation "x ≡ y" :=
  (existT _ (existT _ (x%APROP, y%APROP)) :
    {n & {m & (APROP (fin 4) n m * APROP (fin 4) n m)%type}})
  (at level 70).

Notation m := (Agen (0%fin) 2 1).
Notation u := (Agen (1%fin) 0 1).
Notation n := (Agen (2%fin) 1 2).
Notation v := (Agen (3%fin) 1 0).

Definition Frob : Signature bool := {|
  gens := fin 4;
  gens_equiv := eq;
  rules := rules_of_rule_list [

    (* (m, u) forms a monoid *)
    (* rule assoc : *) m * Aid 1 ;' m ≡ Aid 1 * m ;' m ;
    (* rule unitL : *) u * Aid 1 ;' m ≡ Aid 1 ;
    (* rule unitR : *) Aid 1 * u ;' m ≡ Aid 1 ;

    (* (n, v) forms a comonoid *)
    (* rule coassoc : *) n ;' n * Aid 1 ≡ n ;' Aid 1 * n ;
    (* rule counitL : *) n ;' v * Aid 1 ≡ Aid 1 ;
    (* rule counitR : *) n ;' Aid 1 * v ≡ Aid 1 ;

    (* rule frob : *) n * Aid 1 ;' Aid 1 * m ≡ Aid 1 * n ;' m * Aid 1
```

```
  ];
|}.
```

Once we have defined our signature, we must lift these notions from the relation $\equiv$ specifying the rules of the signature to a relation == expressing that terms have the same semantic interpretation in the generated semiring. This comprises restating each rules with this new relation, following a boilerplate procedure.

```
Notation "x == y" :=
  (APROP_semantics x ≡ APROP_semantics y) (* Semantics into the generated semiring
    *)
  (at level 70).

(* (m, u) forms a monoid *)
Lemma assoc : m * id ;' m == id * m ;' m.
Proof. apply rules_hold. repeat constructor. Qed.
Lemma unitL : u * id ;' m == id.
Proof. apply rules_hold. repeat constructor. Qed.
Lemma unitR : id * u ;' m == id.
Proof. apply rules_hold. repeat constructor. Qed.

(* (n, v) forms a comonoid *)
Lemma coassoc : n ;' n * id == n ;' id * n.
Proof. apply rules_hold. repeat constructor. Qed.
Lemma counitL : n ;' v * id == id.
Proof. apply rules_hold. repeat constructor. Qed.
Lemma counitR : n ;' id * v == id.
Proof. apply rules_hold. repeat constructor. Qed.
```

```
Lemma frob : n * id ;' id * m == id * n ;' m * id.
Proof. apply rules_hold. repeat constructor. Qed.
```

This is also the point where we give names to our rules, as we are now lifting them to be used in Rocq proofs. The `rules_hold` lemma shows our equality holds for the generated semiring if that rule is present in the list given as a part of our APROP. Now that our rules are encoded, we can use the string rewriting tactic `srw` to perform string rewrites over Frobenius terms. Practically this means for Frobenius terms, we no longer have to care about associativity of terms but instead are able to reason as if we were looking at a string diagram. We now prove the first of our other two Frobenius rules.

```
Lemma frobL : n * id ;' id * m == m ;' n.
Proof.
  transitivity (u * n * id ;' m * m)%APROP;
  [srw unitL; smcat|].
  (* u * n * ⟦ id ⟧;' m * m == m;' n *)
  srw ← frob.
  (* u * ⟦ id 2 ⟧;' Aswap 2 1;' ⟦ id ⟧ * (n * ⟦ id ⟧;' ⟦ id ⟧ * m);' (Aswap 2 1;' m
    * ⟦ id ⟧;' sw) == m;' n *)
  srw assoc.
  (* u * ⟦ id 2 ⟧;' n * ⟦ id 2 ⟧;' ⟦ id ⟧ * (⟦ id ⟧ * m;' m) == m;' n *)
  srw frob.
  (* m * u;' sw;' (⟦ id ⟧ * n;' m * ⟦ id ⟧) == m;' n *)
  srw unitL.
  (* m;' (n;' sw);' sw == m;' n *)
  smcat.
Qed.
```

Two tactics are present here, `smcat` which solves simple equations using hypergraph

isomorphism. This tactic is sufficient to solve equations where the only difference is syntactic, up to connectivity. If the underlying hypergraphs are isomorphic, then `smcat` can solve the equation. The second tactic `srw` is the true rewriting tactic. It takes a term `lhs == rhs` and attempts to perform a replacement of the `lhs` term with the `rhs` term in the current goal as a hypergraph.

We can now use `frobR` in further rewrites. This means we can prove the other Frobenius rule with just two rewrites.

```
Lemma frobR : id * n ;' m * id ==  m ;' n.
Proof.
  srw ← frob.
  (* n * ⟦ id ⟧;' ⟦ id ⟧ * m ==  m;' n *)
  srw frobL.
  (* m;' n ==  m;' n *)
  smcat.
Qed.
```

This part of TensorRocq is implemented in such a way to give us feature parity with Chyp. Much effort has been put towards making this as simple as possible, including automatically generating `APROP_semantics` and lemmas such as `rules_hold` to allow for simple boilerplate code in order to lift lemmas from the rules of the signature to Rocq's rewrite system. Nevertheless, it is still somewhat inconvenient to include this boilerplate, and it is possible that it could be avoided with the development of a Rocq plugin. We also currently require that all generators and relations of the signature are given at once, whereas Chyp allows them to be added throughout the development. This difference arises primarily because we give semantic meaning to our rewrite system from the signature itself, so modifying the signature changes the semantics.

Our semantic backing allows us to prove our syntactic reasoning sound: Any semantics for APROPs over the signature's generators that satisfies the relations of the signature will

also satisfy any relation we prove syntactically. This means that a theory can be developed once and applied to many different projects. However, this method of reasoning still requires using signatures, which is not quite suitable for reasoning about existing developments with their own notions of semantic correctness. In the next section, we will discuss how we bridge the gap between our APROPs and existing work, showing that TensorRocq is a tool for reasoning not just about the theory of SMCs, but also concrete instances of SMCs, using powerful diagrammatic rewriting.

3.4.2 Instantiated Reasoning: The ZX Calculus

A major advantage of working in a proof assistant is that TensorRocq can make statements grounded in mathematical descriptions of semantics, not just abstract rewrite systems. To apply TensorRocq reasoning to other projects, we provide a framework to apply its tactics to any goal that can be framed using tensor semantics.

As our rewrite engine is based on the APROP structure, it is necessary to frame goals in that context. However, a vast number of theories can be described as symmetric monoidal categories, and we do not wish to constrain projects to using the APROP type directly[1]. Therefore, we provide a system of typeclasses that can be instantiated to enable TensorRocq to reason about an existing project.

The general process by which our rewrite engine can be used on a target SMC is as follows:

1. Specify an APROP representing the theory by defining generators and their semantics.
2. Define tensor semantics for the target theory, and show that terms with equal tensor semantics are equivalent in the theory.
3. Provide the definitions of sequential and parallel composition in the target theory via `APROPlike`, and prove their tensor semantics are the expected contraction and product.

1. For instance, the theory of rings can be described as an SMC with generators for 0, 1, +, and $*$ and SMC equations representing the axioms.

4. Instantiate typeclasses describing how to convert a term in the target theory to an APROP with equivalent tensor semantics (which we call "quoting" the term), and how to convert an APROP to a term in the target theory (which we call "denoting" the APROP).

5. Instantiate our rewrite tactics with these interfaces.

This system is designed to make minimal expectations of the target theory. We do not require proofs that every term is represented by an APROP, nor that every APROP represents a term; the tactics will only work in contexts where the relevant terms can be represented by APROPs. Providing quotation and denotation by user-instantiated typeclasses means tactics can be used in contexts with derived terms that are not written directly as generators, increasing their flexibility. These quotation and denotation processes can also be extended with new terms throughout the development of the project without needing to redefine the APROP interpretation.

For a practical example, we describe the application of TensorRocq to the work from Chapter 2, called *Vy*ZX. To use our rewrite engine, we first have to have an idea of the theory we are targeting. We bring up the inductive datastructure defined in Chapter 2.

