THE UNIVERSITY OF CHICAGO

PROOF ENGINEERING FOR QUANTUM REASONING VIA E-GRAPHS

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

ADRIAN ERNST LEHMANN

CHICAGO, ILLINOIS

DECEMBER 2025

In loving memory of my grandmother *Momo*.

Her pride would not have been in the title this work confers, but in the joy of its pursuit.

She taught us that the greatest achievement is to pour your heart into what you love; for us, that became computer science.

May we all learn from her example: to love and accept people just as they are.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Thank you, John, for taking a leap of faith and admitting me to the PhD; thank you, also, for your wisdom, knowledge, and feedback that you shared with me over the years. Thank you, also, for being very supportive when I switched slightly out of your area.

Thank you, Robert, for taking me on as a student while already at UChicago and for advising me through all our paper submissions and my main projects.

Thank you, Jens, for agreeing to be part of the committee and for providing valuable feedback that greatly enhanced this thesis.

Thank you, Robert, John, Fred, Mathias, Chris, and Lindsey, for writing letters on my behalf throughout the PhD.

Thank you, Ben, for starting *VyZX* with me and for being the theoretician to my engineer.

Thank you, William, Bhakti, and Laura, for your work on our shared projects from *VyZX* via ZXViz to ViCAR.

Thank you, Byron, for organizing our reading group (and getting the best catering).

Thank you to everyone who gave feedback on my practice talks.

On a final note, thank you, Ashley, Anton, and August, for keeping me sane through all the PhD and for always believing in me.

ABSTRACT

This dissertation is an exploration of quantum formal verification through the development of VyZX, a new formalization of the ZX-calculus. VyZX uses the categorical structure of the ZX-calculus to reason about ZX diagrams, providing a framework to establish completeness, encode common rules, and investigate applications like peephole optimization. However, working within a symmetric monoidal category introduces challenges, as the same ZX-diagram can have many equivalent representations in VyZX. This results in significant manual proof effort, as there is no single "best" normal form, which is both tedious and difficult to fully automate.

To address this, use E-Graphs, a data structure for managing equivalence classes, to streamline reasoning about equivalence. We build a system that can automatically use VyZX lemmas to prove the equivalence of two semantically equivalent diagrams if they are equal up to structure. The system performs well, solving complex associativity problems in less than a second. In benchmarks, the system can handle associativity chains of up to 47, where in practice we don't see any larger chains than 8. In addition, we explore the practical limits of solving more complex equalities using custom rewrite databases. The system is able to replace and outperform autorewrite for a common problem class in the scope of VyZX.

CHAPTER 1

INTRODUCTION

1.1 Thesis

In this thesis, we explore proof engineering strategies for formal verification for quantum operations. Quantum computing lies in the sweet spot for formal verification [Ringer et al., 2019]. Quantum programs are uniquely hard to debug [García de la Barrera et al., 2023], expensive to run, and there is only a small class of algorithms we can practically use at the moment [Nielsen and Chuang, 2010a]. Therefore, the additional time and effort needed to verify tools and algorithms is warranted. The problem, however, is that in its current state, verifying quantum process in proof assistants utilizing the *Calculus of Inductive Constructions* (CIC) [Pfenning and Paulin-Mohring, 1990] requires proofs of matrix equivalence. Writing such proofs is often tedious and requires extensive knowledge of advanced linear algebra. While this method may be appealing for mathematicians, our work is directed at making the problem more approachable to the computer scientist. We conjecture that using graphical reasoning together with advanced proof automation techniques, we can reduce the friction of formally reasoning about quantum systems.

1.2 Problem I: Verified ZX-calculus

Current approaches to quantum verification often focus on either building bespoke proof assistants that have a much larger and more error-prone trusted code base or on verifying circuit identities with measurement [INQWIRE Developers, 2022, Hietala et al., 2021b]. Reasoning about circuits by themselves is an incomplete way of viewing quantum computing, however, as it only allows for unitary reasoning. Typically, once measurement

is involved, reasoning is complete but these tools also fall back on density matrix manipulation, which is typically quite involved. While this approach is powerful and complete with respect to any quantum operation, it is also very unintuitive and foundational. We do not typically reason over large boolean algebra expressions to figure out a classical program's behavior, so why should we be doing the analog for quantum computing? This is where the ZX-calculus fills the gap, a graphical reasoning system, that abstracts away from linear algebra and rather diagrammatically focuses on interactions [Coecke and Duncan, 2007]. Although there are formalizations of ZX-calculus, such as ZXLive by Shaikh et al. [2023] or quantomatic by Kissinger and Zamdzhiev [2015], these rely on axioms, which caused bugs in the real world in the past.[1] The ZX-calculus rules can be proven correct using matrix semantics, eliminating bugs of this type.

1.3 Solution I: *VyZX*

To bridge this gap, we present *VyZX*, a formally verified implementation of the ZX-calculus in Rocq. VyZX proves all common rules, up to Clifford+T completeness, and enables proof engineers to reason about quantum programs graphically. *VyZX* also comes with a visualizer to aid graphic reasoning, called ZXViz. To achieve graphical reasoning in Rocq, *VyZX* views the ZX-calculus as a symmetric monoidal category [Selinger, 2010], instead of a graph based on adjacency. This means in practice that *VyZX* views the ZX-calculus as a series of vertical and horizontal compositions of basic objects. This "block view" allows for matrix semantics to be derived from ZX-diagrams and hence gives greater confidence than existing tools by relying on a much smaller trusted code base: Rocq and its real number library.

1. See `https://github.com/zxcalc/zxlive/issues/361` as an example of a bug that would have been caught by verification with respect to the underlying semantics

1.4 Problem II: Structural Automation

Unfortunately, the block view obscures connectivity and adjacency information. Diagrams are often associated in a way that is not conducive to the next proof step. To resolve this, proof-engineers often need to rewrite between semantically equivalent but structurally different diagrams. In fact, about one quarter of the total lines of proof code in *VyZX* are to deal with such structural rewrites. While these structural problems are typically not necessarily difficult proofs, they distract from the original proof, and given their abundance still add up in time taken.

1.5 Solution II: e-graph automation

To rectify this situation, we looked at different proof automation options, and decided that using e-graphs, as introduced by Nelson [1980], are likely the best way. e-graphs are data-structures that allow reasoning over equivalence classes and avoid ordering problems and recomputation. This makes them very well suited for a "smart" proof search over brute-force approaches such as `autorewrite` or custom algorithms implemented in `Ltac`/`Ltac2`. Modern e-graph systems also allow explaining the steps they took to reach the equivalence using a process called proof explanation Nieuwenhuis and Oliveras [2005]. Further, we will build on this approach to allow for rewrites with arbitrary *VyZX* lemmas databases as a replacement for `autorewrite`. While this problem is in general NP-hard [Nieuwenhuis and Oliveras, 2005], we aim to explore the practical limitations and show that there are practical proof-automation applications in this domain specific approach that previous approaches do not capture.

