*Vy*ZX

Formal Verification of a Graphical Language

Adrian Lehmann **Benjamin Caldwell** Bhakti Shah Robert Rand

Department of Computer Science
University of Chicago

Why Verify?

Interactive proof is a useful tool to assist our reasoning process

Why Verify?

Interactive proof is a useful tool to assist our reasoning process

Fully Axiomatic:

- Quantomatic (https://quantomatic.github.io/)
- ZX Calculator (zx.cduck.me)
- Chyp (https://github.com/akissinger/chyp)

Why Verify?

Interactive proof is a useful tool to assist our reasoning process

Fully Axiomatic:

- Quantomatic (https://quantomatic.github.io/)
- ZX Calculator (zx.cduck.me)
- Chyp (https://github.com/akissinger/chyp)

We want instead to embed the ZX-Calculus in a proof assistant without axiomatizing anything.

Why Coq?

Three primary benefits

- Extraction
- SQIR
- QuantumLib

Research Question

How can we embed a diagrammatic language (The ZX-Calculus) into Coq in a way that best utilizes the existing tools in Coq?

Our Approach

Diagrams must have a "semantic backing", a function that evaluates diagrams as matrices

Enables:

- Smaller core of truth
- Verified conversions between circuits and diagrams
- Easy to state equivalence of diagrams based on equivalence of semantics

Our Approach

Diagrams must have a "semantic backing", a function that evaluates diagrams as matrices

Enables:

- Smaller core of truth
- Verified conversions between circuits and diagrams
- Easy to state equivalence of diagrams based on equivalence of semantics

Downside:

- Difficult to apply semantics to a undirected and potentially cyclic graph

Easy Form for Diagram Computation

A common trick to compute semantics for a diagram is to break them down into single pieces that are stacked together and composed horizontally. You can also arrive here using the language of category theory.

We take this inspiration to define a simple structure that we can define semantics for in Coq.

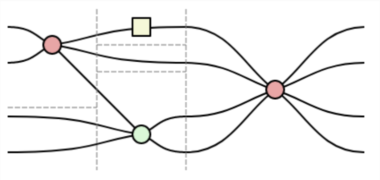
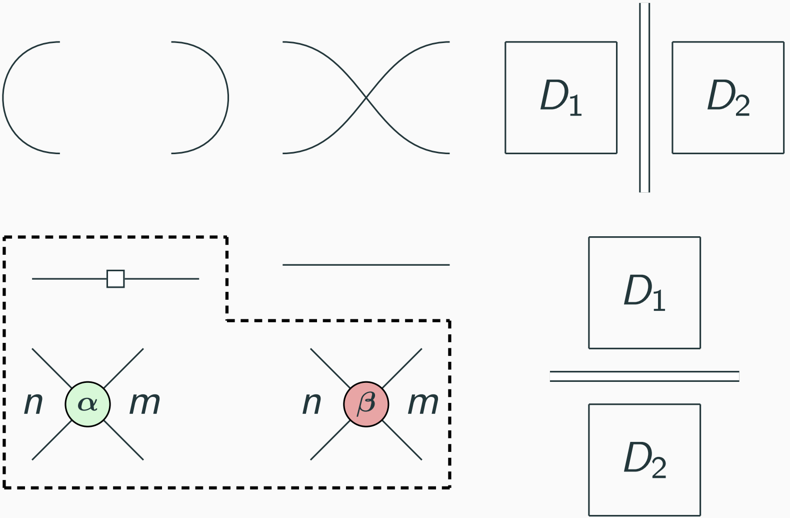


Figure 1: An image from the ZX-calculus website tutorial with the caption "An indication of how to break a diagram down into smaller diagrams, so that each cell contains only one element"

ZX Diagrams as string diagrams



Inductive ZX Diagrams

To define our ZX diagrams, we take these string diagram constructions and add Z and X spiders.

$$\frac{\textsf{in out} : \mathbb{N} \qquad \alpha : \mathbb{R}}{\texttt{Z\_Spider}\ \textsf{in out}\ \alpha : \texttt{ZX}\ \textsf{in out}} \qquad \frac{\textsf{in out} : \mathbb{N} \qquad \alpha : \mathbb{R}}{\texttt{X\_Spider}\ \textsf{in out}\ \alpha : \texttt{ZX}\ \textsf{in out}}$$

$$\frac{}{\texttt{Cap} : \texttt{ZX}\ 0\ 2} \qquad \frac{}{\texttt{Cup} : \texttt{ZX}\ 2\ 0} \qquad \frac{}{\texttt{Swap} : \texttt{ZX}\ 2\ 2} \qquad \frac{}{\texttt{Empty} : \texttt{ZX}\ 0\ 0}$$