```
Inductive ZX : nat → nat → Type :=
  | Empty : ZX 0 0
  | Cup  : ZX 0 2
  | Cap  : ZX 2 0
  | Swap : ZX 2 2
  | Wire : ZX 1 1
  | Box  : ZX 1 1
  | X_Spider n m (α : R) : ZX n m
  | Z_Spider n m (α : R) : ZX n m
  | Stack {n_0 m_0 n_1 m_1} (zx0 : ZX n_0 m_0) (zx1 : ZX n_1 m_1) :
```

```
        ZX (n_0 + n_1) (m_0 + m_1)
  | Compose {n m o} (zx0 : ZX n m) (zx1 : ZX m o) : ZX n o.
```

Listing 3.5: CoreData/ZXCore.v in the VyZX repository

Equivalence on *VyZX* diagrams is typically given as *proportionality* of semantics by a constant, nonzero complex multiple `c : C`, denoted in *VyZX* by the relation $\propto[c]$. However, the definition of equivalence corresponding directly to the equality of tensor semantics is $\propto=$, which stands for $\propto[1]$, i.e. semantic equality. For this instantiation, we chose to only work with $\propto=$, as *VyZX* provides tools to automatically convert statements of proportionality to statements of semantic equality by adding a scalor factor encoded as a ZX-diagram. To represent ZX-diagrams, our APROP type is chosen to be `option (bool * R + C)`. All we need to do is choose a type with enough information for each of the processes we want to represent. The value `None` stands for the Hadamard box, `Box`, while the `C` represents a scalar factor via `zx_of_const`. The values (`false`,α) and (`true`,α) represent the `Z_Spider` and `X_Spider`, respectively, with phase α. These generators are sufficient to represent all other definitions of *VyZX*. We then provide tensor semantics for this APROP by giving the standard tensor interpretations of each generator, completing step (1).

The given semantics of *VyZX* diagrams as `QuantumLib` matrices (INQWIRE Developers [2022]) can be easily converted to and from tensors. Therefore, we can easily show that ZX-diagrams with equivalent tensor semantics have equivalent matrix semantics, which is step (2). Providing the composition and stacking of `ZX` completes step (3).

```
#[refine] Instance ZX_APROPlike : APROPlike C bool ZX (@Compose) (@Stack) := {
  interpretDiagram n m zx := ZX_tensor_semantics zx;
}.
Proof.
  abstract (intros n m d d' Heq%matrix_of_tensor_of_equiv;
    rewrite 2 ZX_tensor_semantics_correct in Heq;
```

```
    prep_matrix_equivalence;
    exact Heq).
  abstract (easy).
  abstract (easy).
Defined.
```

Once we have built an APROPLike for our target theory, we must be able to quote between our APROP and the target theory to allow translation. This is accomplished through mostly boilerplate code using the DiagramQuote typeclass. The statement

```
DiagramQuote (APROPlikeD:=APROPModel_for_theory) (TensT:=
    APROP_tensor_interpretation) d a.
```

says that the diagram `d` in our target theory is semantically equivalent to the term `a` in our APROP.

```
Local Notation Quote := DiagramQuote (APROPlikeD:=ZX_APROPlike) (TensT:=ZXCCALC).
```

For example, we can take a term `n_wire n` and the term `Aid n` and show they are semantically equivalent with the instance `zx_quote_n_wire_n`.

```
#[export] Instance zx_quote_n_wire n : Quote (n_wire n) (Aid n).
Proof.
  constructor.
  apply matrix_of_tensor_inj.
  rewrite ZX_tensor_semantics_correct.
  rewrite matrix_of_tensor_delta.
  now rewrite n_wire_semantics.
Qed.
```

Implementing a quotation instance for all of the basic structures allows us to use typeclass resolution to translate terms in our target theory into APROPs. Other terms can be given quotations later as desired, if new derived constants are defined. To make typeclass resolution

behave well[2], we must separately specify the other conversion, from APROP to ZX-diagram. For this we use the typeclass `DiagramDenote`, which follows the same pattern of boilerplate.

```
Local Notation Quote := (DiagramDenote (APROPlikeD:=ZX_APROPlike) (TensT:=ZXCCALC
    )).

#[export] Instance zx_denote_n_wire n : Quote (n_wire n) (Aid n).
Proof.
  constructor.
  apply matrix_of_tensor_inj.
  rewrite ZX_tensor_semantics_correct.
  rewrite matrix_of_tensor_delta.
  now rewrite n_wire_semantics.
Qed.
```

Once the quoting and denoting of step (4) are finished, we can now translate automatically between APROP terms and terms of our target theory. The last step needed is to instantiate new rewrite tactics for this target theory which use these instances to move freely between our APROPs and the target terms, so that the tactics can determine the proper instances to apply.

```
(* Prove that ZX terms corresponding to isomorphic hypergraphs are ∝= *)
Ltac zxcat := wild_cat ZX_APROPlike ZXCCALC.

(* Simplify the LHS by removing extraneous identities *)
Ltac zxclean_lhs := wild_clean_lhs ZX_APROPlike ZXCCALC.

(* Simplify the RHS by removing extraneous identities *)
```

2. Specifically, we want `DiagramQuote` to examine the concrete term and try to generate an APROP, while `DiagramDenote` should examine the APROP to generate a concrete term. Combining the two can lead to major performance issues, and gives less control over the generated terms.

```
Ltac zxclean_rhs := wild_clean_rhs ZX_APROPlike ZXCCALC.

(* Simplify both sides of the goal by removing extraneous identities *)
Ltac zxclean := wild_clean ZX_APROPlike ZXCCALC.

(* Rewrite [lem] at occurence number [match_num] in the LHS, up to SMC equivalence
   *)
Ltac zxrw_lhs lem match_num := wild_rw_lhs ZX_APROPlike ZXCCALC lem match_num.

(* Rewrite [lem] at occurence number [match_num] in the RHS, up to SMC equivalence
   *)
Ltac zxrw_rhs lem match_num := wild_rw_rhs ZX_APROPlike ZXCCALC lem match_num.