CHAPTER 2

PRIOR WORK

2.1 Quantum Computing

This section is not intended to be a full review of quantum computing. Rather, it is a pragmatic overview of prior related work that is useful for understanding this dissertation.

2.1.1 Quantum Computing Model

For this dissertation, we use the classic quantum QRAM model, where a quantum machine is an auxiliary component that gets instructions from classic processors [Giovannetti et al., 2008]. At the core of this model is the "qubit" (quantum bit). Much like the classical bit it is the basic unit of information for quantum computing. Unlike a classical bit, however, it can exist as a linear combination of the zero and one state, called a "superposition". Hence a qubit ψ's state can be expressed as

$$\alpha\begin{pmatrix}1\\0\end{pmatrix}+\beta\begin{pmatrix}0\\1\end{pmatrix}, \forall \alpha,\beta \in \mathbb{C}, |\alpha|^2+|\beta|^2=1$$

Typically, we notate this using the "Dirac" or "bra-ket" notation where the basis states are notated as "ket 0" and "ket 1" respectively: $|0\rangle \coloneqq (1\ 0)^T, |1\rangle \coloneqq (0\ 1)^T$ [Dirac, 1939]. Further, for notational convenience bit-strings in kets are understood to be the Kronecker products of said kets: $|x_1 x_2 \cdots x_n\rangle \coloneqq \bigotimes_{i\in\{1,\cdots,n\}} |x_i\rangle$. "Bra"-states are defined as the adjoint (or conjugate-transpose) of ket-states: $\langle x| \coloneqq |x\rangle^\dagger$. The state of a single qubit $|\psi\rangle$ is thus a vector in a 2D complex vector space (specifically, a Hilbert space). This is often represented as the bloch sphere [Bloch, 1946].

Most commonly, today's quantum programs are represented as circuits. Circuits have a fixed number of qubits (quantum bits) and gates that operate on one or more qubits.

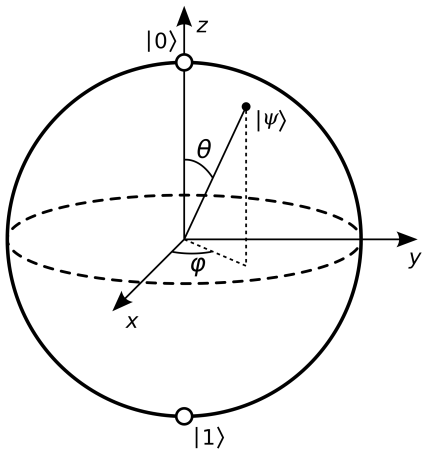


Figure 2.1: The bloch sphere: Showing all possible states of a qubit. (Image courtesy of Smite-Meister and wikimedia commons; used under Creative Commons)

These gates, such as the Hadamard, X, Z, and CNOT gates, manipulate the state vectors of the qubits. The Hadamard gate (H) is particularly important as it transforms the computational basis states from the "Z-basis" into the "X-basis" states, defined as: $|+\rangle := H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|-\rangle := H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. Gates, and hence circuits, mathematically are "unitary" matrices; a unitary matrix U has the property $U^\dagger = U^{-1}$. We shall break from this classic notion of representing quantum programs, however, and instead use the ZX-calculus.

2.1.2 ZX-calculus

The ZX-calculus, introduced by Coecke and Duncan in 2007, is a graphical calculus for quantum operations based on ZX-diagrams and an associated complete set of rewrite rules as shown by Jeandel et al. in 2020. Therefore, we can diagrammatically reason about any quantum circuit with the ZX-calculus using rewrite rules that allow any transformation. Based on this calculus, there have been numerous research papers, including PyZX an optimizing compiler by Kissinger and van de Wetering [2019]. Furthermore, there are numerous extensions to the ZX-calculus, such as the ZH-calculus by Backens and Kissinger [2018] or ZW-calculus by Hadzihasanovic [2015], to provide it with more

capabilities. While these developments are interesting, they will not be directly relevant to the remainder of this dissertation. The ZX-calculus is universal, meaning that it can represent any linear map (and hence any quantum circuit) as shown by Jeandel et al. [2020], Hadzihasanovic et al. [2018], Coecke and Kissinger [2017]. The reverse is not true, however: There exist ZX-diagrams that have no corresponding circuits, and hence extracting ZX-diagrams into quantum circuits proves to be a difficult problem, shown in Duncan et al. [2020]. Intuitively we can see this greater expressiveness, as circuits can represent any unitary matrix and unitary matrices are restricted to be square; linear maps do not have that requirement.

ZX-diagrams & Semantics

ZX-diagrams are a graphical representation of quantum operations. In this section, we will look at how to construct and interpret ZX-diagrams and at some analogies to quantum circuits. Our presentation draws on the 2020 survey by van de Wetering.

The core components of every ZX-diagrams are so-called "spiders," which are linear maps performing Z or X rotations. Each spider is represented as a dot as shown in Figure 2.2, with an angle α and n inputs and m outputs. If the spider in question is green, it is a "Z-spider," meaning a spider performing a Z rotation, and if it is red, it is an "X-Spider".[1] A Z-spider with n inputs, m outputs, and angle α is interpreted as the following $2^m \times 2^n$ matrix:

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

1. The specific shades of green and red are accessible.
See `https://zxcalculus.com/accessibility.html` for more details.

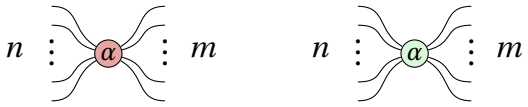


Figure 2.2: A red X and green Z-spider with n inputs, m outputs, and rotation angle α

Likewise, an X-Spider with n inputs, m outputs, and angle α is interpreted as

$$\underbrace{|+\rangle\cdots|+\rangle}_{n}\underbrace{\langle+|\cdots\langle+|}_{m}+e^{i\alpha}\underbrace{|-\rangle\cdots|-\rangle}_{n}\underbrace{\langle-|\cdots\langle-|}_{m}$$

Using these semantics, we can see that both an X- and Z-spider with one input and one output and no rotation, i.e., $n = m = 1, \alpha = 0$ is the identity matrix. The rotation angle of 0 is omitted when writing spiders, meaning spiders without explicit angles are assumed to have no rotation. As a shorthand, we write such spiders as a "wire," as shown in Figure 2.3.

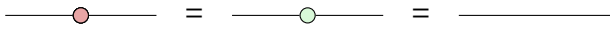


Figure 2.3: Identity removal: we see a spider with no rotation as a wire.

Other important constructions are so-called caps and cups. The cap is defined to be the matrix $(1,0,0,1)^T$, whereas the cup is defined to be $(1,0,0,1)$. They can be seen in Figure 2.4.

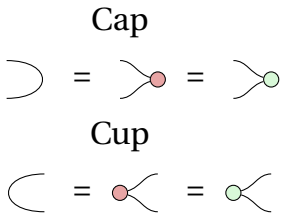


Figure 2.4: The constructions of cap and cup.

