THE UNIVERSITY OF CHICAGO

PROOF VISUALIZATION FOR GRAPHICAL LANGUAGES

A THESIS SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN PARTIAL FULFILMENT OF THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY

BHAKTI SHAH

CHICAGO, ILLINOIS

JUNE 2024

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This work would not have been possible without a lot of people. I would like to thank my advisor, Robert Rand, for constant support and motivation. Adrian Lehmann and Ben Caldwell – without whom *Vy*ZX would not exist – for their mentoring, graduate school advice, LaTeX hacks, and category theory knowledge. William Spencer and Laura Zielinski, for their work on ViCAR, and inputs on design decisions. Emilio Jesús Gallego Arias, for his invaluable help with `coq-lsp` integration. To Ravi Chugh, for introducing me to the world of programming languages. To my friends Stella, Derek, David, and Lorenzo, and my cat Qubit, for their constant support. Finally, to my sister and my parents, without whom I would not be here today.

ABSTRACT

Reasoning about graphical languages in a proof assistant can be hard. Canonical diagrammatic representations are optimized for readability, and may abstract away details irrelevant to a pen-and-paper proof, but these details are often important to a proof assistant. We develop a methodology for working with graphical languages associated with classes of categories, and equip it with an automated visualizer integrated with the Coq proof assistant, enabling diagrammatic reasoning. We instantiate this with the ZX-calculus, a language for quantum computation, as an example of specialized diagrammatic reasoning.

CHAPTER 1

INTRODUCTION

The question driving this thesis is: **How do we reason about graphical languages diagrammatically in an interactive proof assistant?**. Let us look at the graphical language of process diagrams associated with the semantics of process theories, adapted from Coecke and Kissinger [3], as a motivating example.

A *process* refers to anything that has some non-negative number of inputs and outputs. We represent a process as a *box* with *wires* representing the input and output systems. We label wires with *types*. As an example, consider the simple function operating on integers.

$$f(x, y) = x + y$$

This can be viewed as a process with two integer inputs and one integer output, and a diagram of this process can be seen in Figure 1.1.

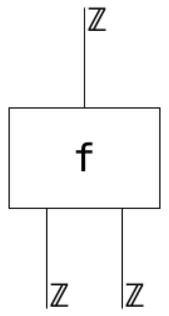
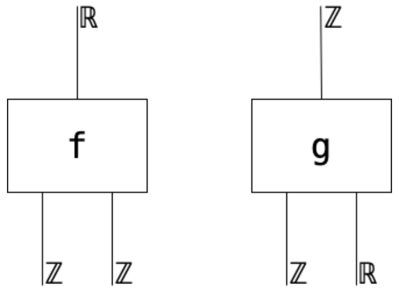


Figure 1.1: The process $f(x, y) = x + y$

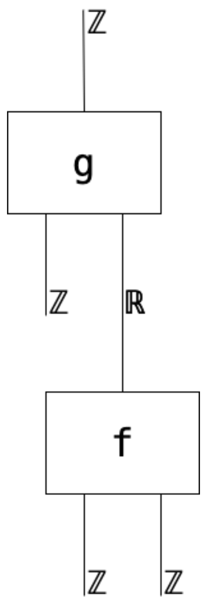
We can compose simple processes to form more complex processes, provided the required input and output types match. For example, if we have the two processes in Figure 1.2a, then they can be combined as in Figure 1.2b, but not Figure 1.2c.

Definition 1 (Process theory)**.** *A process theory consists of:*

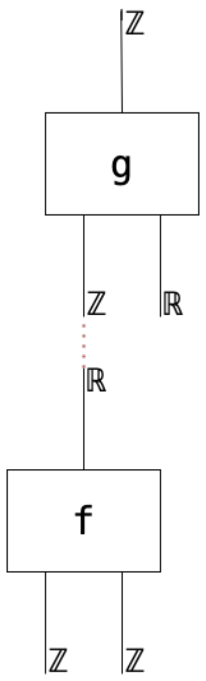
1. *a collection T of system-types represented by wires,*



(a) Two example processes.



(b) Valid composition of processes.



(c) Invalid composition of processes.

Figure 1.2: Composing processes.

2. *a collection P of processes represented by boxes, where for each process in P the input types and output types are taken from T,*

3. *a means of "wiring processes together", that is, an operation that interprets a diagram of processes in P as a process in P.*

For the purpose of this example, we restrict ourselves to the process theory of *functions*, where the system types are sets. Our processes are just functions in mathematics, as seen in Figure 1.1. We can *sequentially compose* functions the usual way, using the $\circ$ operator. In the process theory, since function composition is the action corresponding to process composition. This is intuitively imposing an ordering on the occurrence of processes: $A \circ B$ means B occurs, and then A occurs. We can also compose processes *in parallel*, essentially forming a pair of processes, using the $\otimes$ operator. Parallel composition is represented using the $\otimes$ operator, and diagrammatically, it corresponds to placing two processes next to one another, as in

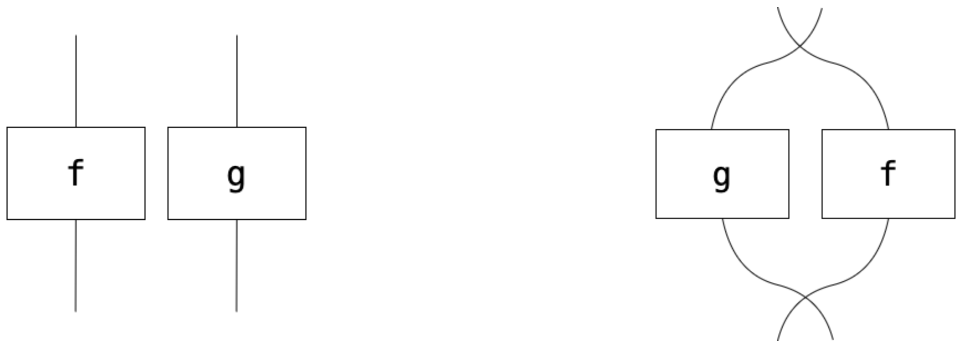
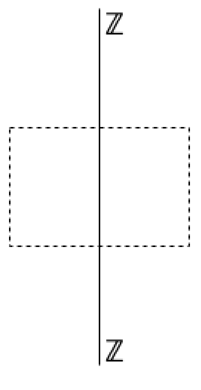


Figure 1.3: Two equivalent processes.

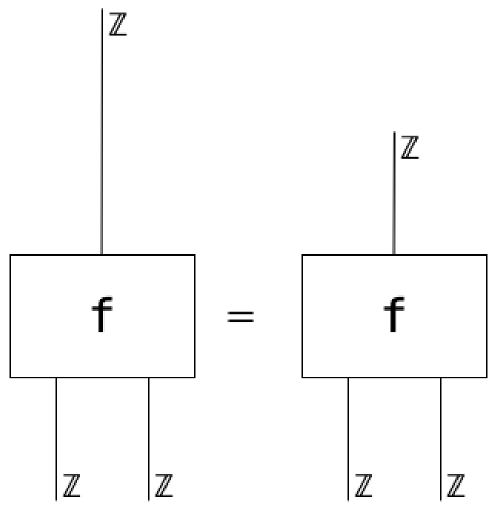
Figure 2.9. Intuitively, this means executing two processes at the same time.

Process diagrams can be *deformed* without semantically changing, provided the order of inputs and outputs remains consistent. For example, the diagrams in Figure 2.9 are equivalent.

Standalone wires can be seen as boxes that "do nothing": Figure 1.4a corresponds to the identity function on integers. Attaching a wire to a process has no effect, as in Figure 1.4b: it is equivalent to `id` $\circ f$.



(a) A wire is a process.



(b) A wire serves as the identity.

Figure 1.4: Wires.

We introduce process theories so we can propose the following:

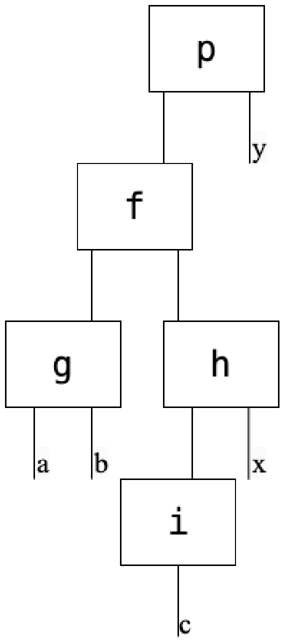


Figure 1.5: The process $p(f(g(a,b),h(i(c),x)),y)$

- **The simplicity of diagrams is not easy to translate into a purely textual form.** Consider the process $p(f(g(a,b),h(i(c),x)),y)$. This is a nightmare to parse, especially if one is trying to understand its structure. On the other hand, the process diagram in Figure 1.5 is also a self contained definition, and is much easier to look at. One can immediately tell the relation between nodes as inputs and outputs for one another.

- **A diagram may itself be a proof.** Consider the statements $(g1 \otimes g2) \circ (f1 \otimes f2)$ and $(g1 \circ f1) \otimes (g2 \circ f2)$. We can prove that these are equal algebraically using properties of the operators that we have not introduced yet, but diagrammatically, the two are identical: Figure 1.6 is the diagram for both statements. Thus, this diagram alone is a proof of equality of two statements.

Now, we add in the additional constraint of working in a proof assistant. Specifically, we consider the dependently typed proof assistant, Coq. Typically, this class of provers is most powerful when used to reason about structure, and, with Coq being an interactive proof

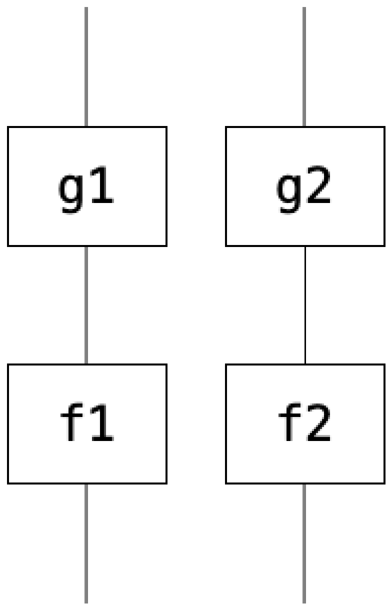


Figure 1.6: The diagrammatic representation of both $(g1 \otimes g2) \circ (f1 \otimes f2)$ and $(g1 \circ f1) \otimes (g2 \circ f2)$

assistant, we must reason about structure *interactively.* The main consequence of this is having to explicitly deal with several intermediate states. Further, as we are working in a formal system, steps that may be elided in pen-and-paper proofs must be explicitly addressed.

- **We must reason about several intermediate states.** Consider identity wires as seen before. Figure 1.4b shows how adding identity wires does not change the diagram semantically. However, the two are structurally different, and proving them equivalent would involve an explicit rewrite. Two diagrams that are structurally different but semantically identical can only be unified via a rewrite, and this must be done for every individual structural difference. A proof assistant requires us to reason about each one of these intermediate states, rather than the entire solution itself. One can see how constructing such diagrams by hand can get cumbersome, especially with more complex graphical structures, and we have already established that a textual representation is not easy to work with.