(* Rewrite [lem] at occurence number [match_num] in the goal, up to SMC equivalence
    *)
Ltac zxrw lem match_num := wild_rw ZX_APROPlike ZXCCALC lem match_num.
```

Listing 3.6: Rewriting tactics lifted to the VyZX grammar

With that, `zxrw` can now be used in proofs in our target theory. To show the power of this rewriting tactic, we take a look at an example proof from *Vy*ZX, which shows that three CNOT gates implement a swap operation: `Lemma _3_cnot_swap_is_swap : _CNOT_` $\leftrightarrow$ `_NOTC_` $\leftrightarrow$ `_CNOT_` $\propto[/\ (2 * \sqrt{2})]\ \bigtimes$. (Lehmann et al. [2026]). In *Vy*ZX, this proof took 45 lines of code. A large majority of these lines are dedicated to associating terms to expose a subdiagram exactly, so that Rocq's `setoid_rewrite` tactic can perform the rewrite. This reassociation code can be annoying to write, and it is very fragile, as it depends intimately on the term structure of the goal. Using our rewrite tactic, this is reduced to only 17 lines of proof, 5 of which have to do with the specifics of how we integrate with *Vy*ZX's $\propto=$ relation. Moreover, our rewrites depend only on the hypergraph interpretations, so are much more resilient to

any changes in definition.

We first show that two consecutive `_CNOT_` gates (in *VyZX*, `Z 1 2 0 * id ; id * X 2 1 0`) cancel:

```
Lemma _CNOT_CNOT_cancels : _CNOT_ ↔ _CNOT_ ∝[/ 2] n_wire 2.
Proof.
  apply prop_by_iff_zx_scale.
                      (* Change relation to ∝= by adding in scalar ZX diagram *)
  split. 2:{ apply nonzero_div_nonzero; nonzero. }
                                      (* Show above operation is reasonable *)
  zxrw (@dominated_Z_spider_fusion_top_left 2 0 1 1 0 0).
                                                 (* Fuse the top Z spiders *)
  zxrw (@dominated_X_spider_fusion_bot_right 2 0 1 1 0 0).
                                               (* Fuse the bottom X spiders *)
  rewrite Rplus_0_l.                                  (* Correct rotations *)
  zxrw (to_gadget hopf_rule_Z_X_vert 1 1 1 1 0 0 eq_refl).
                       (* Remove 2 connections between top and bottom spiders *)
  rewrite Z_is_wire, X_0_is_wire.                    (* Now they are wires *)
  zxcat.                    (* Equivalent because connections are all the same *)
Qed.
```

And then we can show that three alternating CNOTs form an identity:

```
Lemma _3_cnot_swap_is_swap : _3_CNOT_SWAP_ ∝[/ (2 * √2)] ⨉.
                            (* _CNOT_ ; _NOTC_ ; _CNOT_ = sw *)
Proof.
  apply prop_by_iff_zx_scale.
                        (* Change relation to ∝= by adding scalar ZX diagrams *)
  split. 2:{ apply nonzero_div_nonzero, Cmult_neq_0; nonzero. }
                                    (* Show the above operation is reasonable *)
```

```
  rewrite cnot_is_swapp_notc at 2.     (* Add in a swap and change the third  *)
  rewrite notc_is_notc_r.                       (* Move the cnot back and forth *)
  zxrw (to_gadget bi_algebra_rule_X_over_Z).
                        (* Find and automatically perform a bialgebra rewrite *)
  zxrw (to_gadget _CNOT_CNOT_cancels).    (* Cancel the two CNOTs that result *)
  zxrw (symmetry (zx_of_const_mult (/ C2) (/ √ 2))).      (* Fix up constants *)
  rewrite Cinv_mult_distr.                          (* More constant corrections *)
  zxcat.                                                (* Graphs are equivalent *)
Qed.
```

These proofs correspond more closely to the diagrammatic version of these proofs than the original *Vy*ZX proofs, as they emphasize diagrammatic transformations and suppress term-level associativity concerns. We did find some limitations with using our rewrite engine on *Vy*ZX, namely to deal with caps and cups. Because we chose to implement caps and cups within the APROP structure, we are able to reason about isomorphism up to the yanking equations. However, the matching and decomposition functions currently target SMC terms exclusively, and do not support caps and cups. Moreover, the "spiders" that serve as generators for the ZX-calculus have very strong permutative properties, with the ability to convert between inputs and outputs, as well as absorb permutations into the generators. TensorRocq makes reasoning about these permutations easier, as yanking can be inferred, but it is still necessary to manually state the transformations to apply to spiders to make them isomorphic as hypergraphs. Encoding this permutativity directly could allow for much stronger isomorphism checking, or even rewriting, up to the full notion of only connectivity matters for the ZX-calculus.

3.5 Generalizing further

This system is sufficient for describing a specific structure of processes, one given by autonomous structures. However, other structures may exist. In this section we explore how to generalize our APROP definition further so we can talk about the different types of structural morphism present in a theory. This allows us to set the stage to work with different structures over a theory involving the same processes.

3.5.1 Separating Structure

We first generalize the Prop structure given in Section 3.3.3 to include terms which can have restricted size. We also treat these terms differently, translating them precisely into hypergraph terms without hyperedges, meaning they do not represent a process just connections between processes. This is a small generalization that allows us to think about the different ways information can flow through our graph.

```
Inductive PRO  {Struct : nat → nat → Type} {Ty : Type} : nat → nat → Type :=
  (* Composition of processes *)
  | compose {n m o} (ap1 : PRO n m) (ap2 : PRO m o) : PRO n o
  (* Parallel products of processes *)
  | stack {n1 m1 n2 m2} (ap1 : PRO n1 m1) (ap2 : PRO n2 m2) :
    PRO (n1 + n2) (m1 + m2)
  (* Structural generators which can restrict sizes they operate over *)
  | Pstruct (n m : nat) (s : Struct n m) : PRO n m
  (* Nonstructural generators which must be defined for all sizes *)
  | Pgen (t : Ty) n m : PRO n m.
```

We give this the name `PRO` standing for `Products` as we can take the compositional product or parallel product of processes. The remaining parts of the inductive give us structural morphisms `s` which are used to represent elements such as `Acup` or `Acap` from our original

definition. These definitions are translated into hypergraphs containing only vertices. The term `Pgen` behaves similarly to `Agen`, where it represents a hyperedge in our `PRO` term.

To recover `AProp`s as we had before, we must define the `Struct` which gives us caps, cups, and swaps. We can define this in two separate layers, the first being permutations and the second being Autonomy.

```
Inductive Permutation : nat → nat → Type :=
  | Swap n m : Permutation (n + m) (m + n)
  | Pid  n   : Permutation n n.

Inductive Autonomy : nat → nat → Type :=
  | Cap n : Autonomy 0 (n + n)
  | Cup n : Autonomy (n + n) 0.
```

Defining terms in this way gives us fine-tuned control over the graph structures present in our hypergraphs. It is known that with only permutations, we are representing PROPs. PROPs are equivalent to a class of hypergraphs called monogamous acyclic hypergraphs (Bonchi et al. [2022b]). There are 3 types of structure which relate `PRO`s to Hypergraphs. The last one we define is Cartesian, where we are able to state delta terms directly in our PROP structure. These correspond to a single vertex in a hypergraph.

In order to work with these new structures, we have to define semantic functions and translation into hypergraphs.

```
Inductive SCartesian : nat → nat → Type :=
  | Delta n m : SCartesian n m.

Definition Autonomous (n m : nat) := (Permutation n m + Autonomy n m)%type.

Definition Cartesian (n m : nat) := (Autonomous n m + SCartesian n m)%type.
```

```
Definition PROP := PRO Permutation.

