Visualizing Graphical Proofs in Coq

BHAKTI SHAH, University of Chicago, USA

Advisor: Robert Rand
Category: Undergraduate

1 MOTIVATION

Working with graphical constructs in textual form can be hard. The most common approach to working with graphical structures in textual form is the use of manual visualization — the developer informally sketching out the graphical structure to aid understanding.

In the context of interactive theorem provers, the developer is often working with a graphical structure for an extended period of time, encountering several terms in the form of intermediate proof states. It is impractical to manually visualize each intermediate proof state, as there are way too many, and the cost of manual conversion eventually adds up.

Historically, developers are inclined towards using textual buffers for the production of code — something that visual languages have failed to change. Visual elements work best when used in addition to textual elements, and previous works have concluded the same [1].

We encountered this specific situation when working on VyZX [11], an ongoing effort to build a verified library for the ZX Calculus (both elaborated upon in the following sections). Writing a generalized translation from text to visualization for the relevant grammar was the solution we landed on. Not having to "ask" for a visualization for each term was also valuable — accessing a separate tool to pipe in output from the goal state is avoidable via automation.

2 THE ZX CALCULUS AND VYZX

The ZX calculus [6] is a graphical calculus that can be used to represent quantum circuit diagrams, contrasting the circuit model usually used for the same purpose. It focuses primarily on *connectivity* information between elements. The ZX calculus is built up of ZX diagrams, graphical representations of quantum operations, and graphical rewrite rules, to prove properties of quantum states.

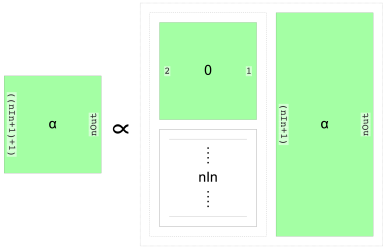
ZX diagrams are graphs that consist of green and red nodes (called Z and X spiders respectively). Each spider has a number of inputs and a number of outputs, as well as a rotation angle $\in [0, 2\pi)$. Spiders can be connected via edges [11]. We do not focus on the semantics of the calculus here, which are nicely summarized in a survey by van de Wetering [16] for the interested reader.

VyZX [10] is a verified library for the ZX Calculus in the proof assistant, Coq [7]. The core language is an inductive structure representing string diagrams & contributing, amongst other things, a set of base morphisms and the ability to compose diagrams sequentially (horizontally) and in parallel (vertically). This can be used to represent the ZX Calculus [11], albeit in a way less direct than the standard graphical representation;a large amount of graphical information is shoe-horned into a single inductive structure. This is because the canonical representation is not well suited for verification purposes — computers make it a lot more rigid than what is desired. The inductive structure makes it easier to give semantics to ZX diagrams, and works better in the context of formalization.

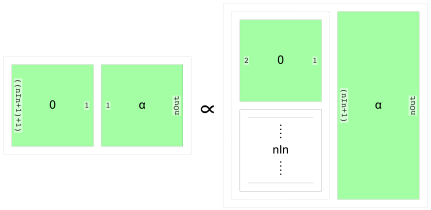
3 ZXVIZ

We implement a visualization of the ZX Calculus as formalized in VyZX, called ZXViz, integrated with the Coq language server coq-lsp [8] and VSCode client for ease of use. Below is an example of intermediate proof stages from a simple proof in the VyZX library[10], along with the corresponding visualizations. At this point, neither the visual nor the textual representation of any term should

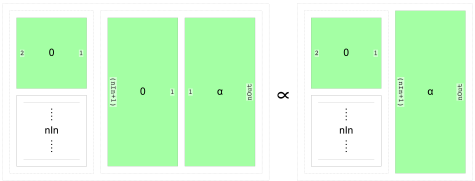
Author's address: Bhakti Shah, University of Chicago, Chicago, USA, bhaktishah@uchicago.edu.