$$\frac{\textsf{zx1} : \texttt{ZX}\ \textsf{in mid} \qquad \textsf{zx2} : \texttt{ZX}\ \textsf{mid out}}{\texttt{Compose}\ \textsf{zx1 zx2} : \texttt{ZX}\ \textsf{in out}} \qquad \frac{}{\texttt{Wire} : \texttt{ZX}\ 1\ 1}$$

$$\frac{\textsf{zx1} : \texttt{ZX}\ \textsf{in1 out1} \qquad \textsf{zx2} : \texttt{ZX}\ \textsf{in2 out2}}{\texttt{Stack}\ \textsf{zx1 zx2} : \texttt{ZX}\ (\textsf{in1} + \textsf{in2})\ (\textsf{out1} + \textsf{out2})} \qquad \frac{}{\texttt{Box} : \texttt{ZX}\ 1\ 1}$$

Semantics

To verify transformations on diagrams, we introduce a semantic function for our diagrams, $[\![\bullet]\!]$ :: ZX n m $\rightarrow \mathbb{C}^{m\times n}$. These semantics are built on the Coq library QuantumLib.

$$
\begin{aligned}
[\![\texttt{Z\_Spider n m } \alpha]\!] &\mapsto \begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & 0 \\ 0 & 0 & e^{i\alpha} \end{bmatrix} \\
[\![\texttt{X\_Spider n m } \alpha]\!] &\mapsto H^{\otimes m} \times \begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & 0 \\ 0 & 0 & e^{i\alpha} \end{bmatrix} \times H^{\otimes n}
\end{aligned}
$$

More Semantics

$$
\begin{aligned}
[\![\texttt{Cap}]\!] &\mapsto \begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix} \\
[\![\texttt{Cup}]\!] &\mapsto \begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix}^\top \\
[\![\texttt{Swap}]\!] &\mapsto \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
[\![\texttt{Empty}]\!] &\mapsto \begin{bmatrix} 1 \end{bmatrix} \\
[\![\texttt{Wire}]\!] &\mapsto I_{2\times 2} \\
[\![\texttt{Box}]\!] &\mapsto H \\
[\![\texttt{Compose zx1 zx2}]\!] &\mapsto [\![\texttt{zx2}]\!] \times [\![\texttt{semantics(zx1)}]\!] \\
[\![\texttt{Stack zx1 zx2}]\!] &\mapsto [\![\texttt{semantics(zx1)}]\!] \otimes [\![\texttt{semantics(zx2)}]\!]
\end{aligned}
$$

Proportionality

We choose to define proportionality up to constant factor as follows, using the notation $\propto$.

$$\exists c \neq 0 : [\![zx1]\!] = c * [\![zx2]\!] \implies zx1 \propto zx2$$

We can encode this proportionality definition in Coq easily using the inductive definition and semantic function defined earlier.

Utilizing Coq's Rewrite System

Proof in Coq has 3 important parts:

- Tactics
- Hypotheses
- Goal

The hypotheses and goal make up the proof state, while tactics are applied line by line to update the proof state.

Proof Example

Proofs in Coq must first be stated using keywords like *Lemma* then proofs are surrounded by *Proof* and *Qed*. Each line of a Coq proof has an associated proof state, and tactics operate and update these proof states.

```
Open Scope nat.

Lemma nat_commutation : forall (n m : nat), n + m = m + n.
Proof.
  intros n m.
  induction n as [| k IHn].
  - rewrite Nat.add_0_r.
    reflexivity.
  - rewrite Nat.add_succ_r.
    rewrite ← IHn.
    reflexivity.
Qed.
```

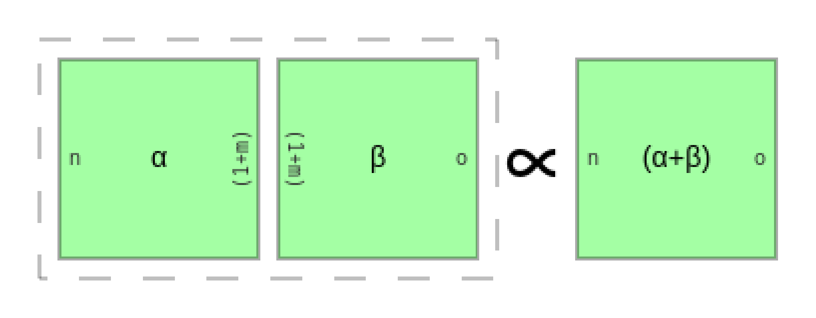
Goal and Hypotheses Example

Goals and hypotheses are rendered based on the currently active line.

This form ends up being difficult to read, so we have a VSCode extension which works with the language server to render diagrams.