Definition APROP := PRO Autonomous.

Definition CPROP := PRO Cartesian.
```

While PROPs have been shown to be equivalent to monogamous acyclic hypergraphs, which is enough for rewriting any type of PROP, including APROPs, the relation between these other structures and hypergraphs remains open. For each of these structures, a more specific form of double-pushout rewriting could also be developed, even though the standard PROP rewriting is sufficient.

Definition 22. *A hypergraph with interfaces is bi-monogamous if all vertices have degree* 2*. This includes connections to inputs or outputs.*

Conjecture 1. *There exists a function from Autonomous PROPs to bi-monogamous labeled hypergraphs which preserves tensor semantics and distributes over composition.*

Conjecture 2. *There exists a similar function for Cartesian PROPs and labeled hypergraphs.*

3.5.2 Denoting and Quoting

Denoting and Quoting are important parts of building a rewrite engine, acting as the ability to ingest and extract terms. The differentiation between denotation and quotation is purely up to which direction they operate in. In our case, denotation takes us from the goal to our internal rewrite engine. Quotation is the return trip, taking a term in our graph to an AProp term.

```
(* A typeclass recording that the [AProp] term [a] is a 'quotation' (reification)
   of the diagram [d]. In particular, [d] should be known and [a] will be
   determined by typeclass resolution. This class must be instantiated for each [
   APROPlike] type [D] of diagrams. *)
Class DiagramQuote
```

```
  {APROPlikeD : APROPlike R rO rI radd rmul req A D compD stackD}
  {Equiv T, Equivalence T equiv} {TensT : !TensorLike R A T}
  {n m} (d : D n m) (a : AProp T n m) := {
  diagram_quote : interpretDiagram d = AProp_semantics (TensT:=TensT) a;
}.