The notation of two wires intersecting indicates a SWAP and can be seen as a fundamental building block for ZX-diagrams, shown in Figure 2.5. Note that a basic SWAP only swaps two adjacent wires; we will later construct more general swaps. we will prove this using our established rules in Section 2.1.2.

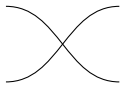


Figure 2.5: A ZX-calculus swap

Furthermore, we can have an empty ZX-diagram. Its semantics are the 1×1 identity matrix.

With these basic blocks established, we can now stack arbitrary ZX-diagrams. Given two ZX-diagrams with matrix interpretations A_1 and A_2, the interpretation of their stack is $A_1 \otimes A_2$, where $\otimes$ is the Kronecker product [Zehfuss, 1858].

We can compose two ZX-diagrams as long as the number of outputs of the first diagram equals the number of inputs of the second diagram. Given this restriction, for two diagrams with corresponding matrices A_1 and A_2, the semantics of composition (notated as $\leftrightarrow$) of such diagrams is defined as $A_2 \times A_1$.

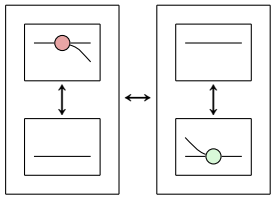


Figure 2.6: An example of a ZX-diagrams block form: A composition (represented by $\leftrightarrow$) of stacks (represented by $\updownarrow$) of spiders and wires. Notice that the number of wires on both sides of the composition are equal.

Figure 2.6 shows an example. Here we can see that we compose two diagrams, each stacking a wire and a spider. We can work out that the result is the matrix of CNOT gate using the stack and compose rules:

$$\left(\begin{bmatrix}1&0\\0&1\end{bmatrix} \otimes \left(|0\rangle\langle 0| \langle 0| + |1\rangle\langle 1| \langle 1||\right)\right) \times \left(\left(|+\rangle|+\rangle\langle +| + |-\rangle|-\rangle\langle -|\right) \otimes \begin{bmatrix}1&0\\0&1\end{bmatrix}\right) = \begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix}$$

This view of ZX-diagrams as a few basic building blocks that get composed and stacked shall henceforth be called the *block view*. We can view the same ZX-diagrams instead based on the adjacency of nodes. we refer to this view as the *graph-view*. The same diagram can be constructed, interpreted, and manipulated using both views interchangeably. Although, we shall stick to the semantics being defined through the block-view due to the inductive nature of said view. Furthermore, block view easily allows us to reason about altering sub-diagrams of a given ZX-diagrams. If there are two equal components, it follows immediately that these components can be used interchangeably for any composition or stacking. Because of these properties, block-view is very relevant for this dissertation.

In many papers, there is also the notation of stacking two nodes right above each other (in graph view) as shown in the CNOT example in Figure 2.7. This notation merely means that the direction of the connection between the two nodes is irrelevant, as both work out to have the same semantics. Since we define the semantics for ZX-diagrams through block view, which does not have such a construction, we adopt the convention of interpreting such stack diagrams as if we moved the lower node to the right, as shown on the right side of Figure 2.7.

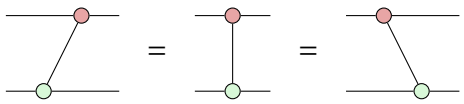


Figure 2.7: Shown are different representations of the CNOT diagram. Note that the stacked notation is only allowed if the ordering does not matter.

With arbitrary swaps in mind, any ZX-diagram can be deformed arbitrarily, as long as inputs and outputs are kept in order. Figure 2.8 illustrates this property: Two ZX-diagrams are equal up to deformation.

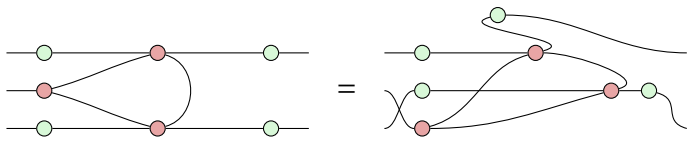


Figure 2.8: Two equal ZX-diagrams, where the right diagram is deformed. Note that all connections, inputs/outputs, and qubit order are maintained.

Scalars & Gates

All ZX-calculus diagrams with no inputs and no outputs represent scalars. These diagrams can be stacked onto any other diagrams in block view to multiply by a scalar constant. This fact becomes evident as all ZX-diagrams with no in/outputs have matrices representations with dimensions 1×1. Therefore, any such scalar constructions are of form $[c]_{1\times 1}, c \in \mathbb{C}$. Hence, when stacking a ZX-diagram with these scalar semantics onto a ZX-diagram with matrix semantics A, $c \cdot (I_{1\times 1}) \otimes A = c \cdot A$.

In Figure 2.9, we show some of the most important and common scalar construction. To multiply scalars, we can merely stack their diagrammatic representations. It is shown by van de Wetering [2020] that these three constructions and multiplication of them are, in fact, sufficient to construct any complex number.

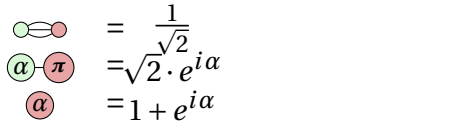


Figure 2.9: A complete set of scalars represented in the ZX-calculus.

We henceforth consider the equivalence relation "proportionality" of ZX-diagrams denoted by $\propto$ instead of equality. This means we can omit scalars that typically clutter up ZX-calculus proofs, as they can be regenerated.

$$\mathrm{ZX}_1 \propto \mathrm{ZX}_2 \Longleftrightarrow \exists c \in \mathbb{C}\setminus\{0\}, \mathrm{ZX}_1 = c \cdot \mathrm{ZX}_2$$

Gates We show a selection of standard gates translated into ZX-calculus in Figure 2.10.

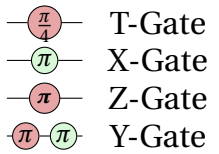


Figure 2.10: Common gates represented in the ZX-calculus.

The Hadamard gate has a shorthand notation "□" and its representation is shown in Figure 2.11.

Figure 2.11: The construction of a Hadamard gate in ZX-calculus. Note: The semantics of this diagram are equal up to a scalar to the Hadamard gate matrix.

Rules

This section outlines a selection of key rules to manipulate and simplify ZX-diagrams. For more straightforward interpretation, we will show all rules in graph view.

Color swapping We define a color-swapped ZX-diagrams as a ZX-diagrams with the same structure but with all Z-spiders being replaced by X-spiders (while keeping the angle) and vice versa. It can be shown that any that if a rule can be applied to a ZX-diagrams zx_1 transforming it into zx_2, then it can be applied to the color swapped diagram to zx_1 transforming it into the color swapped diagram of zx_2 [Coecke and Duncan, 2007]. With this in mind, we show any rule henceforth only for one color configuration, understanding that it applies to the color-swapped version.