- **Canonical diagrammatic representations abstract over structural details.** Process diagrams are designed to abstract over information that does not alter the

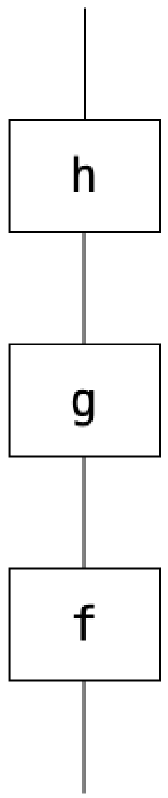


Figure 1.7: $h \circ g \circ f$

semantics of the diagram: for example, adding identity wires. As another example, consider Figure 5.3. While sequential composition is associative, this is not something the proof assistant knows. $h \circ (g \circ f)$ is structurally different from $(h \circ g) \circ f$, but our diagrammatic syntax renders both in the same way. This is a problem: we want to distinguish between these two states, as the proof assistant would require an explicit proof of equivalence to transform one term to the other. We must come up with an alternative diagrammatic notation such that two terms that differ structurally are rendered distinctly.

If we wish to work with graphical structures in a proof assistant, we must do so diagrammatically, but not using the canonical diagrammatic representation; we must reason about several intermediate states, and doing so by hand is awkward, so an automated translation from a textual term to a visual representation is ideal.

This thesis aims to solve this problem for a subset of graphical structures, ones that have a semantics corresponding to a class of categories. Process diagrams are an example of such a graphical structure, and hence we share the same motivations. Chapter 2 elaborates on this further, Chapter 3 and Chapter 4 introduce the semantics of the systems we work in, and Chapter 5 describes our approach to this problem. Finally, we go over related work and future directions in Chapter 6 and Chapter 7 respectively.

CHAPTER 2
BACKGROUND

2.1 The Coq proof assistant

Formal verification aims to prove properties of a computer program, using a mathematical specification. Proof assistants are software that allow for a computer aided process of formal verification. Dependently typed proof assistants utilize the Curry-Howard-Lambek isomorphism [5, 10, 15] to essentially allow the user to construct programs as proofs.

Coq [4] is a widely used dependently typed proof assistant, which uses *tactics* to rewrite proof terms. Rather than explicitly constructing proof terms, the user must apply rewrite rules to transform the proof obligation, until it is reduced to something that can be proved trivially.

2.1.1 coq-lsp

`coq-lsp` [6] is a language server for the Coq proof assistant, along with a client side extension implementation. The server itself extends the language server protocol with functionality specific to proof assistants, allowing for easier integration of external IDE tools. `coq-lsp` is also designed with incrementality in mind, and computations are scheduled to ensure maximum efficiency.

2.2 Category Theory

Category theory is a powerful, unifying framework in mathematics and beyond, allowing for the simplification of complex systems via identification of common structural properties. We iteratively define the classes of categories up to the symmetric monoidal level. We also connect these definitions back to process theories, as we construct string diagrams using elements corresponding to categorical constructs.

The following definitions are adapted from Mac Lane [17], Joyal and Street [12], Selinger [21], and Pierce [20].

Definition 2 (Category)**.** *A* ***category C*** *comprises:*

1. *A collection of* ***objects;***
2. *A collection of* ***arrows*** *(or* ***morphisms****);*
3. *Operations assigning to each arrow f an object $dom f$, its* ***domain****, and an object $cod f$, its* ***codomain****, writing $f : A \to B$ when $dom f = A$ and $cod f = B$;*
4. *A composition operator assigning to each pair of arrows f and g such that $cod f = dom g$, a* ***composite*** *arrow $g \circ f : dom f \to cod g$, satisfying the following associative law:*

$$\forall\ f : A \to B,\ g : B \to C,\ h : C \to D,$$
$$h \circ (g \circ f) = (h \circ g) \circ f;$$

5. *for each object A, an* ***identity*** *arrow $id_A : A \to A$ satisfying the following identity law:*

$$\forall f : A \to B,$$
$$id_B \circ f = f \quad \text{and} \quad f \circ id_A = f.$$

A vanilla category corresponds to a basic process theory, an object is a wire, a morphism is a box, an identity morphism is the identity wire, and morphism composition corresponds to process composition. The diagrammatic forms of each construct can be seen in Figure 2.1.

Definition 3 (Monoidal Category)**.** *A category* ***C*** *is* ***monoidal*** *if it consists of:*

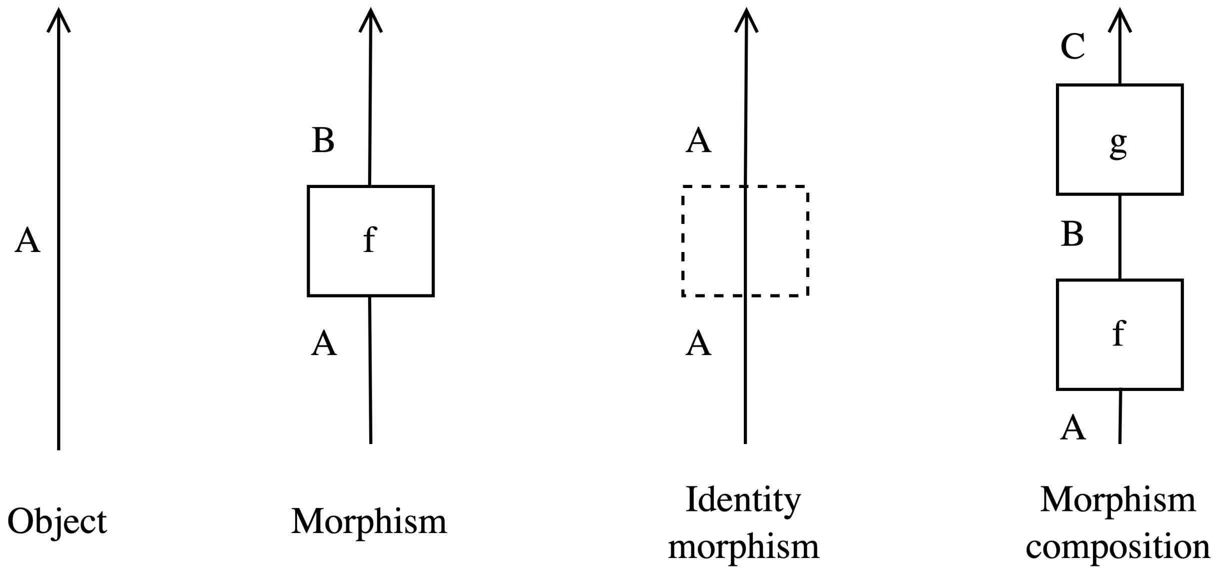


Figure 2.1: The process theory associated with categories.

1. A bifunctor $\otimes : C \times C \to C$, which means:

$$id_A \otimes id_B = id_{A \otimes B}$$
$$(f' \otimes g') \circ (f \otimes g) = (f' \circ f) \otimes (g' \circ g)$$

2. An object $e \in C$ called the unit object,

3. Natural isomorphisms:

$$\alpha = \alpha_{A,B,C} : (A \otimes B) \otimes C \simeq A \otimes (B \otimes C)$$
$$\lambda = \lambda_A : I \otimes A \simeq A$$
$$\rho = \rho_A : A \otimes I \simeq A$$

such that $\forall\ A, B, C, D \in C$, the diagrams in Figure 2.2 commute.

A monoidal category adds the diagrammatic constructs in Figure 2.3 to Figure 2.1.

$$\begin{tikzcd}
& (A \otimes (B \otimes C)) \otimes D \arrow[rr, "\alpha_{A,B\otimes C,D}"] & & A \otimes ((B \otimes C) \otimes D) \arrow[dr, "id_A \otimes \alpha_{B,C,D}"] & \\
((A \otimes B) \otimes C) \otimes D \arrow[ur, "\alpha_{A,B,C} \otimes id_D"] \arrow[drr, "\alpha_{A\otimes B,C,D}"'] & & & & A \otimes (B \otimes (C \otimes D)) \\
& & (A \otimes B) \otimes (C \otimes D) \arrow[urr, "\alpha_{A,B,C\otimes D}"'] & &
\end{tikzcd}$$

$$\begin{tikzcd}
(A \otimes I) \otimes B \arrow[rr, "\alpha_{A,I,B}"] \arrow[dr, "\rho_A \otimes id_B"'] & & A \otimes (I \otimes B) \arrow[dl, "id_A \otimes \lambda_B"] \\
& A \otimes B &
\end{tikzcd}$$

Figure 2.2: The pentagon and triangle coherence axioms.

Definition 4 (Braiding)**.** *A* ***braiding*** *for a monoidal category* ***C*** *consists of a natural family of isomorphisms*

$$c = c_{A,B} : A \otimes B \simeq B \otimes A$$

such that $\forall\ A, B, C \in C$*, the diagrams in Figure 2.4 commute.*

Definition 5 (Braided Monoidal Category)**.** *A* ***braided monoidal category*** *is a pair* (C, c) *consisting of a monoidal category* ***C*** *and a braiding* ***c****.*

Definition 6 (Symmetric Monoidal Category)**.** *A* ***symmetric monoidal category*** *is a braided monoidal category with a self-inverse braiding, that is*

$$c_{A,B} = c^{-1}_{B,A}.$$

A symmetric monoidal category adds the diagrammatic constructs in Figure 2.5 to Figure 2.3.

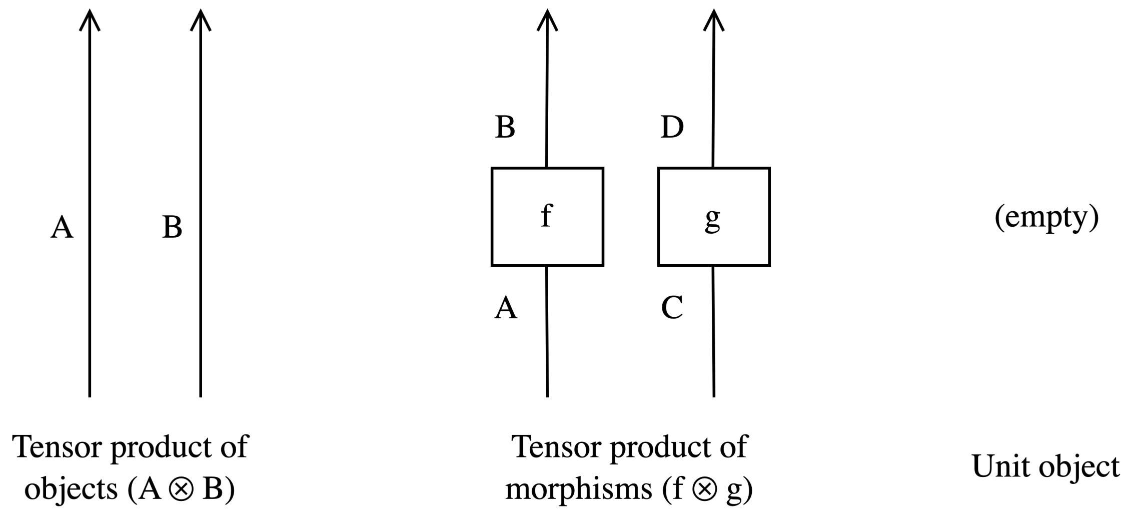

A
B
B
f
A
D
g
C
(empty)
Tensor product of objects (A ⊗ B)
Tensor product of morphisms (f ⊗ g)
Unit object

Figure 2.3: Additional diagrammatic constructs for monoidal categories as processes.

Definition 7 (Dagger Category). *A **dagger category** is a category **C** such that for every morphism $f : A \to B$ there is an associated morphism $f^\dagger : B \to A$, known as the adjoint of f, such that $\forall f : A \to B$, $g : B \to C$:*

$$id_A^\dagger = id_A : A \to A,$$
$$(g \circ f)^\dagger = f^\dagger \circ g^\dagger : C \to A,$$
$$f^{\dagger\dagger} = f : A \to B.$$

A dagger category simply represents the adjoint of a diagram by annotating it with an adjoint symbol (†).