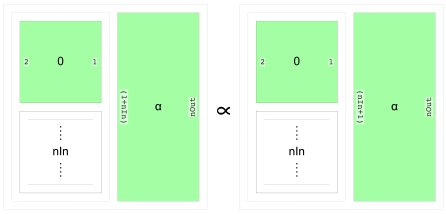
[1] `Z (S (S nIn)) nOut` $\alpha \propto$ `Z 2 1 0` $\updownarrow$ `n_wire nIn` $\leftrightarrow$ `Z (S nIn) nOut` α



[2] `Z (S (S nIn)) 1 0` $\leftrightarrow$ `Z 1 nOut` $\alpha \propto$ `Z 2 1 0` $\updownarrow$ `n_wire nIn` $\leftrightarrow$ `Z (S nIn) nOut` α



[3] `Z 2 1 0` $\updownarrow$ `n_wire nIn` $\leftrightarrow$ `Z (S nIn) 1 0` $\leftrightarrow$ `Z 1 nOut` $\alpha \propto$ `Z 2 1 0` $\updownarrow$ `n_wire nIn` $\leftrightarrow$ `Z (S nIn) nOut` α



[4] `Z 2 1 0` $\updownarrow$ `n_wire nIn` $\leftrightarrow$ `Z (1 + nIn) nOut` $\alpha \propto$ `Z 2 1 0` $\updownarrow$ `n_wire nIn` $\leftrightarrow$ `Z (S nIn) nOut` α

Fig. 1. Intermediate proof terms.

make sense. We will only expand upon the syntax and semantics absolutely necessary to understand this example. The concrete textual syntax for a `Z` spider is `Z in out rotation`, where `in, out` $\in \mathbb{N}$, `rotation` $\in \mathbb{R}$. Visually, it is represented as a green box with `in`, `out` labeling the edges, and `rotation` in the center. Vertical composition is represented textually by `term` $\updownarrow$ `term`, while horizontal composition is `term` $\leftrightarrow$ `term`. Visually, vertical composition is the placement of two terms in the same column, while horizontal composition is two terms in the same row. Equivalence of terms is represented both visually and textually by `term` $\propto$ `term`, where equivalent terms evaluate to the same semantics (up to a constant factor). `n_wire` is a function from a number `n` to a ZX diagram, that constructs a ZX diagram consisting of `n` *wires* composed vertically.

Syntax may be enough to understand the individual terms, but the diagrammatic rewrite semantics is what makes the translation from one term to another clear. The rules utilized here relate to *spider fusion*. Essentially, the composition of two spiders `Z in 1` α $\leftrightarrow$ `Z 1 out` β can be *fused* into a single spider `Z in out` γ if $\alpha + \beta = \gamma$. Informally, two spiders can fuse into a single spider if the parameters meet some constraints. The application of this rule can be seen visually in the transition from 1 to 2, and from 3 to 4. The transition from 2 to 3 can be attributed to a slightly more complex rewrite rule that allows us to split a specific spider out in a diagram, which can be seen as a restricted form of spider fusion. The visualization makes it a lot easier to understand what rule is applied where; it also aids in deciding what rule to apply next.

The ZXViz workflow can roughly be divided into five stages. First, the input term is lexed and parsed, using the parser combinator based library ts-parsec [15]. The parser is customized to work

with notation commonly used in VyZX. This allows for arbitrary VyZX proof states produced by Coq to be 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.

One of the goals of this project was the integration into the Coq workflow. A visualizer for custom data structures alone would be useful, but manual input diminishes the ease of use, and hence we wanted to look into ways to interleave it into the Coq ecosystem. We settled on Visual Studio Code (VSCode) as the target editor, and coq-lsp as the target development client. coq-lsp allowed for great integration with the interactive Coq workflow due to the compliance with the language server protocol, as well as the clear client-server architecture. The primary coq-lsp client is for VSCode, and VSCode is further equipped with several features to create web-based user extensions — both of which led to our decision to develop a tool customized to the editor.

It is worth understanding the high level workflow of how ZXViz links with coq-lsp (Figure 2). 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. This simple workflow is ideal for ZXViz 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 ZXViz 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.