Spider fusion/splitting One of the most important rules is that spiders connected by an arbitrary (non-zero) number of edges can be fused into a single node with the angles added. This rule is shown in Figure 2.12. Further, the reverse is true: any spider can be

split such that the two new spiders add to the original angle. A corollary of this is that spiders with 0 angle can be split off on any input or output, as long as they have the color of the original spider.

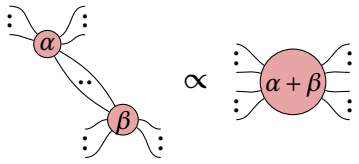


Figure 2.12: The spider fusion rule: Two connected nodes fused into a new node with added angles.

Self-loop removal Another simplification is that we can remove self-loops, as shown in Figure 2.13. Intuitively, this rule says that a node cannot provide more information about itself. A self-loop is an edge that goes from an edge to itself.

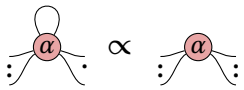


Figure 2.13: The self-loop removal rule: Edges from a node to itself are removed

Bi-algebra rule The bialgebra rule is a rule that, while not intuitive, is crucial for many ZX-calculus proofs as it allows for the rearranging of edges between nodes. The rule is shown in Figure 2.14.

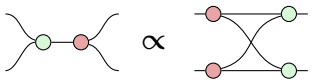


Figure 2.14: The bialgebra rule

Hopf rule The Hopf rule is, similar to the bialgebra rule, a rule that uses the fact that maximum information about the Z basis gives minimal information about the X basis. This rule allows us to disconnect specific nodes in the diagram instead of changing their connection. The Hopf rule is shown in Figure 2.15.

Figure 2.15: The Hopf rule

Bi-π rule For a given X spider, the spider is equal to itself with pi Z rotations on each side. Intuitively, this rule is true due to the orthogonal nature of the X and Z basis and the fact that we are performing in total a full rotation. Figure 2.16 shows this rule.

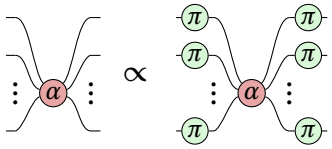


Figure 2.16: The bi-π rule

A corollary of this rule is the π-copy rule, as shown and derived in Figure 2.17.

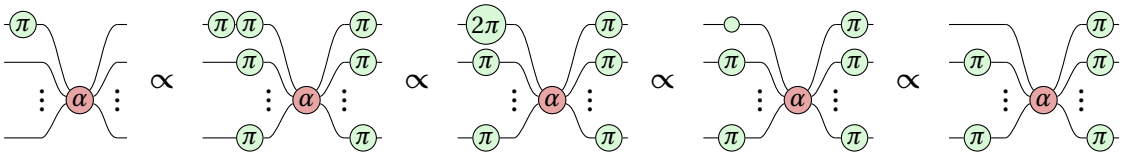


Figure 2.17: The π-copy rule derivation using the bi-π rule

Bi-Hadamard color changing For a given spider, one can swap the spider's color while keeping the angle by composing a stack of Hadamards to the in and outputs. This "bi-Hadamard" construction is shown in figure Figure 2.18.

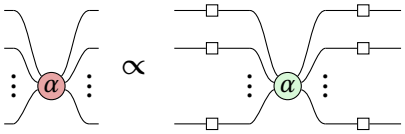


Figure 2.18: Swapping colors using a bi-Hadamard construction

Y-rule The Y-rule is an intriguing rule mainly used in Clifford diagrams. It is a rule that targets strict Clifford spiders and allows for manipulation of signs. Figure 2.19 shows this rule.

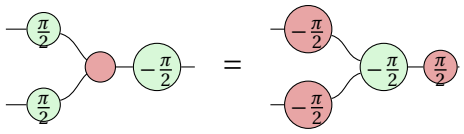
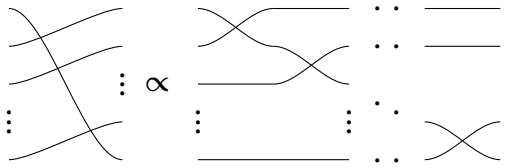


Figure 2.19: The Y-rule

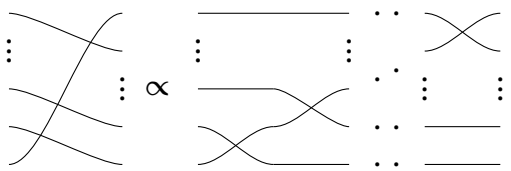
Constructions

Using the aforementioned rules and basic ZX-calculus facts, we show some other important basic ZX facts.

Arbitrary SWAPs To construct an arbitrary swap, we shall first build a construction to shift qubit 1 to n, whereby all other qubits are shifted up. We also build the inverse (i.e., shifting qubit n to 1). Figure 2.20 shows this construction. Note that in block-view, the only native swaps that exist swap two adjacent qubits.



Shifting qubit 1 to n



Shifting qubit n to 1

Figure 2.20: Shifting qubits. The base construction for arbitrary swaps.

By shifting qubit 1 to n using this construction, we shift qubit n to $n-1$. Hence, if we compose a qubit $n-1$ to 1 shift (stacked on a wire) to the 1 to n shift, we have constructed a gate that takes qubit 1 to n, n to 1 and leaves all others untouched, creating a SWAP of qubits 1 and n as shown in Figure 2.21 This is a swap of qubits 1 and n. We shall denote this swap as $\text{SWAP}_{1,n}$.

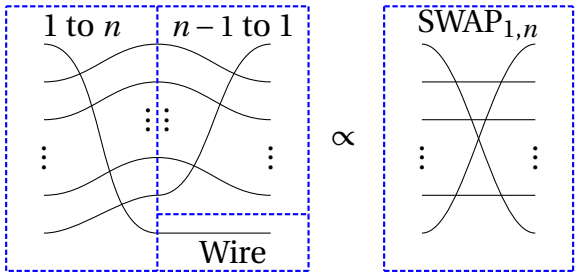


Figure 2.21: $\text{SWAP}_{1,n}$ construction with the individual components explicitly shown in block-view

To now construct a SWAP between n and $n+c$ in a diagram with m qubits, we build the following ZX-diagrams:

Stack n wire on top of $\text{SWAP}_{1,c}$ stacked on top of $m-(n+c+1)$ qubits. Visually, it is clear that this approach leaves the first n and any qubits after $n+c$ untouched; however, it swaps n with $n+c$. Such a construction shall be denoted $\text{SWAP}_{n,n+c}$.

Using arbitrary swaps and shifts, we can now interpret any wire crossing in graph-view with an equivalent block view.

SWAP through 3 CNOTs We will construct a swap from 3 CNOT gates in the following. This is a commonly used rule in quantum computing and a good example proof in ZX-calculus to see the rules we discussed in action. We will also proceed to prove the equivalence of this construction in graph view in Figure 2.22, loosely based on the proof found in van de Wetering [2020].