Definition 8 (Exact Pairing). *In a monoidal category, an exact pairing between two objects A and B is given by a pair of morphisms $\eta : I \to B \otimes A$ and $\epsilon : A \otimes B \to I$, such that the adjunction triangles in Figure 2.6 commute. η and ϵ are known as the unit and the counit of the adjunction respectively. In such an exact pairing, B is called the right dual of A and A is called the left dual of B.*

Definition 9 (Left Autonomous Category). *A monoidal category is left autonomous if every*

$$\begin{tikzcd}
 & (B \otimes A) \otimes C \arrow[r, "\alpha_{B,A,C}"] & B \otimes (A \otimes C) \arrow[dr, "id_B \otimes c_{A,C}"] & \\
(A \otimes B) \otimes C \arrow[ur, "c_{A,B} \otimes id_C"] \arrow[dr, "\alpha_{A,B,C}"'] & & & B \otimes (C \otimes A) \\
 & A \otimes (B \otimes C) \arrow[r, "c_{A,B\otimes C}"'] & (B \otimes C) \otimes A \arrow[ur, "\alpha_{B,C,A}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
 & (B \otimes A) \otimes C \arrow[r, "\alpha_{B,A,C}"] & B \otimes (A \otimes C) \arrow[dr, "id_B \otimes c^{-1}_{C,A}"] & \\
(A \otimes B) \otimes C \arrow[ur, "c^{-1}_{B,A} \otimes id_C"] \arrow[dr, "\alpha_{A,B,C}"'] & & & B \otimes (C \otimes A) \\
 & A \otimes (B \otimes C) \arrow[r, "c^{-1}_{B\otimes C,A}"'] & (B \otimes C) \otimes A \arrow[ur, "\alpha_{B,C,A}"'] &
\end{tikzcd}$$

Figure 2.4: The hexagon coherence axioms.

object has a left dual.

Definition 10 (Right Autonomous Category)**.** *A monoidal category is right autonomous if every object has a right dual.*

Definition 11 (Autonomous Category)**.** *A monoidal category is autonomous if it is both left and right autonomous.*

Autonomous categories add the diagrammatic constructs in Figure 2.7 to Figure 2.5.

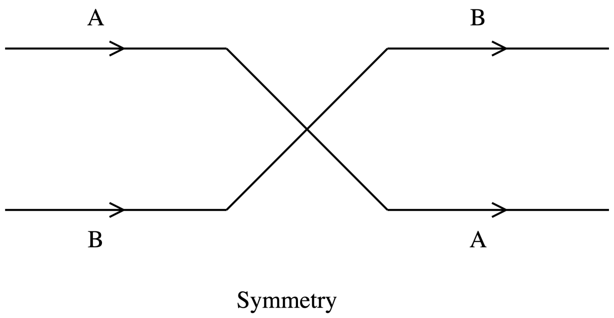


Figure 2.5: Additional diagrammatic construct for symmetric monoidal categories as processes.

$$\begin{tikzcd}
A \arrow[r, "id_A \otimes \nu"] \arrow[dr, "id_A"'] & A \otimes B \otimes A \arrow[d, "\epsilon \otimes id_A"] & B \arrow[r, "\nu \otimes id_B"] \arrow[dr, "id_B"'] & B \otimes A \otimes B \arrow[d, "id_B \otimes \epsilon"] \\
& A & & B
\end{tikzcd}$$

Figure 2.6: Adjunction triangles for exact pairings.

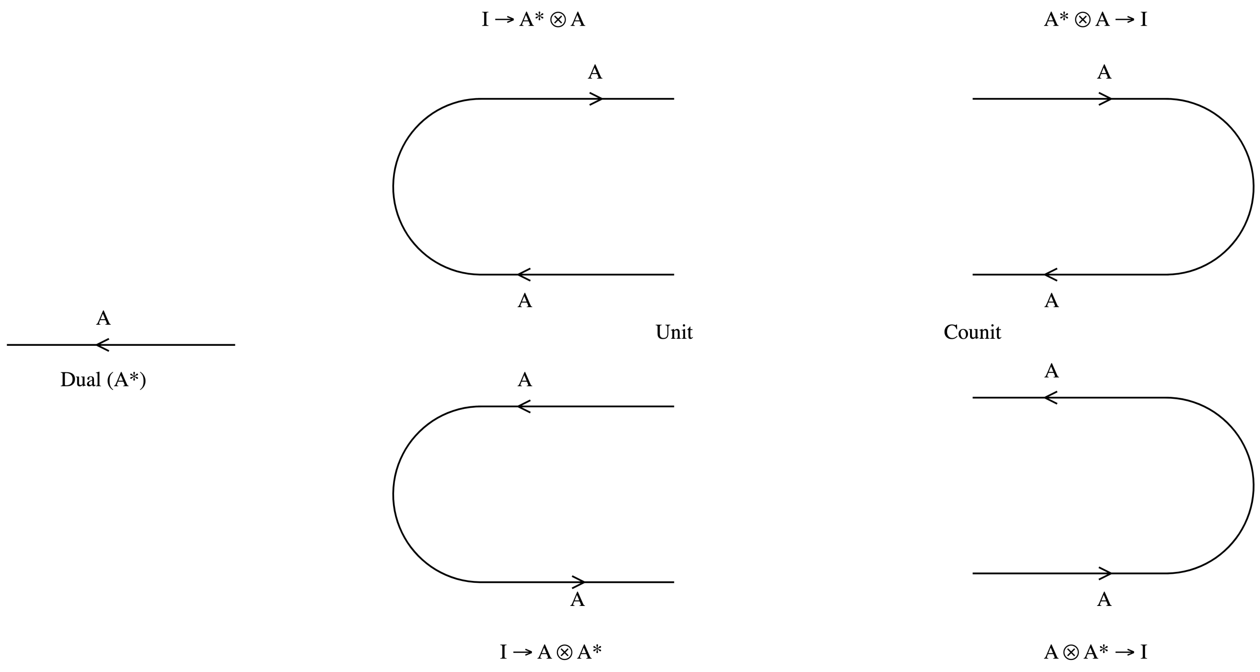


Figure 2.7: Additional diagrammatic constructs for autonomous categories as processes.

Definition 12 (Compact Closed Category)**.** *A compact closed category is a right autonomous symmetric monoidal category.*

Compact closed categories simply add symmetry to the diagrammatic constructs in Figure 2.7.

2.3 ZX-calculus

The ZX-calculus [2] is a complete set of rewrite rules for the manipulation of ZX-diagrams, a graphical representation of quantum operations. ZX-diagrams are graphs that consist of green and red nodes, known as Z and X spiders respectively, with n inputs and m outputs, along with a rotation angle $\alpha \in [0, 2\pi)$. Figure 2.8 is the diagrammatic representation of these spiders.

We briefly present the features and rules making up the ZX-calculus, drawing heavily from van de Wetering [25].

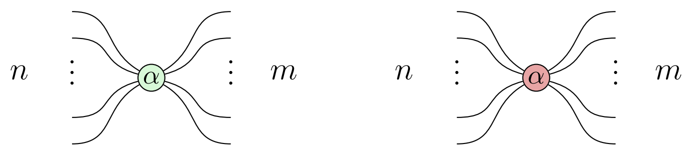
2.3.1 Meta-Rules

Colorswapping A color-swapped ZX-diagram is exactly what it sounds like: structurally identical, but replacing every instance of a Z spider with an X spider and vice versa. Any rules transforming zx_1 to zx_2 can also be applied to the color-swapped zx_1, transforming it to a color-swapped zx_2. Thus, any rules presented apply for their color-swapped versions too.

Only connectivity matters An important meta-rule in the ZX-calculus is *only connectivity matters (OCM)*. Arbitrary wire transforms do not alter the diagram's meaning, if the overall input and output orders are maintained.This rule is very powerful as it allows us to move diagrams into the most convenient form to apply rules to. We show an example of OCM deformations in Figure 2.9.

2.3.2 Rewrite Rules

An important ZX-calculus rule is the *spider fusion* rule, which allows spiders connected by a non-zero number of edges to be *fused* into a single spider, with the rotation angle of the



(a) Z-spider (b) X-spider

Figure 2.8: Canonical ZX spiders

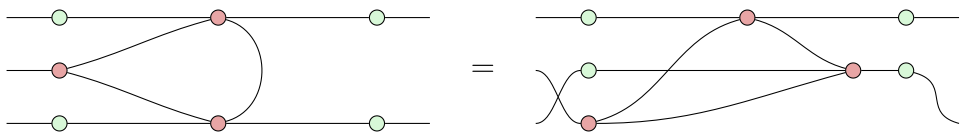
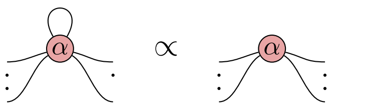
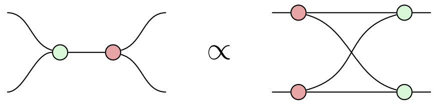


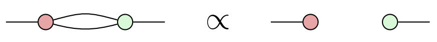
Figure 2.9: Only connectivity matters: Two equal ZX-diagrams, where only connections and qubit order are maintained.



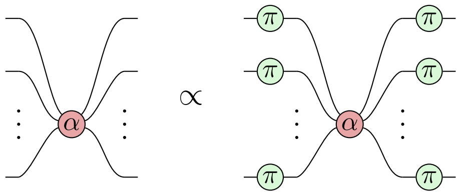
(a) The self-loop removal rule.



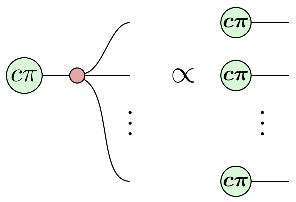
(b) The bialgebra rule.



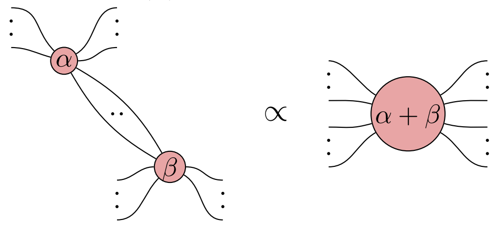
(c) The Hopf rule.



(d) The bi-π rule.



(e) The state copy rule.



(f) The spider fusion rule.

Figure 2.10: Rules of the ZX-calculus, where $\alpha, \beta \in \mathbb{R}, c \in \mathbb{N}$

fused spider being equal to the sum of the rotation angles of the original spiders. This rule is shown in Figure 2.10f. The reverse direction is also true: any spider can be split into two spiders, provided the new spiders have rotation angles that sum up to original angle. With the spider fusion rule, we can manipulate the number of inputs and outputs for all other rules by adding a fuseable node to said input or output.

Another simplification is that we can remove self-loops as shown in Figure 2.10a [25]. Intuitively, this rule says that a node cannot provide more information about itself.

The bialgebra rule (Figure 2.10b), while not intuitive, is crucial for many ZX proofs, as it allows for the rearranging of edges between nodes and the introduction or removal of swaps.

The Hopf rule, shown in Figure 2.10c, allows us to disconnect specific nodes in the diagram instead of changing their connection.

For any X spider, it is equivalent to itself with Z spiders on each edge, where each Z spider has a π rotational angle. Figure 2.10d shows this rule.