(* A typeclass recording that the diagram [d] is a 'denotation' (evaluation) of the
     [AProp] term [a]. In particular, [a] should be known and [d] will be
    determined by typeclass resolution. This class must be instantiated for each [
    APROPlike] type [D] of diagrams. *)
Class DiagramDenote {APROPlikeD : APROPlike R rO rI radd rmul req A D compD stackD}
  {Equiv T, Equivalence T equiv} {TensT : !TensorLike R A T}
  {n m} (d : D n m) (a : AProp T n m) := {
  diagram_denote : interpretDiagram d = AProp_semantics (TensT:=TensT) a;
}.
```

Denotation and Quotation are defined exactly the same. We separate them as classes so we can control when they are used via typeclass resolution. What separates this process from traditional reflection is that we are reflecting relative to some denotational semantics, given by our tensor domain. Typically, reflection is done between boolean valued statements and Props and therefore does not need to rely on denotational semantics.

3.6 Conclusion

TensorRocq closes the gap between on-paper proof and formal verification for symmetric monoidal categories. We enable true diagrammatic reasoning, not only proving structural equivalences automatically, but enabling automatic, verified rewriting modulo SMC equivalence. This allows SMC terms to be thought of as diagrams, with associativity handled being automatically by powerful, efficient automation. We have also demonstrated how our

extensible framework can be applied to existing projects, like the *Vy*ZX ZX-calculus library (Lehmann et al. [2026]).

Diagrammatic rewriting makes proofs much shorter and more readable than the previously-necessary manual manipulation of associativity. TensorRocq's diagrammatic rewriting is verified with respect to tensor semantics, so it can be used soundly in any theory interpretable within tensors. We also provide a structure to define abstract rewriting systems by generators and relations, in the style of Chyp (Kissinger [2023]). These abstract systems can also be instantiated with concrete implementations, and we prove that equations proved in the abstract system hold in the concrete instantiation.

CHAPTER 4

CONCLUSION

4.1 Future Work

There are a few open threads in this work which can still be tugged on. Visualization of AProp terms within Rocq also remains open, due to changing technologies between *Vy*ZX and TensorRocq. A similar tool, Chyp (Kissinger [2023]), has an integrated visualizer which allows for the easy identification of which diagrammatic rules to apply. As TensorRocq is embedded within Rocq, visualization and interactivity are more difficult to implement. Recent work by Damien Pous (Pous [2026]) shows that it is possible to copy Rocq code to an external visualizer, perform graphical rewrites there, and extract the proofs back to Rocq. Integrating this with Rocq-LSP would give the best of both worlds in terms of allowing easy visualization and rewriting.

Further extensions and improvements of the rewrite engine, including a full verification of the system remain open problems. The theoretical backing of the engine is strong and we have found it can be applied to existing projects such as *Vy*ZX.

The checked-rewrite structure of our rewrite engine makes it easy to extend to categories with additional properties. For instance, currently we can only reason about caps and cups for the purposes of isomorphism, not rewriting, as the matching engine is based on decomposition for a symmetric monoidal category. Implementing matching and decomposition procedures would make the rewrite engine significantly more powerful in domains with caps and cups. In principle, it should also be possible to encode other structural conditions, such as the premutativity of the Z and X spiders in the ZX-calculus, to enable rewriting up to these more flexible notions of connectivity.

Lean has a visualization system for string diagrams that this style of rewriting could work well with. Building a hypergraph-based rewriting tactic would be a natural fit for the Lean

proof assistant. This would involve a full rewrite within Lean, but is certainly something we would find interesting. This would also entail building tensors and hypergraphs within Lean.

4.2 Conclusion

Through this thesis, we have connected compositional structures commonly found in categorical semantics to their compositionally defined counterparts, hypergraphs with interfaces, within a proof assistant. We show how to give them a common denotational semantics which allows for the development of a hypergraph-based rewrite system.

Through this, we have closed the gap and enabled true diagrammatic rewrites within the Rocq proof assistant. We have also used these techniques to develop a verified library for ZX-Calculus reasoning. We also develop a rewrite engine which can be applied to external projects, given we can reflect them into our internal APROP structures with tensor semantics. We also provide a system to quickly create a theory from a given collection of rules through our signature datastructure, allowing for axiomatized reasoning.