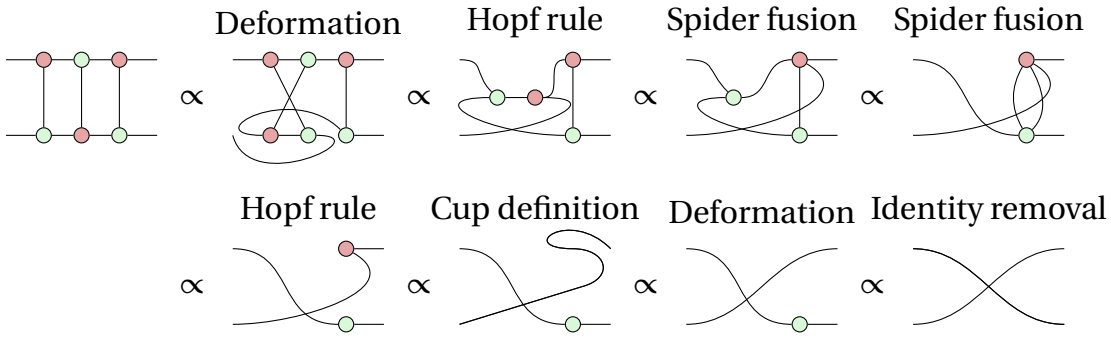
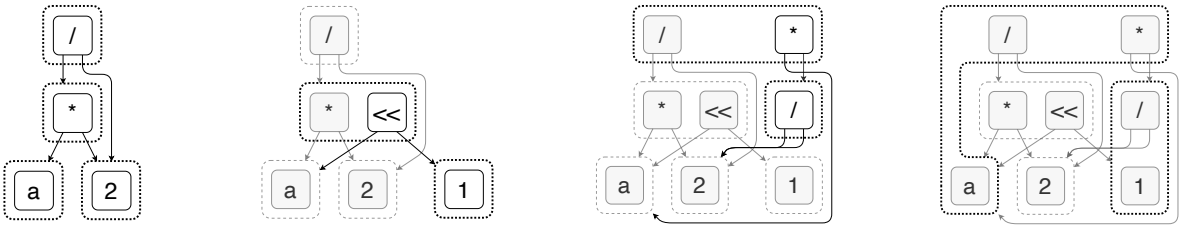


Figure 2.22: Graph view proof that 3 CNOTs is a SWAP gate



(a) Initial e-graph contains $(a \times 2)/2$.

(b) After applying rewrite $x \times 2 \rightarrow x \ll 1$.

(c) After applying rewrite $(x \times y)/z \rightarrow x \times (y/z)$.

(d) After applying rewrites $x/x \rightarrow 1$ and $1 \times x \rightarrow x$.

Figure 2.23: An e-graph consists of e-classes (dashed boxes) containing equivalent e-nodes (solid boxes). Edges connect e-nodes to their child e-classes. Additions and modifications are emphasized in black. Applying rewrites to an e-graph adds new e-nodes and edges, but nothing is removed. Expressions added by rewrites are merged with the matched e-class. In Figure 2.23d, the rewrites do not add any new nodes, only merge e-classes. The resulting e-graph has a cycle, representing infinitely many expressions: a, $a \times 1$, $a \times 1 \times 1$, and so on. This figure is taken from Willsey et al. [2021].

2.2 e-graph systems

e-graphs, first introduced by Nelson in 1980, are data structures that capture the equivalence of terms. In an e-graph the important concepts are e-nodes and e-classes. An e-node represents a term. e-nodes can have children as arguments. For example, $f(x, y)$ would generate the e-node f with children x and y. e-classes group e-nodes into equivalence classes. Any member of an e-class can be used instead of any other member. For example, if we consider synonyms to be equivalent, we could have an e-class: {`walk, stroll, trek`}. If we want to perform any operation on any member of the e-class it does not matter which member we choose as the *representative* of the e-class, we consider them equivalent. So let us say we have the term `Today I took a ____`. We can substitute in any of the e-class' members to capture the meaning of the sentence we want to create within our example.

Let us look at a more concrete example, taken from Willsey et al.'s 2021 paper, in Figure 2.23. We consider the initial term shown in Figure 2.23a. Here we see that each e-class, represented by the dotted lines, contains a single element. In Figure 2.23b we apply

a rewrite. Here we see that two new e-nodes `«(a,1)` and `1` are created. We also see that `«(a,1)` in the same e-class as `*(a,2)`. Also note that `/` from the top level e-class points to the newly expanded e-class, not an individual e-node, as the representative does not matter. In Figure 2.23c we see that through the rewrite there is a new member in the top-level e-class. Colloquially, we say the e-graph is equal to another term if the top-level e-class contains said term. Lastly, we apply another rule in Figure 2.23d and see that the e-graph is equal to `a`. This example also shows another great advantage for the intended goal of the proposed thesis. The order of rewrites does not matter, as the e-graph can find any representative of an e-class to perform a rewrite on and then unify the new term with the existing e-class.

2.2.1 e-graphs good (egg)

egg by Willsey et al. [2021] is an equality-saturation based e-graph tool that rekindled strong interest in the data-structure. In this dissertation we use egg, due to its good performance and developed proof explanation. Typically, e-graph systems use a union find data-structure to canonicalize each e-class and then use a matching algorithm to find rewrite candidates. In this approach, typically invariants are restored at every step, however, egg defers such rebuilding until an entire equality saturation step is completed. This allows for faster saturation and some work duplication avoidance to greatly speed up the equality saturation process. Further, at the same time a union find data-structure is used for proof explanation in egg.

From a developer egg requires instructions to perform such saturation. Most basically, it requires a term algebra or abstract syntax that represents the terms one wants to reason over. Note, egg does not support types, if a term can have children there are no restrictions on what the children can be. egg also requires so-called rewrites to be specified. A rewrite in egg allows for rewriting of one parameterized term into another under

conditions. Formally, if you have a language make up of n terms $L = (t_1, \ldots, t_n)$, where for each term has children given by $c(t_i) = [\ldots]$ with fixed arity. A *rewrite rule* (or simply, a rewrite) is defined as a four tuple $(l, r, C, \mathcal{V})$, commonly written as:

$$l \to r \quad \text{if} \quad C$$

Where:

1. $\mathcal{V}$ are variables that can be used in l, r and C.
2. l is the left-hand side, a pattern constructed from operators in L and variables in $\mathcal{V}$. This is the pattern that is searched for in the expression graph, with variables allowing any expression. All variables of $\mathcal{V}$ must be bound in l.
3. r is the right-hand side, also a pattern. This is the expression that replaces the LHS.
4. C is a set of **conditions** or predicates, $\{c_1, \ldots, c_k\}$. Each condition is a function $c_j : \sigma \to \texttt{boolean}$, where σ is a substitution for $\mathcal{V}$.