The state copy rule (Figure 2.10e) allows an X spider to copy arity-1 Z spiders.

2.3.3 Representing Unitary Gates

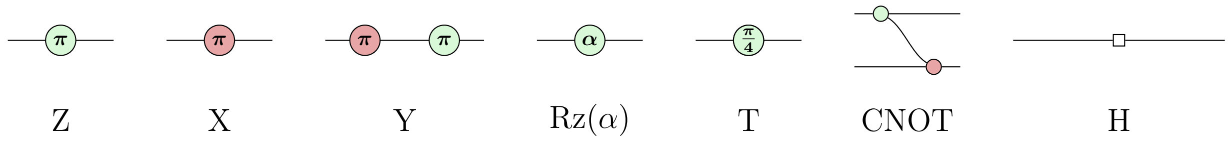


Figure 2.11: Common gates represented in the ZX-calculus.

We translate some standard unitary gates from the quantum circuit model into ZX-calculus in Figure 2.11. Z, T, and 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. H switches between the Z and X bases; following van de Wetering [25] 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. Note that OCM counterintuitively tells us that we can slide the green "controlling node" past the red "NOT" without changing the semantics of the diagram.

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 circuit model's rigid structure. Note that the gates shown in Figure 2.11 form the RzQ gate set [18], a complete gate set for quantum computing.

The ZX-calculus has semantics rooted in complex matrices, but its purpose is to facilitate diagrammatic reasoning – as such, once one is convinced the semantics are sound and complete, the ZX-calculus can be independently used to reason about quantum operations, without referring to the linear algebraic interpretation at all. For this reason, we do not go into the denotational semantics of the calculus at all, and instead focus on the diagrammatic language itself.

CHAPTER 3

VYZX

*Vy*ZX [16] is a project to **V**erif**y** the **ZX**-calculus, in the proof assistant Coq. *Vy*ZX has the expected denotational semantics attributed to symmetric monoidal categories using matrices, but these exist only for verification purposes – reasoning about terms in *Vy*ZX can be done purely using the diagrammatic rewrite rules of the ZX-calculus.

Eliding the technical details of *Vy*ZX, we present the concrete syntax of the library, as well as the major design choices that are relevant to the visualization, in Figure 3.1. We use a dependent type to represent a ZX object, parameterized by its input and output arity. We define Z and X spiders in the expected manner, and also define explicit constructors for `Cap`, `Cup`. `Wire`, `Box`, `Swap`, and `Empty` for convenience, though we note that they can be defined in terms of Z and X spiders. We define sequential and parallel composition, `Stack` and `Compose`, which enforce semantic well-formedness via the dependently typed structure ensuring input and output arity constraints are met. These form *parametric morphisms* [23], allowing us to define the equivalence relation for proportionality:

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

while ensuring safe rewrites. We need a notion of proportionality to reason about semantic equality, as two semantically equivalent terms may not be structurally equal, and equality only reflects structural equality.

$$\frac{\text{in out} : \mathbb{N} \qquad \alpha : \mathbb{R}}{\texttt{Z}\ \text{in out}\ \alpha : \texttt{ZX}\ \text{in out}} \qquad \frac{}{\texttt{Cap} : \texttt{ZX}\ 0\ 2} \qquad \frac{}{\texttt{Cup} : \texttt{ZX}\ 2\ 0} \qquad \frac{\text{in out} : \mathbb{N} \qquad \alpha : \mathbb{R}}{\texttt{X}\ \text{in out}\ \alpha : \texttt{ZX}\ \text{in out}}$$

$$\frac{}{\texttt{Wire} : \texttt{ZX}\ 1\ 1} \qquad \frac{}{\texttt{Box} : \texttt{ZX}\ 1\ 1} \qquad \frac{}{\texttt{Swap} : \texttt{ZX}\ 2\ 2} \qquad \frac{}{\texttt{Empty} : \texttt{ZX}\ 0\ 0}$$

$$\frac{\texttt{zx}_0 : \texttt{ZX}\ \text{in mid} \qquad \texttt{zx}_1 : \texttt{ZX}\ \text{mid out}}{\texttt{Compose}\ \texttt{zx}_0\ \texttt{zx}_1 : \texttt{ZX}\ \text{in out}} \qquad \frac{\texttt{zx}_0 : \texttt{ZX}\ \text{in}_0\ \text{out}_0 \qquad \texttt{zx}_1 : \texttt{ZX}\ \text{in}_1\ \text{out}_1}{\texttt{Stack}\ \texttt{zx}_0\ \texttt{zx}_1 : \texttt{ZX}\ (\text{in}_0 + \text{in}_1)\ (\text{out}_0 + \text{out}_1)}$$

Figure 3.1: Inductive constructors for ZX-diagrams in *Vy*ZX

The ZX-calculus forms a dagger compact category (symmetric monoidal + autonomous + dagger category). We map categorical concepts to inductive constructors in *Vy*ZX in Table 3.1.

Categorical Concept	Inductive Constructor	Symbol
id_A	Wire	$-$
I	Empty	$\emptyset$
$\circ$	Compose	$\leftrightarrow$
$\otimes$	Stack	$\updownarrow$
Symmetric braid	Swap 1 1	$\times$
Unit	Cap	$\subset$
Co-unit	Cup	$\supset$

Table 3.1: Translating categorical concepts to inductive constructors with their respective symbolic notation.

CHAPTER 4
VICAR

ViCAR [22] is a framework for reasoning about (monoidal) categories in Coq. ViCAR uses a nested hierarchy of typeclasses to define several classes of categories, which can be instantiated by user-defined types. ViCAR aims to use category theory as a generalization technique in formal verification, allowing for the identification of common structural patterns within conceptually diverse topics, and enabling reuse between different instantiations with the same structure. It also provides a set of expressive automated tactics, allowing for repetitive but mundane patterns to be solved trivially.

ViCAR aims to build up typeclasses for the categories seen in Figure 4.1. Objects of a category have the type of the category itself, and morphisms are parameterized by their object arguments. ViCAR takes advantage of Coq's type inference mechanism [24], allowing for automated unification. For example, suppose an instance `catC` of our category typeclass has been declared whose objects have type `C`. Then, if `A`, `B` and `M` are terms of type `C` and `f` and `g` have types `A` $\to$ `B` and `B` $\to$ `M`, Coq would be able to automatically infer the type of `f` $\circ$ `g` without the user having to explicitly point to the particular instance. ViCAR also separates structure from coherence. Categorical structures are specified by typeclasses with only structural properties, and coherence conditions exist in a separate typeclass. This allows users to use the visualizer without formally specifying coherence conditions, which is a harder task. ViCAR supplies a number of automation tactics, including but not limited to foliation, simplification, and rewriting.

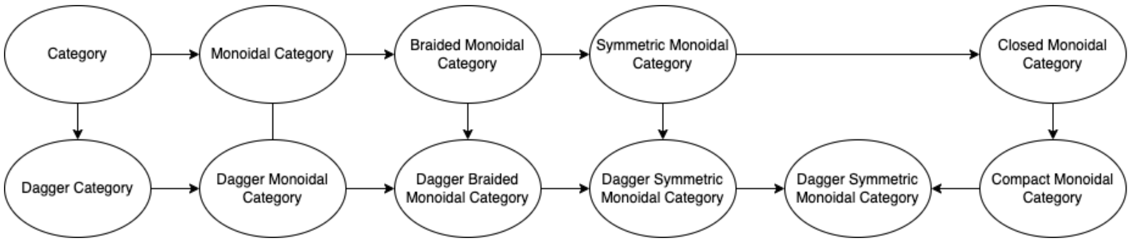

Category
Monoidal Category
Braided Monoidal Category
Symmetric Monoidal Category
Closed Monoidal Category
Dagger Category
Dagger Monoidal Category
Dagger Braided Monoidal Category
Dagger Symmetric Monoidal Category
Dagger Symmetric Monoidal Category
Compact Monoidal Category

Figure 4.1: ViCAR hierarchy of category classes.

CHAPTER 5

VISUALIZATION

The visualizer for ViCAR (VizCAR) can be seen as a generalization of the visualizer for *Vy*ZX (*Vi*ZX). Rather than providing a chronological account of the design decisions made, we go through the features common to both visualizers first, and then highlight the added features of the *Vy*ZX visualizer.

5.1 Visualization workflow

The visualization workflow can roughly be divided into five stages. First, the input term is lexed and parsed, using a combinator based library. The parser is customized to work with notation commonly used in *Vy*ZX and ViCAR. This allows for arbitrary proof states produced by Coq to be parsed. A map from variables to types is created based on hypothesis information that is also parsed. Next, the parsed term and all nested sub-terms are assigned graphical objects based on the structure of the term. These objects are assigned sizes and coordinates on an HTML canvas of size defined by the user-specified scale. The term is then rendered using those coordinates, and additional visual components including text and color are added. Finally, this canvas is displayed as a webview in the VSCode user interface, rendered as a panel alongside the code and goals.

It is worth understanding the high level workflow of how the visualizers link with coq-lsp (Figure 5.1). On a change in the user's cursor position, the coq-lsp extension client requests updated goals from the server. The server sends back the updated goals, and the client renders them in the user-facing goal panel. In the case of VizCAR, the proof context is also parsed, enhancing the visualizer with type information so that the textual goal can be annotated when necessary. This allows for arbitrary morphisms to be rendered with their input and output objects, which are only present in the typing information and not the surface term

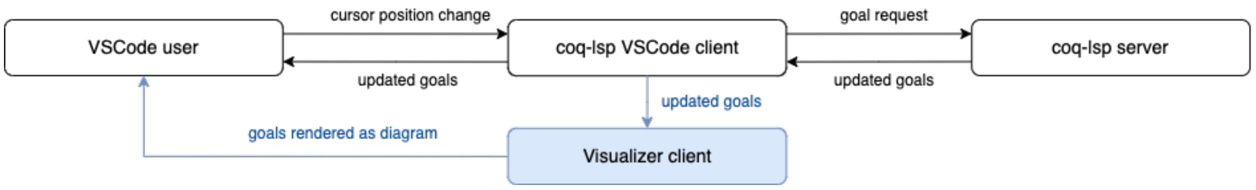
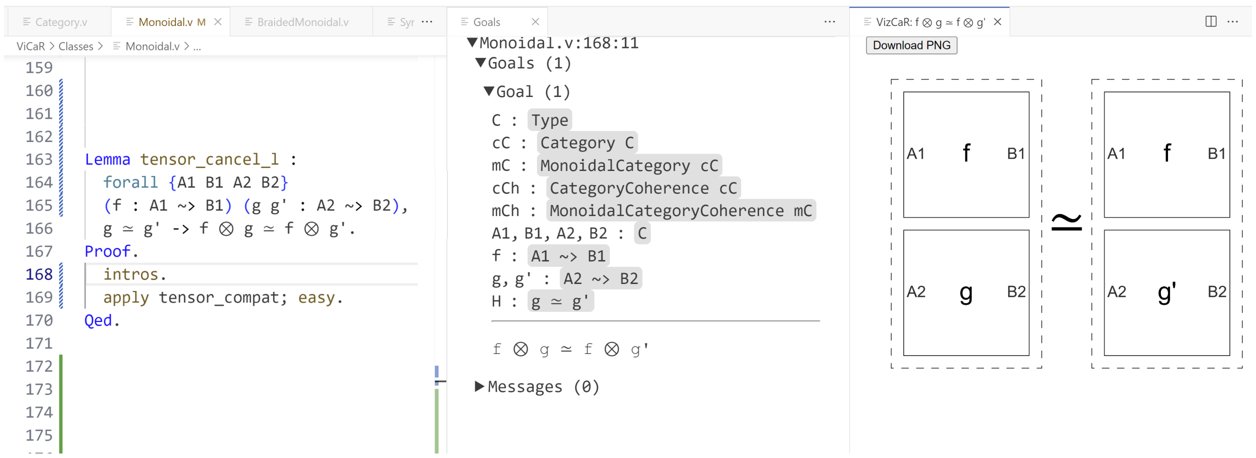