Blocky Renderings

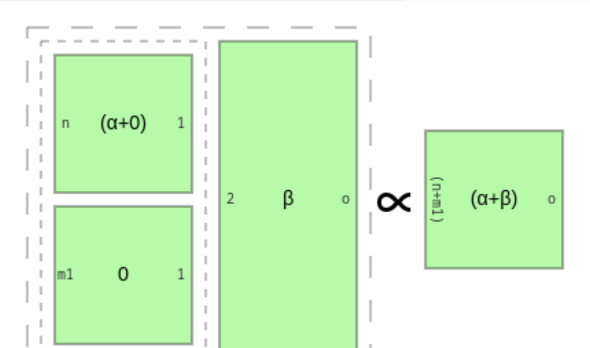
$\texttt{Z n (S m)}\ \alpha \leftrightarrow \texttt{Z (S m) o}\ \beta \propto \texttt{Z n o}\ (\alpha + \beta)$



Associativity information

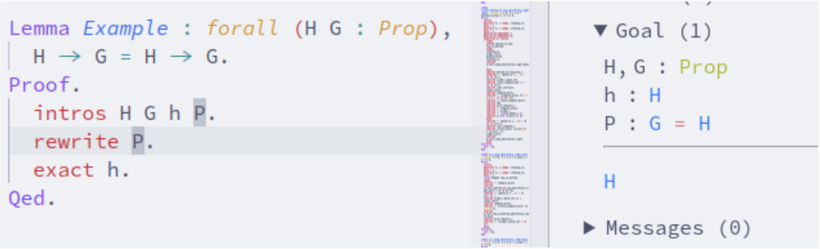
```
▼ Goal (1)
  n, m0, m1, o : nat
  α, β : R
  ─────────────────────────────────────────
  Z n 1 (α + 0) ↕ Z m1 1 0 ⟷ Z 2 o β ∝ Z (n + m1) o (α + β)

▶ Messages (0)
```

ViZX: Z n 1 (α + 0) ↕ Z m1 1 0 ⟷ Z 2 o β ∝ Z (n + m1) o (α + β)



Tactics: Rewrite

Given a goal state G and a hypothesis $Hyp : G = H$, we can apply the tactic `rewrite G` to update our proof state to be H.



We extend $\propto$ so that it can rewrite within ZX-diagrams using a tool within Coq called *parametric relations* and *parametric morphisms*.

Tactics: Apply

Given a goal state G and a hypothesis $Hyp : H \to G$, we can use the tactic `apply Hyp` to update our proof state to be H.

```
▼ Goal (1)
 H, G : Prop
 h : H
 Hyp : H → G
 ─────────────
 G
```

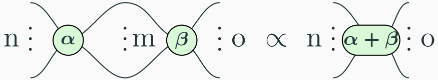
Three Proof Strategies

1. Proof through semantics

Three Proof Strategies

1. Proof through semantics

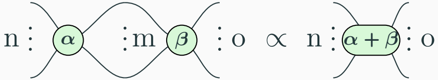
2. Inductive proof



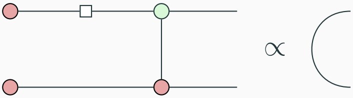
Three Proof Strategies

1. Proof through semantics

2. Inductive proof



3. Diagrammatic proof



Current Features

- Complete set of rewrite rules proven in Coq
- Automation to simplify diagrams
- Reasoning about

Discussion

- Any ZX diagram can be expressed
- Multiple ways to encode
- Deal with associativity information
- Dimensionality issues
- Complete set of rules have been verified, still there are reasonsa to do proof via semantics.

How can I verify my graphical language?

- Find underlying categorical structure
- Formally extend structure with generators
- Translate into inductive constructors
- Define a semantic for the inductive constructors
- Deal with resulting associtativity issues

How can I verify my graphical language?

- Find underlying categorical structure
- Formally extend structure with generators
- Translate into inductive constructors
- Define a semantic for the inductive constructors
- Deal with resulting associtativity issues

We have separated the category theory work from VyZX into its own project, ViCaR. ViCaR also includes proofs that VyZX satisfies the definitions of dagger compact categories!

Future work

- Restore connection information, potentially have connection information generate these blocky diagrams.
- Verify ZX-based compiler.
- Rewriting diagrams without having to worry about associativity information.

Summary

- Defined ZX diagrams inductively
- Inspired by string diagrams
- Multiple proof strategies

Find *Vy*ZX on GitHub

https://github.com/inQWIRE/VyZX

arXiv

https://arxiv.org/abs/2311.11571

References

- Bob Coecke and Aleks Kissinger, *Picturing quantum processes: A first course in quantum theory and diagrammatic reasoning*, Cambridge University Press, 2017.
- Jonathan Castello, Patrick Redmond, and Lindsey Kuper, *Inductive diagrams for causal reasoning*, 2023.
- Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks, *A verified optimizer for quantum circuits*, Proc. ACM Program. Lang. **5** (2021), no. POPL.
- John van de Wetering, *Zx-calculus for the working quantum computer scientist*, 2020.