A rewrite rule is applied to an expression in an e-graph through a two-step process:

1. *Matching*: When looking at a term t, `egg` searches for a sub-expression s that matches the LHS pattern l. This matching process yields a substitution, σ for all variables in $\mathcal{V}$. For a match to be found, applying the substitution to the pattern must result in the sub-expression: $\sigma(l) = s$.
2. *Applying*: If a match is found with substitution σ, the conditions in C are evaluated using this substitution. If all conditions are met $(\bigwedge_i c_i(\sigma))$, the rewrite is executed by adding $\sigma(r)$, to the e-class of the original sub-expression s.

Consider the following example of arithmetic expressions over rational numbers, where $L = \mathbb{Q} \cup \{+, \cdot, -, \log\}$. We then consider the following rewrite rule:

$$\log(x \cdot y) \to \log(x) + \log(y) \quad \texttt{if} \quad \texttt{is\_positive}(x) \wedge \texttt{is\_positive}(y)$$

Here, $\mathcal{V} = \{x, y\}$. If we apply this to the expression $5 + \log(8 \cdot 2)$, the matching will find the sub-expression $s = \log(8 \cdot 2)$, where $\sigma = [x \to 8, y \to 2]$. The applier then checks the conditions, $c_1 = \texttt{is\_positive}(x)$ and $c_2 = \texttt{is\_positive}(y)$ and finds that $\bigwedge_{\{1,2\}} c_i(\sigma) = \texttt{true}$. Hence, it then adds $r(\sigma) = \log(8) + \log(2)$ to the e-class containing $\log(8 \cdot 2)$.

2.3 Verification

Before discussing our implementation, we want to give a brief background on verification. While we assume readers have decent familiarity with the space, we do believe setting the stage benefits the reader.

Henceforth, we define the word verification to mean formal verification using a general purpose *Calculus of Inductive Constructors* (CIC) [Pfenning and Paulin-Mohring, 1990] proof assistant. In this context, when we think about verification, we think about a proof engineer defining a specification of a problem and using the specification to prove facts about a solution to said problem. Specifically, use the CIC based proof assistant "Rocq prover" by The Coq Development Team [2023] (also known as just "Rocq" and formerly as "Coq"). In many ways, this method is very analogous to proofs on paper, with the difference being that the proof assistant verifies the proof. This verification comes with benefits and drawbacks. The great benefits to verification are that one can trust a formally verified proof much more, as the proof assistant does not allow hand-waving, and humans are notoriously bad at finding errors in proof by hand.[2] Further, one can ver-

2. Some mathematicians might disagree strongly with me on this point. However, we think this comes from a place of great overconfidence. Famous proofs have been shown to be wrong. Such as the Kapranov

ify the exact implementation of the software they have. Even if theory is perfectly sound, software developers naturally introduce bugs, such as in the infamous "Heartbleed" SSL vulnerability as published in CISA [2016]. Notably, miTLS, a verified implementation of SSL by Bhargavan et al. [2013], was unaffected.

This power, however, comes with great ~~responsibility~~ effort. Proving in a theorem prover is a science by itself. It is no wonder why many published papers "simply" formally verify known facts. But often the verification requires new approaches to existing proofs, significant engineering, and dedicated domain experts. In fact, *VyZX*, one of the works that are part of this dissertation, verifies the ZX-calculus. While novel in its approach, it does fundamentally prove facts proven before informally on paper. But for certain applications, where mistakes are to be avoided at all costs, this effort can pay off. It is our belief that given the noisy and expensive nature of running quantum programs, quantum computing is such an application.

2.3.1 Proof Automation

Because of the difficulty of proof in a proof assistant, proof engineers may want to automate repetitive steps or use proof search to save time and focus their efforts on problems that matter. Most proof assistants, like Rocq, have a built-in automatic brute force search that can be fed hints of which lemmas to try to work with, and a tactic language. Using such a tactic language, the proof engineer can specify rules based on the goal and context. For example, one could specify that if the goal is $x \leq y$ and there is a hypothesis $x < y$, then one should use the lemma that states $\forall ab, a < b \rightarrow a \leq b$. Of course, this is a very simple example and some have built entire algorithms to search for proof steps within these tactic languages, such as in our collaborators' work (Shah et al. [2024]). Proof engineers

and Voevodsky [1991] work that was proven incorrect by Simpson [1998], or the incorrect proof of the four color theorem by DeMorgan recounted by Biggs [1983] (the four color theorem was in-fact proven using a computer by Appel and Haken [1977] and later in Rocq by Gonthier et al. [2008]).

can also still remain confident in results as the resulting proof steps are checked by Rocq's proof checker. The main issue with these tactic languages is that they are hard to work with, often run very slowly, and in the case of Rocq are poorly documented.[3] In short, proof automation is either quite limited in its applicability or hard to develop and often fragile. It is our goal to bring a pragmatic approach to this area with the work we present in Chapter 4.

3. Side note: we believe this field would become a lot more accessible if writers of documentation for proof assistants don't assume they are writing for PhDs. If anyone working on a proof assistant reads this, we encourage you to write easy-to-follow, yet in-depth tutorials of such features.

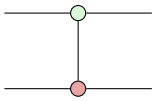
CHAPTER 3

VERIFYING THE ZX-CALCULUS

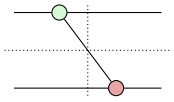
Typically, when verifying programs, we focus on languages that have easy operational semantics, such as declarative and functional languages. When verifying graphical languages, the naive approach is to use graph theory formalized in Rocq. Current formalizations of graphical concepts, however, typically rely on axiomatic data structures.[1] This approach creates the problem of not being able to directly extract code or increasing the size of the trusted code base from that of the core proof assistant. So how can verify a graphical language? Especially one where, like the ZX-calculus, we care about "only connectivity matters" and the connections between nodes? In the ZX-calculus's case, semantics are fundamentally assigned by topology. To assign these semantics, we ought to assign an ordering. For example, consider the ZX-diagram representing a CNOT gate in Chapter 3. In this example, we need to choose whether the red or green spider comes first to assign semantics, even if both diagrams are semantically equivalent. However, the construction of semantics is distinct.

In practice, we can view any ZX-diagram in terms of the box decomposition. This additional structure, however, can lead to some obfuscation. If we look at the boxed CNOT diagram in Figure 3.1b, we see that it is not immediately obvious anymore which boxes are next to each other. In later chapters, we explore strategies to deal with this issue, but for now we will work in this structure and look at the practical impact of it in proof. The great advantage for now is that we can explicitly construct matrix semantics in the Rocq matrix library `QuantumLib` [INQWIRE Developers, 2022]. By using this library, we keep the trusted code base of the Rocq kernel and Rocq real numbers from its standard library.

1. While there is no complete list of such projects, the interested reader can look at projects based on `https://github.com/coq-community/graph-theory`



(a) A simple graphical program, corresponding to a CNOT in the ZX-calculus.



(b) A CNOT adjusted for ordering.

Figure 3.1: The difference of traditional ZX-calculus representation and desugaring for semantic evaluation.