Figure 5.1: Automated visualization workflow.



```
Lemma tensor_cancel_l :
  forall {A1 B1 A2 B2}
  (f : A1 ~> B1) (g g' : A2 ~> B2),
  g ≃ g' -> f ⊗ g ≃ f ⊗ g'.
Proof.
  intros.
  apply tensor_compat; easy.
Qed.
```

```
▼Monoidal.v:168:11
 ▼Goals (1)
  ▼Goal (1)
   C : Type
   cC : Category C
   mC : MonoidalCategory cC
   cCh : CategoryCoherence cC
   mCh : MonoidalCategoryCoherence mC
   A1, B1, A2, B2 : C
   f : A1 ~> B1
   g, g' : A2 ~> B2
   H : g ≃ g'
   ________________________________
   f ⊗ g ≃ f ⊗ g'
 ▶Messages (0)
```

Figure 5.2: User's proof-writing IDE state.

itself. This simple workflow is ideal for a visualizer to fit into. When the server sends back updated goals to the coq-lsp client, one copy of these goals is sent to the goal panel, while another is sent to the visualizer to render an updated diagram. The server continues listening for cursor movements from the user, and hence new goal states are rendered automatically. Thus, the visualizer is silently integrated into the user's proof workflow (Figure 5.1).

The visualizer is intended as an *addition* to the textual goal state, rather than a replacement: the ideal IDE setup can be seen in Figure 5.2. As the goal is visualized alongside its textual representation, the proof engineer can map visual components to textual components, allowing for her to choose the correct textual rewriting tactics.

5.2 Diagrammatic grammar

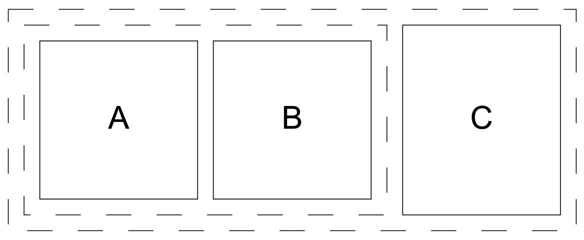
For the sake of completeness, we provide a full diagrammatic grammar for the two visualizers. All diagrams in this section are images directly generated by the visualizers.

5.2.1 *ViZX*

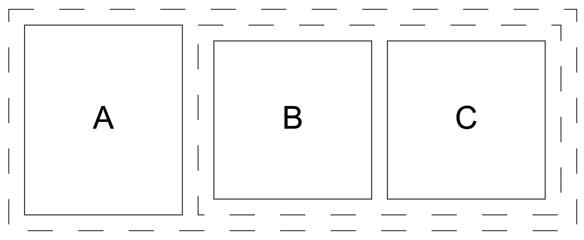
The individual elements of the *Vi*ZX grammar are shown in Figure 5.8. Since the visualizer is intended to be used in the context of a Coq proof, there are no semantic checks enforced, just static ones: any term that fits the defined grammar may be visualized, even if it does not fit the conditions defined in Figure 3.1: for example, something like $Z\ 1\ 1\ \alpha \leftrightarrow Z\ 4\ 1\ \alpha$ may be rendered, despite the output arity of the first spider not being equal to the input arity of the second spider, as is required in the semantics in Figure 3.1. Specific "transforms", such as the colorswap ($\odot$), adjoint ($\dagger$), etc. are rendered the same as arbitrary functions applied to diagrams.

The individual elements may seem simple enough, but their utility is best seen in more complex diagrams combining several elements.

Let us look at our motivating example regarding associativity. We want $(A \leftrightarrow B) \leftrightarrow C$ to be rendered distinctly from $A \leftrightarrow (B \leftrightarrow C)$. Figure 5.3 shows the rendering of both terms distinctly, and the explicit associative structure is clear.



(a) `(A ↔ B) ↔ C`



(b) `A ↔ (B ↔ C)`

Figure 5.3: Explicit associativity in *Vi*ZX.

Another important difference is the use of *foliations* to highlight *structure*. A foliation is a composition of ”stacks,“ each of which is the tensor of identity morphisms and a single non-identity morphism. Any morphism in a symmetric monoidal category can be represented as a foliation [9]. *Vi*ZX diagrams are essentially foliations of string diagrams, with identities, units, and counits represented as explicit nodes. This follows quite naturally from the inductive definition of ZX-diagrams in *Vy*ZX, and importantly, aids in reasoning about graphical structures inductively. We see in Figure 5.4 the explicit nodes allocated to wires and caps, rather then them just being implicit tools for connection. Figure 5.5 shows an example of the visualizer's representation compared to the canonical diagrammatic representation presented earlier.

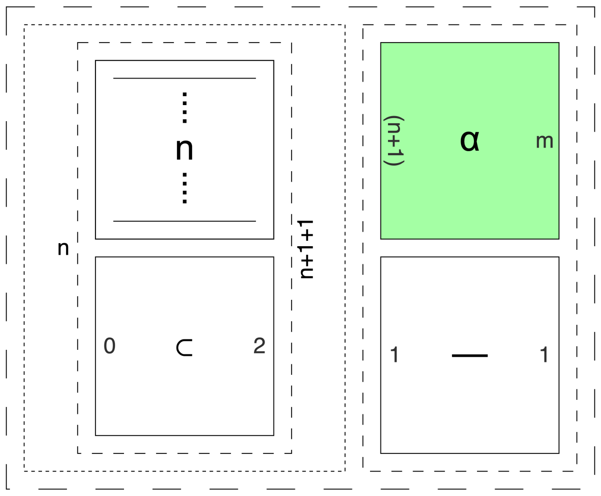


Figure 5.4: An example of explicit structure.

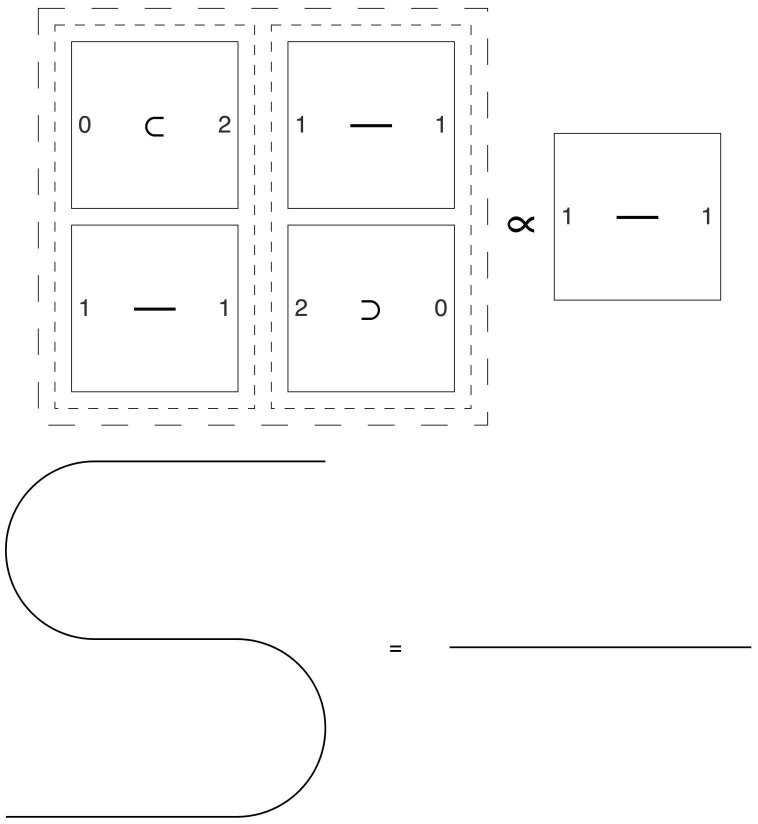
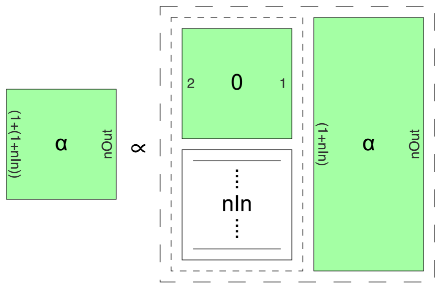
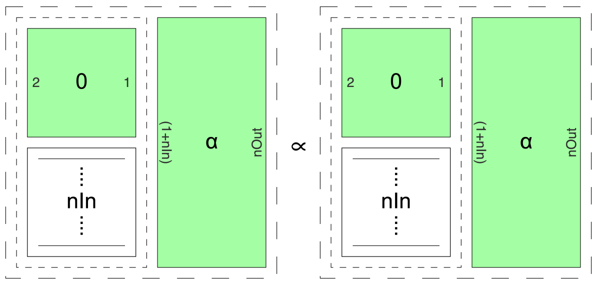


Figure 5.5: The yanking rule in *Vi*ZX and canonically.

Proofs in *Vy*ZX reason about proportionality. We use $\propto$ as the primary constructor for diagrammatic proof: most proofs aim to establish the equivalence of two ZX-diagrams, and this can be done by proving the two are proportional. A statement of proportionality is essentially what must be proved (5.6a), which must be reduced to a reflexive statement that can be discharged trivially (5.6b).



(a) Proportionality of two diagrams.



(b) Final, reflexive proof obligation

Figure 5.6: The initial and final state of a proof.

For ease of structural visibility, we manipulate the scale of stacked and composed diagrams: stacked components have the same width, while composed components have the same height. This can be seen in Figure 5.6b.