In the development of the *Vy*ZX project we found a large amount of proof lines were dedicated to manually manipulating the associativity of terms. By working with hypergraphs, this is completely avoided. In avoiding this not only are we reducing the number of lines of proof but also the complexity of understanding what is actually necessary to prove a statement. Through *Vy*ZX and TensorRocq we close the gap between proof assistants and diagrammatic reasoning, developing a robust rewrite system which can be readily applied to existing projects.

APPENDIX A

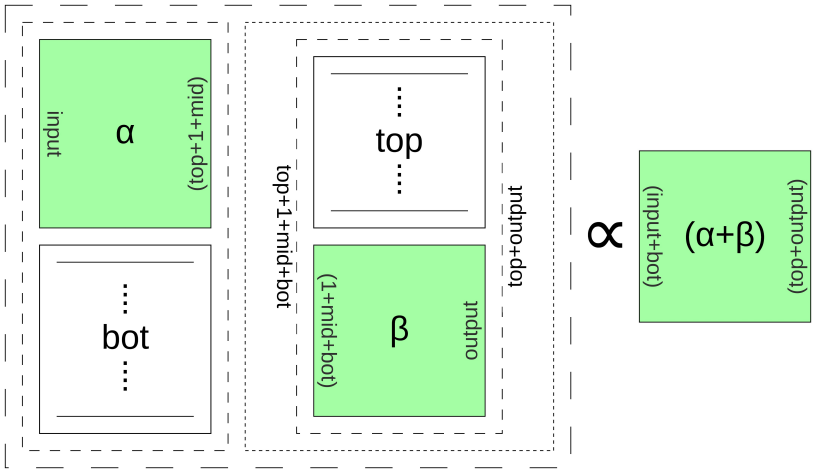
THE COMPLETE EQUATIONAL THEORY OF *VYZX*

In this appendix, we present a complete set of ZX-calculus rules proven within *Vy*ZX and rendered using ZXViz. We do not show all of the rules proven within *Vy*ZX, but instead a sufficient subset for complete equational reasoning.

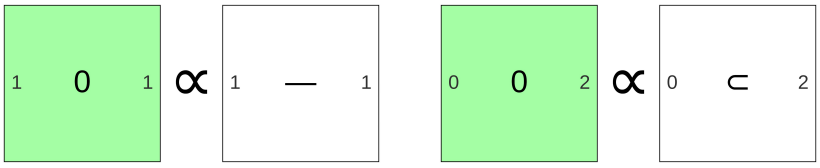
The primary source for these rules is Jeandel's work(Jeandel et al. [2020]), but the rules are slightly modified with inspiration from v. d. Wetering's survey(van de Wetering [2020]). These rules should be compared to Jeandel's work([Jeandel et al., 2020, Figure 2]), where the main differences are:

1. *Vy*ZX lacks a rule E, as it is a scalar based rule,
2. *Vy*ZX uses a more general rule, Figure A.1e, instead of rule K,
3. *Vy*ZX uses a more general rule, Figure A.2a, instead of rule B1,
4. *Vy*ZX does not have explicit scalars in its rewrites.

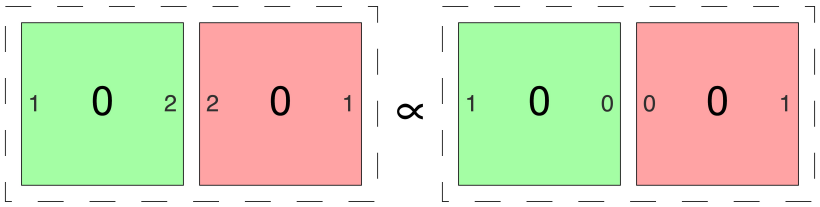
These differences do not impact the universality of the system as *Vy*ZX contains more general rules (Jeandel et al. [2020]).



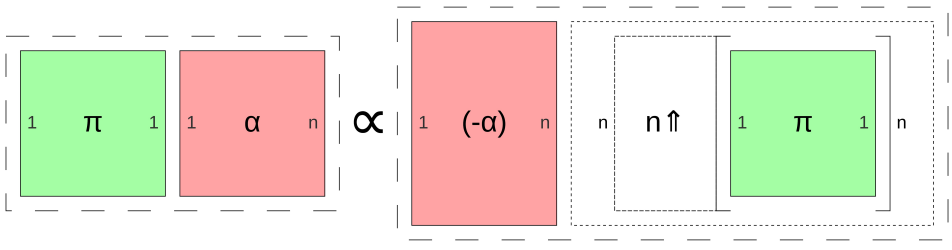
(a) Spider fusion in *Vy*ZX



(b) Blank spider removal to wire (c) Blank spider removal to cup

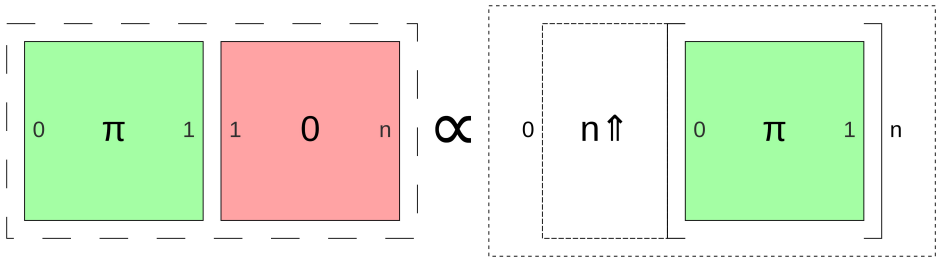


(d) Hopf rule in *Vy*ZX

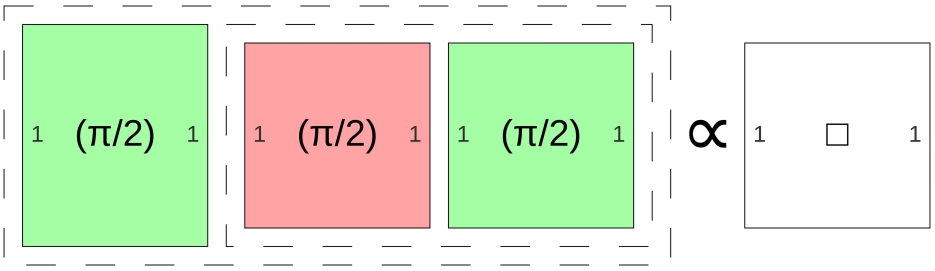


(e) Pi-copy rule in *Vy*ZX

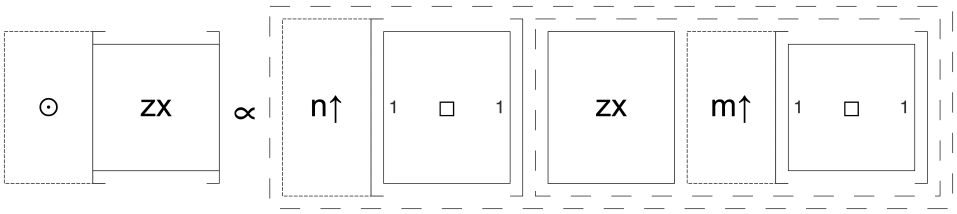
Figure A.1: Directlystateable *Vy*ZX rules



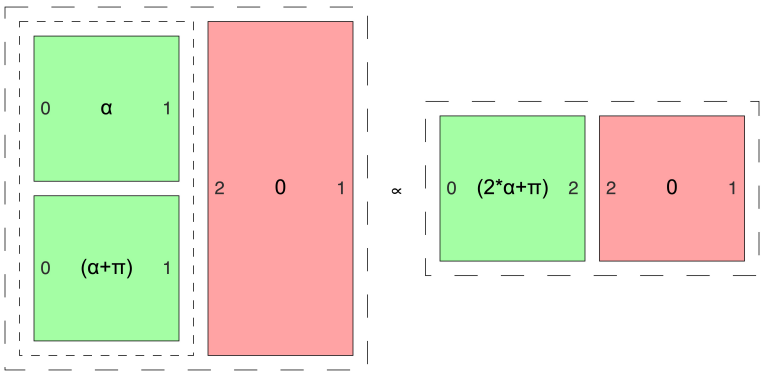
(a) State copy rule in *Vy*ZX, also holds for a Z rotation of 0



(b) Hadamard decomposition in *Vy*ZX



(c) Bi-Hadamard color swapping *Vy*ZX, where ⊙ represents a "color-swapped" diagram where all red spiders are replaced with green spiders and vice-versa



(d) The SUP rule in *Vy*ZX

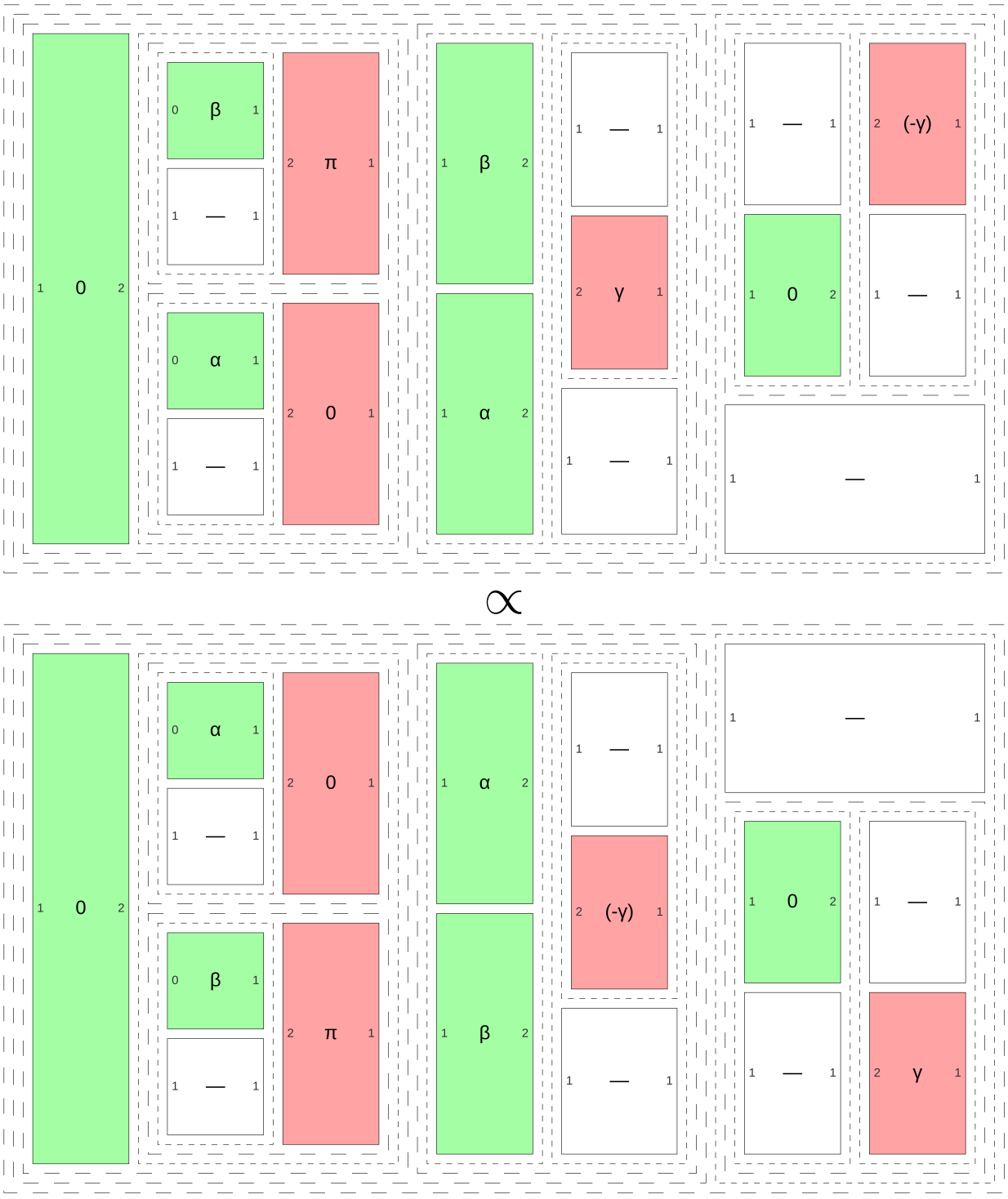


Figure A.3: The C rule in *Vy*ZX

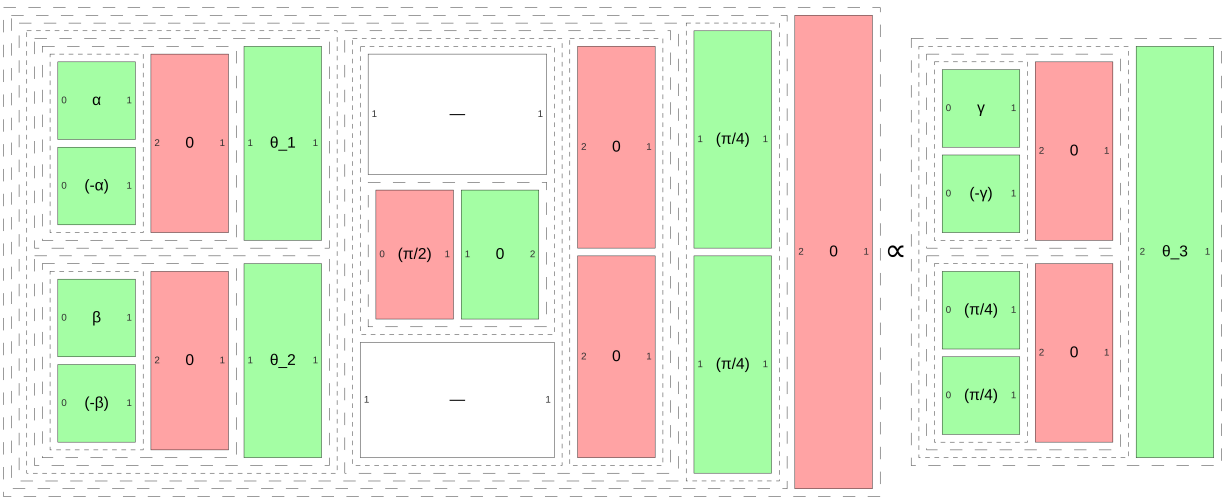
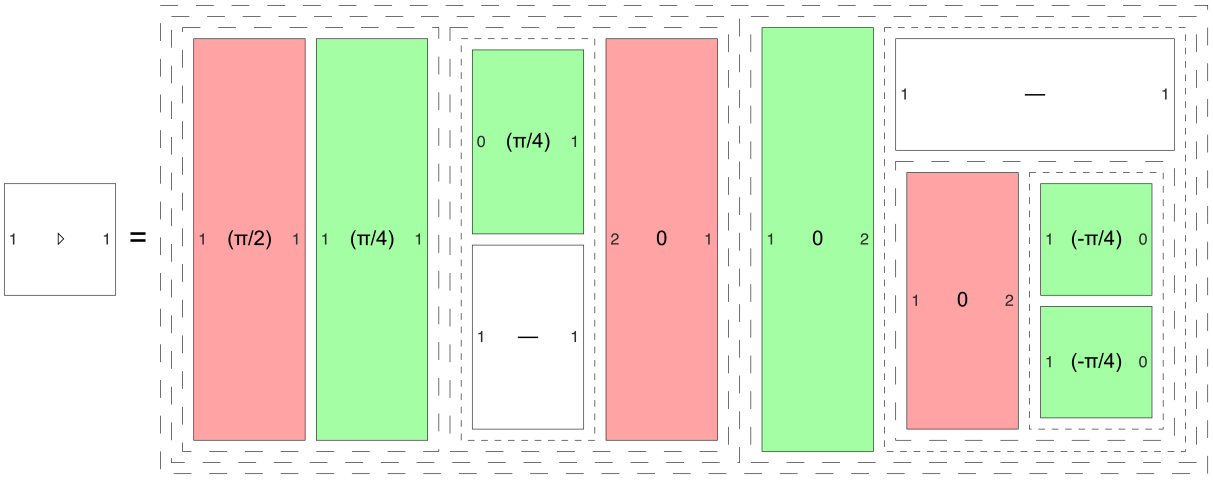
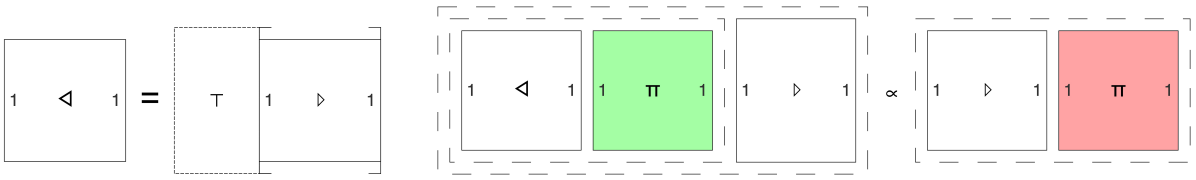


Figure A.4: The N rule in *VyZX*



(a) Definition of ▷



(b) Definition of ◁

(c) The BW rule

Figure A.5: The BW rule in *VyZX*

REFERENCES

Edward W. Ayers, Mateja Jamnik, and W. T. Gowers. A Graphical User Interface Framework for Formal Verification. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:16, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-188-7. doi:10.4230/LIPIcs.ITP.2021.4. URL `https://drops.dagstuhl.de/opus/volltexte/2021/13899`.

Miriam Backens and Aleks Kissinger. ZH: A complete graphical calculus for quantum computations involving classical non-linearity. *Electronic Proceedings in Theoretical Computer Science*, 287:23–42, Jan 2019. ISSN 2075-2180. doi:10.4204/eptcs.287.2.

Frédéric Besson. Fast reflexive arithmetic tactics the linear case and beyond. In *Types for Proofs and Programs: International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, pages 48–62. Springer, 2007.

Jaap Boender, Florian Kammüller, and Rajagopal Nagarajan. Formalization of quantum protocols using coq. In Chris Heunen, Peter Selinger, and Jamie Vicary, editors, *Proceedings of the 12th International Workshop on Quantum Physics and Logic (QPL), Oxford, U.K., July 15–17, 2015*, volume 195 of *Electronic Proceedings in Theoretical Computer Science*, pages 71–83, Waterloo, NSW, Australia, November 2015. Open Publishing Association. doi:10.4204/EPTCS.195.6.

Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. String diagram rewrite theory ii: Rewriting with symmetric monoidal structure. *Mathematical Structures in Computer Science*, 32(4):511–541, 2022a.

Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. String diagram rewrite theory ii: Rewriting with symmetric monoidal structure, 2022b.

Jonathan Castello, Patrick Redmond, and Lindsey Kuper. Inductive diagrams for causal reasoning, 2023.

Nicholas Chancellor, Aleks Kissinger, Joschka Roffe, Stefan Zohren, and Dominic Horsman. Graphical structures for design and verification of quantum error correction, 2018.

Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoît Valiron. An automated deductive verification framework for circuit-building quantum programs. In Nobuko Yoshida, editor, *Programming Languages and Systems, ESOP 2021*, volume 12648 of *Lecture Notes in Computer Science*, pages 148–177, Cham, March 2021. Springer International Publishing. doi:10.1007/978-3-030-72019-3_6.

Alexandre Clément, Nicolas Heurtel, Shane Mansfield, Simon Perdrix, and Benoît Valiron. A complete equational theory for quantum circuits. In *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13. IEEE, 2023.