In addition to linear algebra semantics, *VyZX* is designed with consideration for building a proof library for verifying tools built on top of the ZX-calculus.

We have an expressive language that can represent all ZX-diagrams and allows for all ZX rules to be represented. We discuss this language in Section 3.1. In Section 3.3, we show how we can use *VyZX* for applications, specifically, to convert quantum circuits to ZX-diagrams and peephole optimizations. Of course, these applications are fully verified. The circuit translation also shows us how having underlying semantics in `QuantumLib` allows us to work with other verified quantum libraries. We also built *VyZX* to allow for rules to be parametric, meaning arbitrarily sized. With that we can perform proofs inductively based on number of inputs or outputs. We show this in Section 3.2. In Section 3.2, we also explore some automation strategies to simplify proof.

The presentation of this chapter will draw on our 2024 paper [Lehmann et al., 2024].

3.1 Graphical Languages in Rocq

As alluded to before, representing graphs in proof assistants based on the CIC, such as Rocq, is difficult as they are fundamentally based on working with inductive datatypes. To still work with the ZX-calculus, we look at it in its categorical interpretation. The ZX-calculus is fundamentally a symmetric monoidal category as described in Selinger [2010] amended with the Z and X spider generators. The exact category theory is not required to understand the rest of the dissertation, but the interested reader can look at our 2024

paper or at "ViCAR" [Shah et al., 2024], our generalization of *VyZX* to abstract categorical semantics for general applied categorical reasoning. For the readers who do not have a category theory background, the basic idea of a viewing the ZX-calculus as a symmetric monoidal category is as follows: We break down the elements of the ZX-calculus into constructors and assign them relative positions by having sequential and parallel composition. We show the constructors in Figure 4.1. With these constructors, we can represent any ZX-diagram. Before we discuss this further, we will introduce a more effective way to display *VyZX* diagrams.

Typical ZX-diagrams don't show the information added by *VyZX*, namely the explicit composition and stacking. Further, ZX-diagrams focus on connections between spiders, whereas *VyZX* focuses on the structure of the generators. To address this gap, Bhakti Shah and I built a visualizer for terms in *VyZX*, called "ZXViz." ZXViz integrates into VSCode through the Rocq language server to provide visualizations as the proof engineer works on ZX-calculus proofs. We discuss the implmentation later in Section 3.4.1. In Figure 3.3 we show the renderings of each block constructors using ZXViz.

There are a few other constructs we add for convenience or for expressivity. We add `nStack c` to stack a diagram c times on top of itself. This allows us to create common structures such as `nWire n` which is a shorthand for n wires. These common constructs are required for many rules in the ZX-calculus. Another important construct is `cast`, a common concept found in dependently typed systems, to prove equality between dependent type arguments. In our case we define `cast` with the following type:

```
cast (n m : ℕ) {n' m' : ℕ} (prfn : n=n') (prfm : m=m') (zx : ZX n' m') : ZX n m.
```

Given `nat` equality is decidable, this construct is also sound.

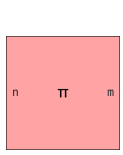
Typically, in the ZX-calculus we can consider equality up to a constant, non-zero factor, as discussed in Section 2.1.2; hence we will also define proportionality for *VyZX* as discussed.

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

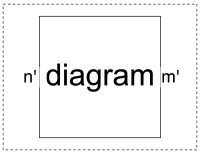
$$\frac{}{\texttt{Wire} : \texttt{ZX 1 1}} \qquad \frac{}{\texttt{Box} : \texttt{ZX 1 1}} \qquad \frac{}{\texttt{Swap} : \texttt{ZX 2 2}} \qquad \frac{}{\texttt{Empty} : \texttt{ZX 0 0}}$$

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

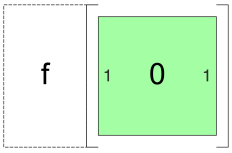
Figure 3.2: The inductive constructors for block representation ZX-diagrams



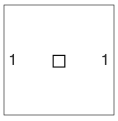
(a) A π X spider. *VyZX* term: `X n m PI`



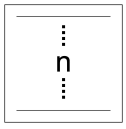
(b) A cast of `diagram` to `n'` inputs and `m'` outputs. *VyZX* term: `cast n m diagram`



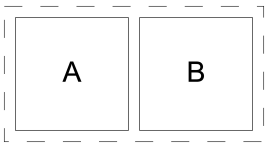
(c) A function `f` applied to a Z-spider. *VyZX* term: `f (Z 1 1 0)`



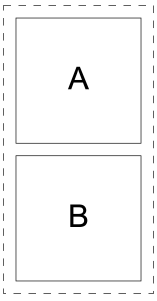
(d) Hadamard box. *VyZX* term: □



(e) `n_wire n`



(f) A composed with B. *VyZX* term: `A ↔ B`

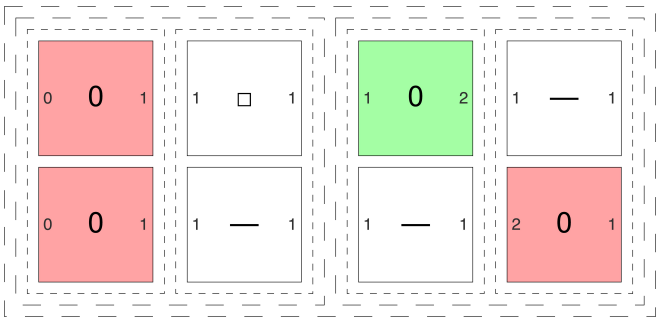


(g) A stacked with B. *VyZX* term: `A ↕ B`

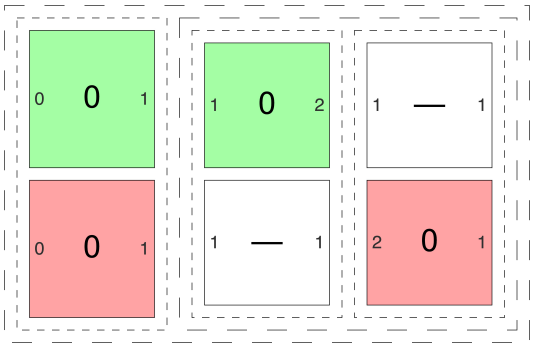
Figure 3.3: *VyZX* constructors and functions visualized using ZXViz. See Figure 4.1 for details on the constructors.

3.2 Graphical proofs

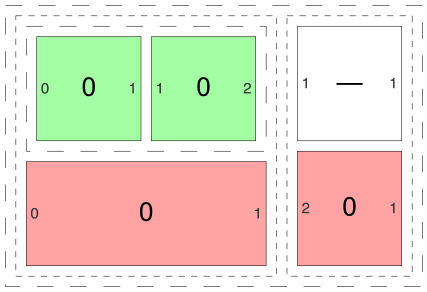
When discussing graphical proof, we specifically mean proofs that directly manipulate ZX-diagrams, instead of reasoning about the underlying semantics. It is one of the core design goals of *VyZX* to enable proof engineers to solely prove ZX-calculus problems via graphical reasoning, without having to appeal to the underlying matrix semantics. To this end, we prove a complete set of rewrite rules for the ZX-calculus, including all rules shown in Section 2.1.2.