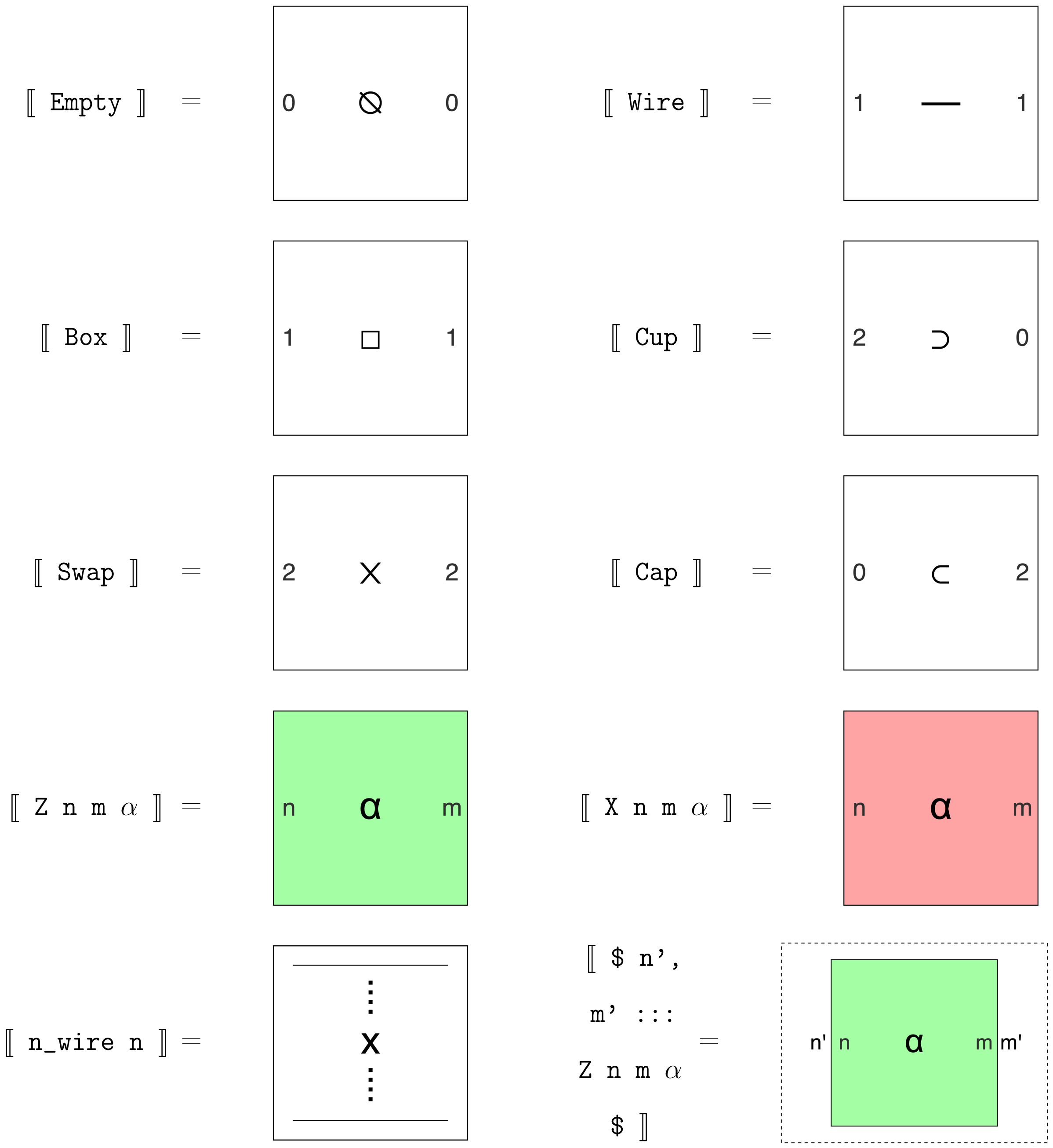


Figure 5.7: Diagrammatic grammar for *ViZX*

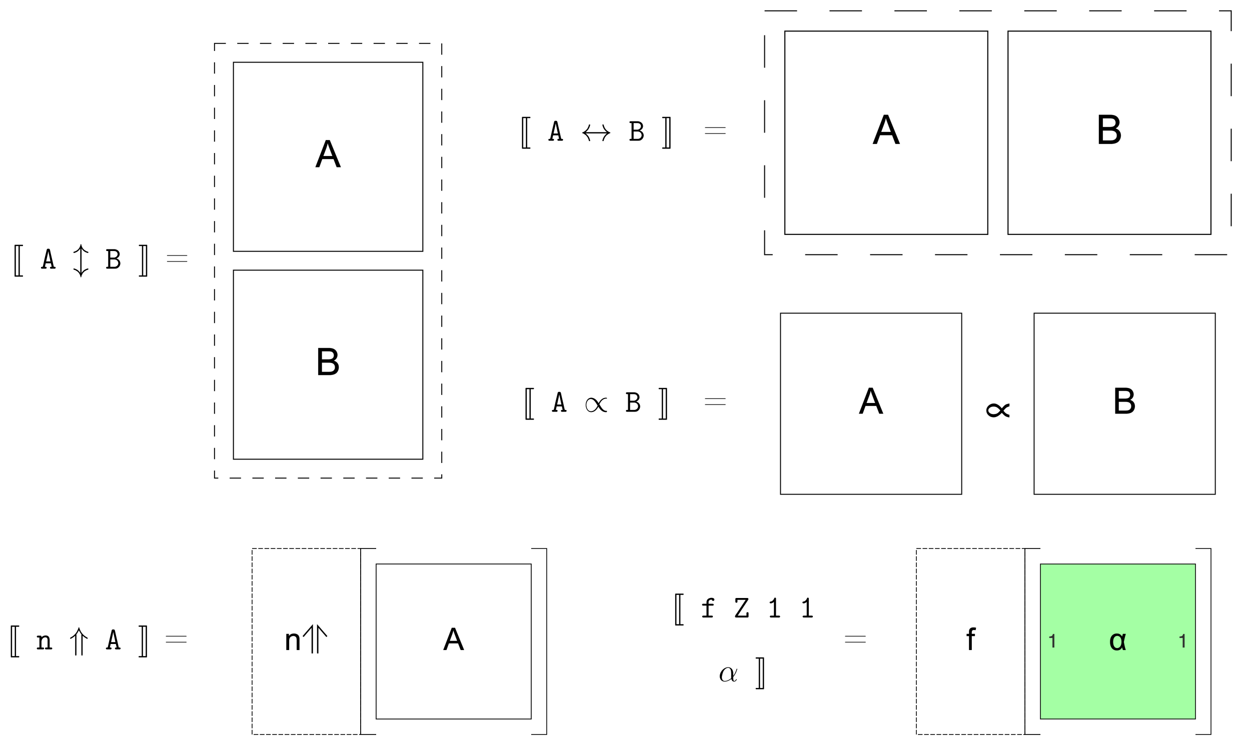
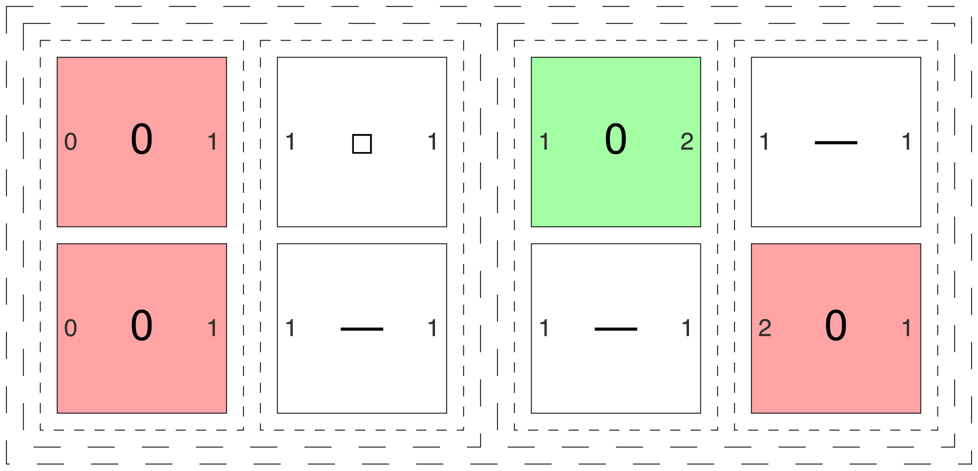
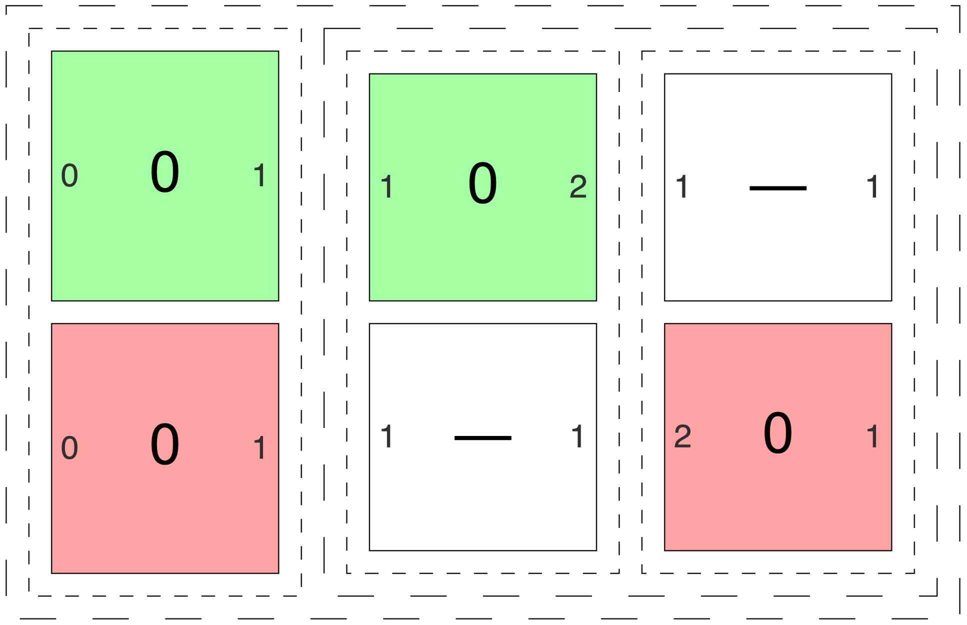


Figure 5.8: Diagrammatic grammar for *Vi*ZX (cont.)

To show an example of structural editing within *Vy*ZX, we prove the correctness of the bell state preparation circuit. First, we transform the standard Bell pair circuit into a ZX-diagram by translating gates as shown in Section 2.3.3. Figure 5.9a shows the outcome of this translation: The two X spiders on the left prepare qubits in the $|0\rangle$ state, which is followed by a Hadamard and CNOT gate. We start the proof by reassociating to use bi-Hadamard color-swapping on the top wire in Figure 5.9b. After that, we reassociate (see Figure 5.9c) and fuse the two Z spiders in Figure 5.9d. The next step is to reassociate (Figure 5.9e) and to fuse the X spiders in Figure 5.9f. Lastly, in Figure 5.9g, we simplify the resulting diagram and then convert the spider into a cap in Figure 5.9h.