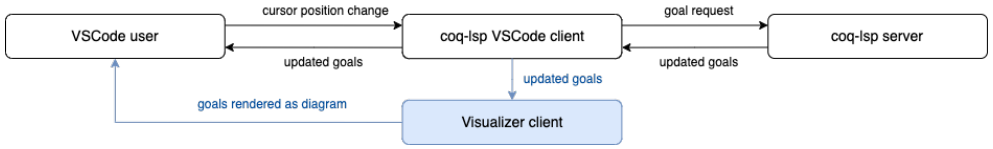


Fig. 2. coq-lsp & visualizer integrated workflow.

4 RELATED WORK

While integrating visualizations into textual buffers is an expanding field of work, visualizations in proof assistants with non-educational use cases are more limited. The natural tree like structure of proofs makes proof tree visualization useful [14]. Works in visualizing proof terms outside of a proof assistant [5], or domain-specific proof assistants [4] are also popular. In contrast to several existing use cases, we wanted a tool for advanced users working with complex structures that are hard to parse by hand. One great effort in this realm is ProofWidgets 4 [13][3][2], an extensible library of UI components in the theorem prover Lean 4 [9] that utilizes the Penrose system [17]. ProofWidgets 4 was developed concurrently with ZXViz, and is a promising step towards integrating visualizations into proof assistants.

5 CONCLUSION

This project was a step towards making it easier to reason about complex proofs in Coq relating to graphical structures. We foresee this notion being generalized to other graphical calculi such as string diagrams, globular proofs[4], and other quantum calculi such as the ZQ calculus [12].

REFERENCES

[1] Leif Andersen, Michael Ballantyne, and Matthias Felleisen. 2020. Adding Interactive Visual Syntax to Textual Code. *CoRR* abs/2010.12695 (2020). arXiv:2010.12695 https://arxiv.org/abs/2010.12695

[2] Edward William Ayers. 2021. *A Tool for Producing Verified, Explainable Proofs.* PhD thesis. University of Cambridge. Available at https://www.edayers.com/ayers_thesis_final.pdf.

[3] Edward W. Ayers, Mateja Jamnik, and W. T. Gowers. 2021. A Graphical User Interface Framework for Formal Verification. In *12th International Conference on Interactive Theorem Proving (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 4:1–4:16. https://doi.org/10.4230/LIPIcs.ITP.2021.4

[4] Krzysztof Bar, Aleks Kissinger, and Jamie Vicary. 2018. Globular: an online proof assistant for higher-dimensional rewriting. *Logical Methods in Computer Science* 14 (2018).

[5] Joachim Breitner. 2016. Visual theorem proving with the Incredible Proof Machine. In *Interactive Theorem Proving: 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings 7*. Springer, 123–139.

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

[7] The Coq Developers. [n. d.]. GitHub - coq/coq: Coq is a formal proof management system. https://github.com/coq/coq.

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

[9] The Lean 4 Developers. [n. d.]. GitHub - leanprover/lean4: Lean 4 programming language and theorem prover — github.com. https://github.com/leanprover/lean4.

[10] The VyZX Developers. [n. d.]. GitHub - inQWIRE/VyZX: Verifying the ZX Calculus — github.com. https://github.com/inQWIRE/VyZX.

[11] Adrian Lehmann, Ben Caldwell, and Robert Rand. 2022. VyZX : A Vision for Verifying the ZX Calculus. arXiv:2205.05781 [quant-ph]

[12] Hector Miller-Bakewell. 2020. Entanglement and Quaternions: The graphical calculus ZQ. arXiv:2003.09999 [quant-ph]

[13] Wojciech Nawrocki, Edward W. Ayers, and Gabriel Ebner. 2023. 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))*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany. https://doi.org/10.4230/LIPIcs.ITP.2023.8

[14] Hendrik Tews. 2011. Automatic library compilation and proof tree visualization for Coq Proof General. In *Presentation at the 3rd Coq Workshop, Nijmegen.*

[15] The ts-parsec Developers. [n. d.]. GitHub - microsoft/ts-parsec. https://github.com/microsoft/ts-parsec/.

[16] John van de Wetering. 2020. ZX-calculus for the working quantum computer scientist. arXiv:2012.13966 [quant-ph]

[17] Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Jonathan Sunshine, and Keenan Crane. 2020. Penrose: From Mathematical Notation to Beautiful Diagrams. *ACM Trans. Graph.* 39, 4 (2020).