Alexandre Clément, Noé Delorme, and Simon Perdrix. Minimal equational theories for quantum circuits. In *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 1–14, 2024.

Bob Coecke. Quantum picturalism. *Contemporary physics*, 51(1):59–83, 2010.

Bob Coecke. Basic zx-calculus for students and professionals, 2023.

Bob Coecke and Ross Duncan. Interacting quantum observables: categorical algebra and diagrammatics. *New Journal of Physics*, 13(4):043016, apr 2011. doi:10.1088/1367-2630/13/4/043016.

Bob Coecke and Aleks Kissinger. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press, 2017. doi:10.1017/9781316219317.

Alexander Cowtan, Silas Dilkes, Ross Duncan, Will Simmons, and Seyon Sivarajah. Phase gadget synthesis for shallow circuits. *Electronic Proceedings in Theoretical Computer Science*, 318:213–228, may 2020. doi:10.4204/eptcs.318.13. URL `https://doi.org/10.4204%2Feptcs.318.13`.

Niel de Beaudrap and Dominic Horsman. The ZX calculus is a language for surface code lattice surgery. *Quantum*, 4:218, jan 2020. doi:10.22331/q-2020-01-09-218. URL `https://doi.org/10.22331%2Fq-2020-01-09-218`.

Niel de Beaudrap, Xiaoning Bian, and Quanlong Wang. Fast and effective techniques for t-count reduction via spider nest identities, 2020.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, pages 378–388. Springer, 2015.

David Elieser Deutsch. Quantum computational networks. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 425(1868):73–90, 1989.

Cirq Developers. Cirq, December 2022. URL `https://doi.org/10.5281/zenodo.7465577`. See full list of authors on Github: https://github.com/quantumlib/Cirq/graphs/contributors.

The Rocq LSP Developers. GitHub - ejgallego/rocq-lsp: Visual Studio Code Extension and Language Server Protocol for Rocq — github.com. `https://github.com/ejgallego/rocq-lsp`, 2023.

Ross Duncan and Simon Perdrix. Rewriting measurement-based quantum computations with generalised flow. In Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, editors, *Automata, Languages and Programming*, pages 285–296, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-14162-1.

Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, page 235–246, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581134878. doi:10.1145/581478.581501. URL https://doi.org/10.1145/581478.581501.

Benjamin Gregoire and Assia Mahboubi. Proving Equalities in a Commutative Ring Done Right in Coq. In Joe Hurd and Tom Melham, editors, *Lecture Notes in Computer Science*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113, Oxford, United Kingdom, August 2005. Springer. doi:10.1007/11541868_7. URL https://inria.hal.science/hal-00819484.

Amar Hadzihasanovic. The algebra of entanglement and the geometry of composition, 2017.

Michael Hedberg. A coherence theorem for martin-löf's type theory. *Journal of Functional Programming*, 8(4):413–436, 1998. doi:10.1017/S0956796898003153.

Kesha Hietala, Robert Rand, Shih-Han Hung, Liyi Li, and Michael Hicks. Proving Quantum Programs Correct. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:19, Dagstuhl, Germany, 2021a. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-188-7. doi:10.4230/LIPIcs.ITP.2021.21.

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. A verified optimizer for quantum circuits. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021b. doi:10.1145/3434318.

INQWIRE Developers. INQWIRE QuantumLib, 1 2022. URL https://github.com/inQWIRE/QuantumLib.

Emmanuel Jeandel, Simon Perdrix, and Renaud Vilmart. A complete axiomatisation of the zx-calculus for clifford+t quantum mechanics, 2018a.

Emmanuel Jeandel, Simon Perdrix, and Renaud Vilmart. Diagrammatic reasoning beyond clifford+ t quantum mechanics. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 569–578, 2018b.

Emmanuel Jeandel, Simon Perdrix, and Renaud Vilmart. Completeness of the ZX-Calculus. *Logical Methods in Computer Science*, 6 2020. doi:10.23638/LMCS-16(2:11)2020.

Emmanuel Jeandel, Simon Perdrix, and Margarita Veshchezerova. Addition and differentiation of zx-diagrams. *Logical Methods in Computer Science*, Volume 20, Issue 2, 5 2024. ISSN 1860-5974. doi:10.46298/lmcs-20(2:10)2024. URL http://dx.doi.org/10.46298/lmcs-20(2:10)2024.

Siekmann Jörg, Hess Stephan, Benzmüller Christoph, Cheikhrouhou Lassaad, Fiedler Armin, Horacek Helmut, Kohlhase Michael, Konrad Karsten, Meier Andreas, Melis Erica, et al. Loui: Lovely omega user interface. 1999.

André Joyal and Ross Street. Braided tensor categories. *Advances in Mathematics*, 102(1): 20–78, 1993.

Aleks Kissinger. *Abstract Tensor Systems as Monoidal Categories*, pages 235–252. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-642-54789-8. doi:10.1007/978-3-642-54789-8_13. URL https://doi.org/10.1007/978-3-642-54789-8_13.

Aleks Kissinger and John van de Wetering. PyZX: Large Scale Automated Diagrammatic Reasoning. In Bob Coecke and Matthew Leifer, editors, Proceedings 16th International Conference on *Quantum Physics and Logic,* Chapman University, Orange, CA, USA., 10-14 June 2019, volume 318 of *Electronic Proceedings in Theoretical Computer Science*, pages 229–241. Open Publishing Association, 2020. doi:10.4204/EPTCS.318.14.