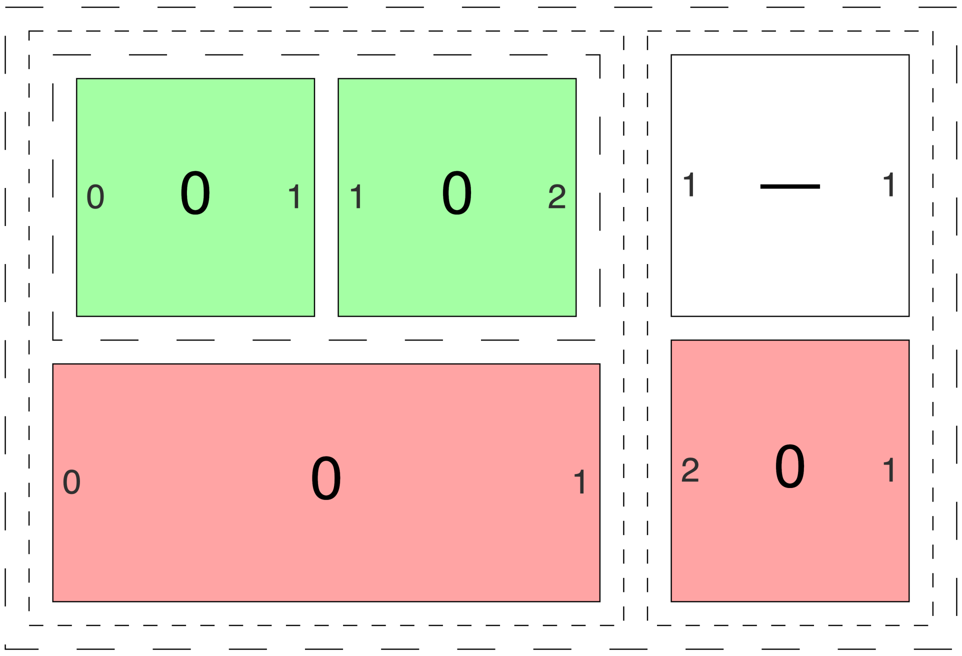


(a) `bell_state_prep_correct` lemma

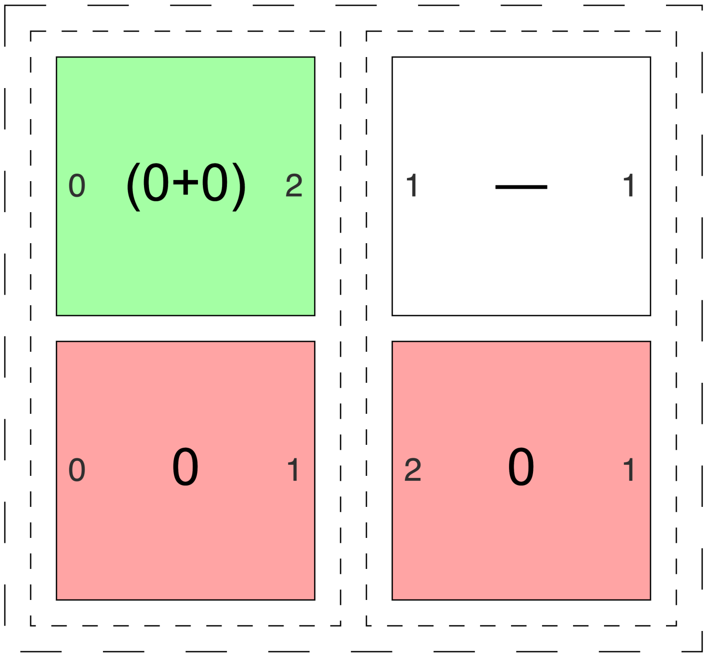


(b) Reassociate and color-swap top-left node

Figure 5.9: Proof of the correctness of bell state preparation in *Vy*ZX.

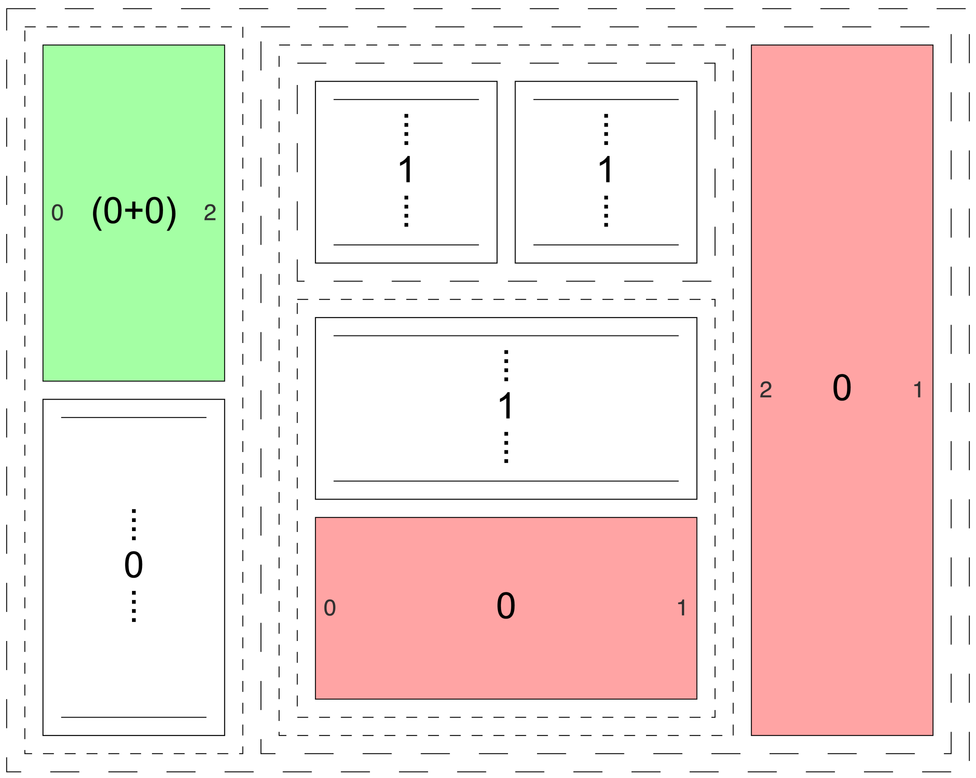


(c) Reassociate for fusion

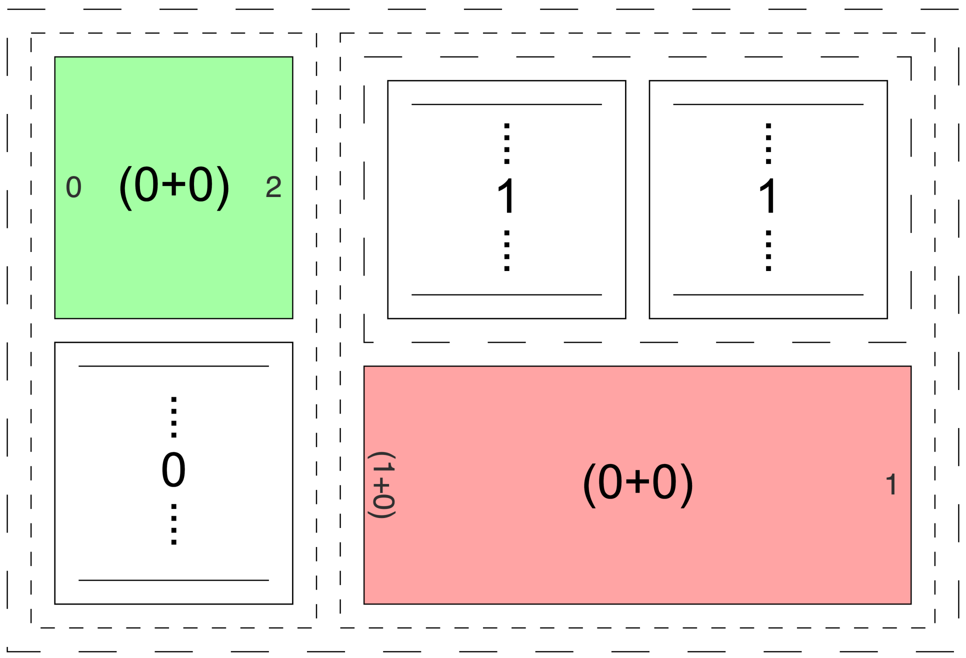


(d) Fuse top-left spiders

Figure 5.9: Proof of the correctness of bell state preparation in $VyZX$. (cont.)

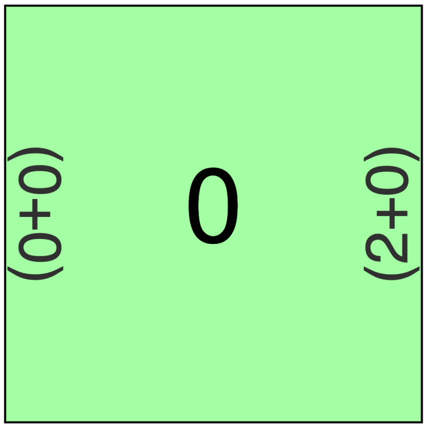


(e) Add auxiliary wires and reassociate

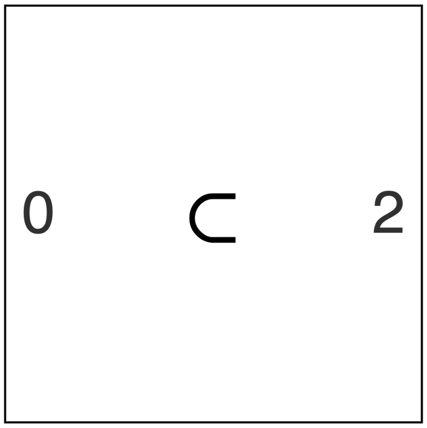


(f) Fuse bottom-right spiders

Figure 5.9: Proof of the correctness of bell state preparation in *Vy*ZX. (cont.)



(g) Remove auxiliary wires and trivial spider



(h) Convert equivalent spider into cap

Figure 5.9: Proof of the correctness of bell state preparation in *Vy*ZX. (cont.)

5.2.2 *VizCaR*

VizCAR has constructors that are similar to *Vi*ZX structurally, defined in Figure 5.12. The main differences lie in the generalization. Since VizCAR can visualize any arbitrary structure as long as it instantiates one of the category classes in its domain, it does not have features such as colors associated with specific constructors. VizCAR also reasons about categorical structures more *complex* than *Vy*ZX, and additional constructs are defined. We also have the notion of *isomorphisms*, which are visualized distinctly from unidirectional morphisms via higher line weights. We also visualize unitors and associators with the relevant parametric information.

VizCAR has the same advantages as *Vi*ZX when it comes to diagrammatic design choices and readability, so we will not repeat them here; instead we direct the reader to Figure 5.10 and Figure 5.11 as examples of composite diagrams.

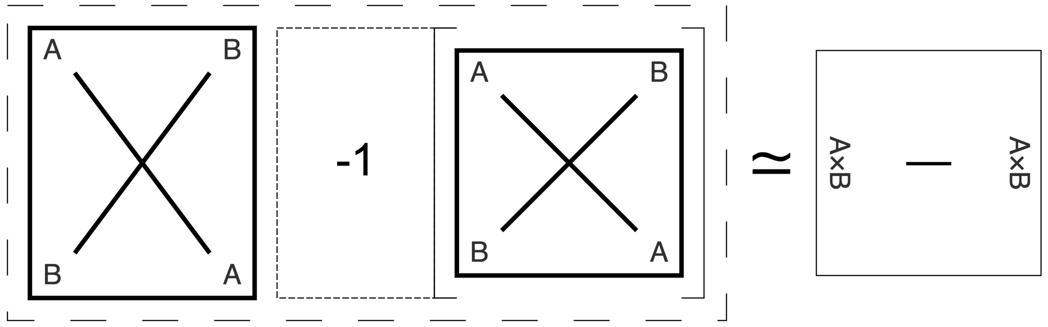


Figure 5.10: The inverse braiding lemma.

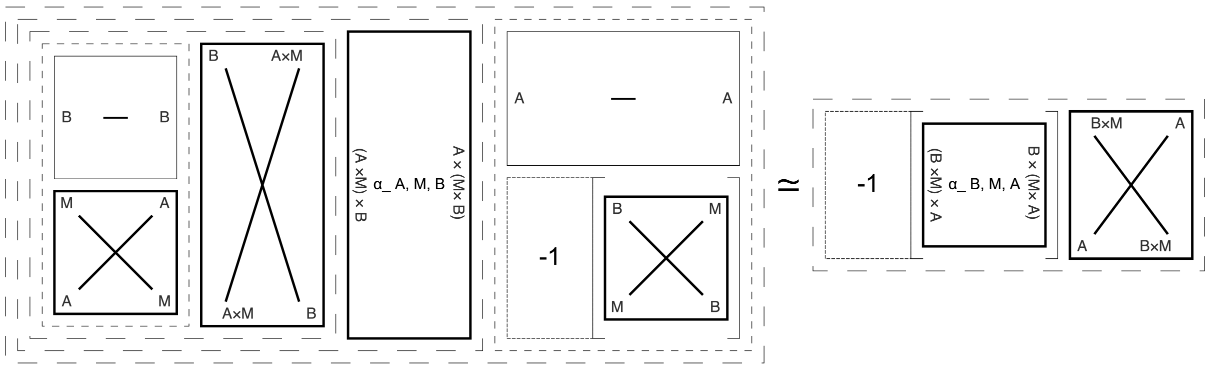


Figure 5.11: A composite VizCAR diagram.

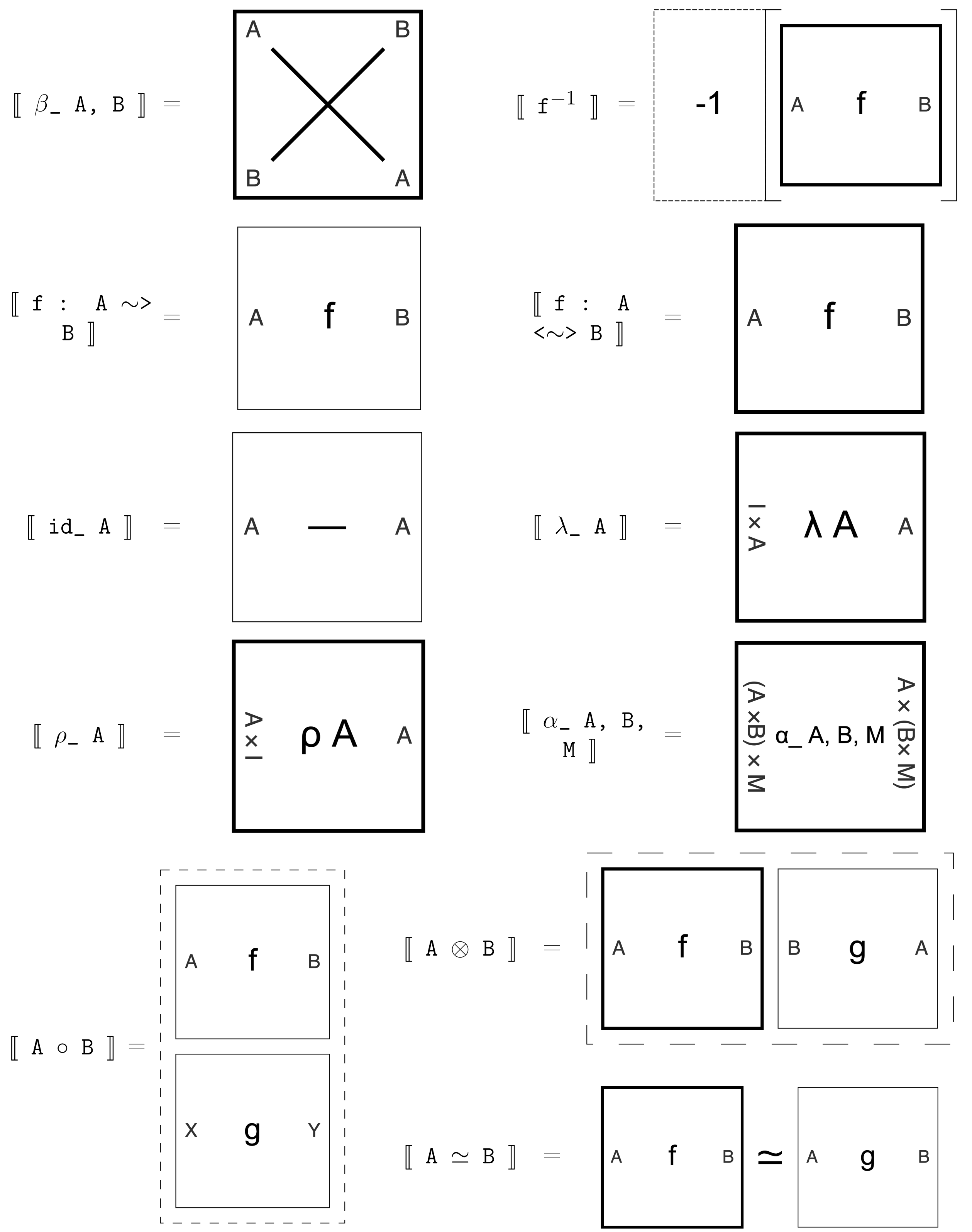


Figure 5.12: Diagrammatic grammar for VizCAR. Note that typing information is not present in terms, but in the context.

CHAPTER 6

RELATED WORK

There is a whole field of study dedicated to data and program visualization; we choose to specifically focus on works in proof visualization.

Proof assistants utilizing visualization are popular. There are examples of primarily visual proof assistants [11], but our domain corresponds more closely to primarily textual proof assistants with graphical components [1]. The proof assistant Lean [7] is equipped with a customizable, extensible user interface, ProofWidgets [19], effectively allowing users to define interactive graphical components that incorporate with the proof engineering workflow. Since this is a completely generalized framework, applicable to pretty much any user defined types with no semantic constraints, users are able to have highly customizable visualizations, at the cost of more precise specifications.

Kissinger [13] developed Chyp, an interactive theorem prover for symmetric monoidal categories with string diagram visualizations. Chyp allows users to define axiomatic rewrite rules and generators, which can then be visualized using the canonical diagrammatic grammar for process theories. Since Chyp utilizes axiomatization, it is able to unify equivalence with equality – terms that are semantically equivalent are automatically treated as equal. This allows Chyp to utilize the familiar conventional string diagram format for visualization, rendering two semantically equivalent terms in the same, abstract representation – eliding the actual structural information from the visualization. Since *Vy*ZX and ViCAR are defined in Coq, semantic equivalence can only be transformed to equality via an explicit rewrite rule, and hence the structural information is an important part of our visualization.

Lafont [14] develops a diagram editor for reasoning about category theory in Coq, providing a bidirectional interface for reasoning about categories in Coq's UniMath library. This editor allows for categorical proof reasoning in a manner almost identical to pen-and-paper proofs. While this editor and the UniMath development aim to aid reasoning about pure category theory itself, our use case is reasoning about *instantiations* of categories, and the two make different design decisions; for instance, our diagrams choose to highlight overall structure.

CHAPTER 7
FUTURE WORK

There are two main directions we would like to explore within the realm of visualizing graphical structures in interactive theorem provers – allowing for user-specified visualization properties, and allowing users to interact directly with the visualizations. We elaborate on both below.

7.1 Customizable visualization

Though *Vi*ZX was developed before VizCAR, *Vi*ZX is an instantiation of the more general categorical block structure visualization. While the two visualizers function independently, they share a lot of structure, and one may wish to extend this common structure to other instantiations of categorical structures, especially with additional visual features such as colors in *Vi*ZX. The current setup would involve creating a new visualizer for each instantiation, which is not scaleable or the best option. We are exploring the use of a *customizable* visualizer, where the user can specify properties of their desired visualizations through a configuration file that the visualizer can parse. While the goal is not to get to a fully featured widget library like Nawrocki et al. [19], it is to allow users to add custom visualization directives for simple constructs such as color, size, textual mappings, etc. THis would also allow for *Vi*ZX to be an instance of VizCAR, rather than the two being entirely independent.

7.2 Interactive visualization

Another future direction is exploring an *interactive* visualizer, where purely graphical rewriting is an option. We envision a system that follows the work of Donato et al. [8] where the process of writing proofs can be done without the user having to interact with the textual level. This approach raises questions about whether we would want to keep the textual representation alongside the graphical one as we do now, establishing a bidirectional editing system, or if we would want to get rid of textual reasoning entirely, enabling pure diagrammatic reasoning.

CHAPTER 8
CONCLUSION

We develop a methodology for the visualization of diagrams of categorical structures, with an instantiation for the ZX-calculus. This visualization grammar is specialized for interactive proof assistants, where contextual typing information may be helpful, and where we must deal with several verbose intermediate stages. Through integration with the Coq proof assistant, we enable automated visualization of all valid proof states, and by conveying structural properties explicitly, we allow for easier identification of patterns by the proof engineer.

REFERENCES

[1] 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`.

[2] Bob Coecke and Ross Duncan. Interacting quantum observables. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming*, pages 298–310, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-70583-3. doi:10.1007/978-3-540-70583-3_25.

[3] Bob Coecke and Aleks Kissinger. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning.* Cambridge University Press, 2017.

[4] The Coq Development Team. The Coq reference manual, version 8.4, August 2012. Available electronically at `http://coq.inria.fr/doc`.

[5] H. B. Curry. Functionality in combinatory logic*. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934. doi:10.1073/pnas.20.11.584. URL `https://www.pnas.org/doi/abs/10.1073/pnas.20.11.584`.

[6] The Coq LSP Developers. GitHub - ejgallego/coq-lsp: Visual Studio Code Extension and Language Server Protocol for Coq — github.com. `https://github.com/ejgallego/coq-lsp`, .

[7] The Lean 4 Developers. GitHub - leanprover/lean4: Lean 4 programming language and theorem prover — github.com. `https://github.com/leanprover/lean4`, .

[8] Pablo Donato, Benjamin Werner, and Kaustuv Chaudhuri. Integrating graphical proofs in coq, Jan 2023.

[9] Dan Ghica and Fabio Zanasi. String diagrams for λ-calculi and functional computation, 2023.

[10] William Alvin Howard. The formulae-as-types notion of construction. In Haskell Curry, Hindley B., Seldin J. Roger, and P. Jonathan, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

[11] 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.

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

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

[14] Ambroise Lafont. A diagram editor to mechanise categorical proofs. In *35es Journées Francophones des Langages Applicatifs (JFLA 2024)*, Saint-Jacut-de-la-Mer, France, January 2024. URL `https://hal.science/hal-04407118`.

[15] J. Lambek. Cartesian closed categories and typed λ-calculi. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, pages 136–175, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg. ISBN 978-3-540-47253-7.

[16] Adrian Lehmann, Ben Caldwell, Bhakti Shah, and Robert Rand. Vyzx: Formal verification of a graphical quantum language, 2023.

[17] Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 2013.

[18] 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.

[19] Wojciech Nawrocki, Edward W. Ayers, and Gabriel Ebner. An Extensible User Interface for Lean 4. In Adam Naumowicz and René Thiemann, editors, *14th International Conference on Interactive Theorem Proving (ITP 2023)*, volume 268 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:20, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-284-6. doi:10.4230/LIPIcs.ITP.2023.24. URL `https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2023.24`.

[20] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. The MIT Press, 08 1991. ISBN 9780262288460. doi:10.7551/mitpress/1524.001.0001. URL `https://doi.org/10.7551/mitpress/1524.001.0001`.

[21] P. Selinger. *A Survey of Graphical Languages for Monoidal Categories*, page 289–355. Springer Berlin Heidelberg, 2010. ISBN 9783642128219. doi:10.1007/978-3-642-12821-9_4. URL `http://dx.doi.org/10.1007/978-3-642-12821-9_4`.

[22] Bhakti Shah, William Spencer, Laura Zielinski, Ben Caldwell, Adrian Lehmann, and Robert Rand. Vicar: Visualizing categories with automated rewriting in coq, 2024.

[23] Matthieu Sozeau. Generalized rewriting, Jun 2023. URL `https://github.com/coq/c`

oq/blob/58ac2d8dd403fa7a96429fc1f839225d83be9566/doc/sphinx/addendum/generalized-rewriting.rst.

[24] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08, page 278–293, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 9783540710653. doi:10.1007/978-3-540-71067-7_23. URL https://doi.org/10.1007/978-3-540-71067-7_23.

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