Aleks Kissinger and John van de Wetering. Simulating quantum circuits with zx-calculus reduced stabiliser decompositions. *Quantum Science and Technology*, 7(4):044001, 2022.

Aleks Kissinger and Vladimir Zamdzhiev. Quantomatic: A proof assistant for diagrammatic reasoning. In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, pages 326–336. Springer, 2015.

Alex Kissinger. GitHub - akissinger/chyp: An interactive theorem prover for string diagrams — github.com. https://github.com/akissinger/chyp, 2023.

Robbert Krebbers and Ralf Jung. Rocq-std++. URL https://gitlab.mpi-sws.org/iris/stdpp.

Ambroise Lafont. A diagram editor to mechanize categorical proofs. https://amblafont.github.io/articles/yade.pdf, 2023.

Adrian Lehmann, Ben Caldwell, Bhakti Shah, William Spencer, and Robert Rand. Vyzx: Formal verification of a graphical quantum language, 2026. URL https://arxiv.org/abs/2311.11571.

Liyi Li, Finn Voichick, Kesha Hietala, Yuxiang Peng, Xiaodi Wu, and Michael Hicks. Verified compilation of quantum oracles. *Proceedings of the ACM on Programming Languages*, 6 (OOPSLA2):146, October 2022. doi:10.1145/3563309. URL https://github.com/inQWIRE/VQO.

Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Naijun Zhan. Quantum hoare logic. *Archive of Formal Proofs*, March 2019. ISSN 2150-914x. https://isa-afp.org/entries/QHLProver.html, Formal proof development.

The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, jan 2020. doi:10.1145/3372885.3373824. URL https://doi.org/10.1145%2F3372885.3373824.

Tommy McElvanney and Miriam Backens. Flow-preserving zx-calculus rewrite rules for optimisation and obfuscation, 2023.

Yunseong Nam, Neil J Ross, Yuan Su, Andrew M Childs, and Dmitri Maslov. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information*, 4(1):1–12, 2018. doi:10.1038/s41534-018-0072-4.

Wojciech Nawrocki, Edward W. Ayers, and Gabriel Ebner. An Extensible User Interface for Lean 4. In *To appear, 14th International Conference on Interactive Theorem Proving (ITP 2023)*, Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:https://doi.org/10.4230/LIPIcs.ITP.2023.8. URL https://voidma.in/assets/papers/23nawrocki_extensible_user_interface_lean_4.pdf.

Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, Cambridge, December 2010. doi:10.1017/CBO9780511976667.

Jennifer Paykin, Robert Rand, and Steve Zdancewic. QWIRE: A core language for quantum circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, pages 846–858, New York, NY, USA, January 2017. Association for Computing Machinery. doi:10.1145/3009837.3009894. URL https://jpaykin.github.io/papers/prz_qwire_2017.pdf.

Yuxiang Peng, Kesha Hietala, Runzhou Tao, Liyi Li, Robert Rand, Michael Hicks, and Xiaodi Wu. A formally certified end-to-end implementation of shor's factorization algorithm. *Proceedings of the National Academy of Sciences*, 120(21):e2218775120, 2023.

Damien Pous. String diagrams for monoidal categories, in rocq. *arXiv preprint arXiv:2602.19806*, 2026.

Qiskit contributors. Qiskit: An open-source framework for quantum computing, 2023.

Robert Rand, Jennifer Paykin, and Steve Zdancewic. QWIRE practice: Formal verification of quantum circuits in coq. In Bob Coecke and Aleks Kissinger, editors, *Proceedings of the 14th International Conference on Quantum Physics and Logic (QPL), Nijmegen, the Netherlands, July 3–7, 2017*, volume 266 of *Electronic Proceedings in Theoretical Computer Science*, pages 119–132, Waterloo, NSW, Australia, February 2018. Open Publishing Association. doi:10.4204/EPTCS.266.8.

Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. QED at large: A survey of engineering of formally verified software. *CoRR*, abs/2003.06458, 2020. URL https://arxiv.org/abs/2003.06458.

The Rocq Development Team. The rocq prover, Jan 2025. Available electronically at rocq-prover.org.

P. Selinger. A survey of graphical languages for monoidal categories. In *New Structures for Physics*, pages 289–355. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-12821-9_4.

Bhakti Shah, Willam Spencer, Laura Zielinski, Ben Caldwell, Adrian Lehmann, and Robert Rand. ViCAR: Visualizing Categories with Automated Rewriting in Coq. In Michael Johnson and David Jaz Myers, editors, Proceedings Seventh International Conference on *Applied Category Theory 2024,* Oxford, United Kingdom, 17 - 21 June 2024, volume 429 of *Electronic Proceedings in Theoretical Computer Science*, pages 234–248. Open Publishing Association, 2025. doi:10.4204/EPTCS.429.13.

Matthieu Sozeau. Generalized rewriting, Jun 2023. URL https://github.com/coq/coq/blob/58ac2d8dd403fa7a96429fc1f839225d83be9566/doc/sphinx/addendum/generalized-rewriting.rst.

John van de Wetering. Zx-calculus for the working quantum computer scientist, 2020.

Li Zhou, Gilles Barthe, Pierre-Yves Strub, Junyi Liu, and Mingsheng Ying. Coqq: Foundational verification of quantum programs. *Proceedings of the ACM on Programming Languages*, 7(POPL):29, January 2023. doi:10.1145/3571222. URL https://github.com/coq-quantum/CoqQ.