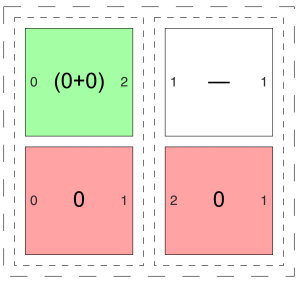
(a) `bell_state_prep_correct` lemma



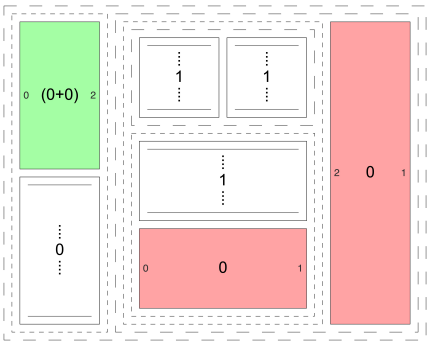
(b) Reassociate and color-swap top-left node



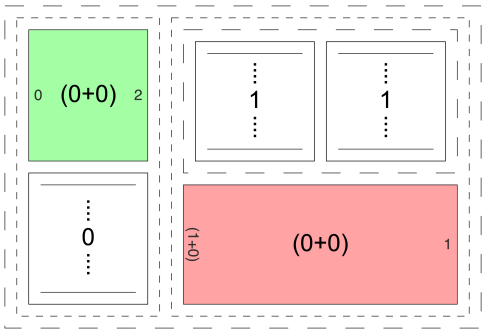
(c) Reassociate for fusion



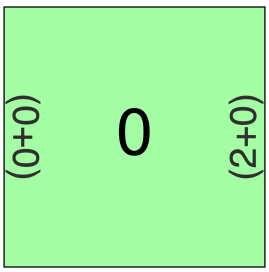
(d) Fuse top-left spiders



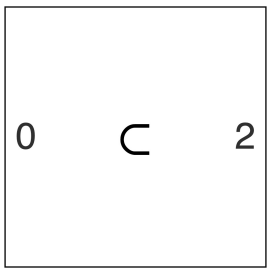
(e) Add auxiliary wires and reassociate



(f) Fuse bottom-right spiders



(g) Remove auxiliary wires and trivial spider

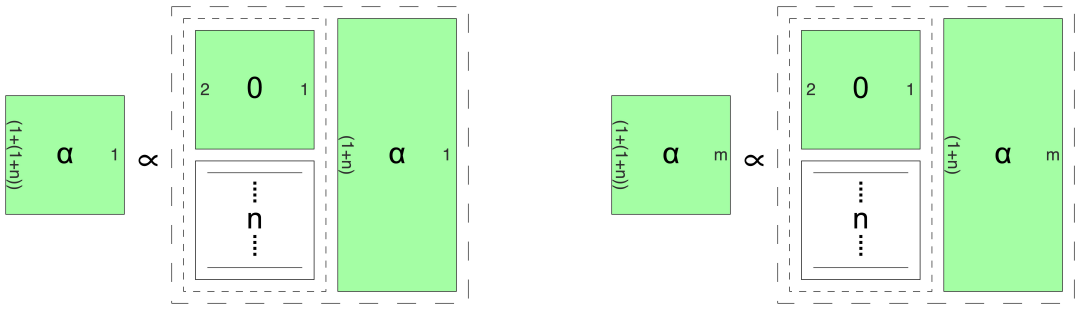


(h) Convert equivalent spider into cap

Figure 3.4: Proof of the correctness of bell state preparation in *VyZX*.

As an example of what this means in practice for the proof engineer, we show the proof that the bell state preparation circuit is equivalent to a cap in ZX-calculus. First, we transform the standard Bell pair circuit into a ZX-diagram by translating gates as shown in Section 3.3. Figure 3.4a 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 3.4b. After that, we reassociate (see Figure 3.4c) and fuse the two Zspiders in Figure 3.4d. The next step is to reassociate (Figure 3.4e) and to fuse the X spiders in Figure 3.4f. Lastly, in Figure 3.4g, we simplify the resulting diagram and then convert the spider into a cap in Figure 3.4h.

However, there are more considerations in our proof system: As is natural in proof assistants, we would like to reason inductively over the number of inputs or outputs of a given spider. For this, we defined a way to inductively reason about spider input/output growth, as shown in Figure 3.5. This allows us to deconstruct any arbitrary sized rule into an inductive problem and prove up from the base case. For example, spider fusion, which is defined on $n > 0$ connections between the spiders, can be broken down into the base case $n = 1$ and then proven inductively. Practically, many rules that are directly derived from semantics are provable inductively: Prove the base case with matrix semantics, and



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

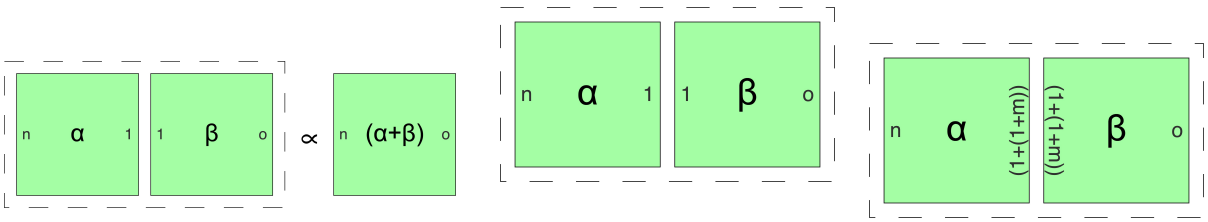
Figure 3.5: `grow_Z_top_left` is built by induction using the base lemma `grow_Z_left_2_1` which is proven directly via the matrix semantics.

then use the inductive hypothesis to graphically rewrite the induction step. This proof is, like the bell pair proof, from our work on *VyZX* Lehmann et al. [2024].

Using these facts, we can prove spider fusion properties inductively, such as the limited property of spider fusion of fully connected spider, shown in Figure 3.6a. When looking at the base ($m = 1$) case in Figure 3.6b, we see that our base case corresponds to a lemma that we previously proved via an appeal to the underlying semantics. In the $m+1$ case (Figure 3.6c), we aim to fuse the top two wires and reduce the goal to our inductive hypothesis. As shown in Figure 3.6d, we first expand both spiders to have one input and two outputs, or two inputs and one output spiders emerging from the top. We then reassociate in Figure 3.6e. Then, we fuse the expanded spiders in Figure 3.6f. We use automation to clean up the diagram; we show the state in Figure 3.6g. Lastly, we complete the proof using the inductive hypothesis in Figure 3.6h.

3.3 Applications: Ingestion & Peephole Optimization

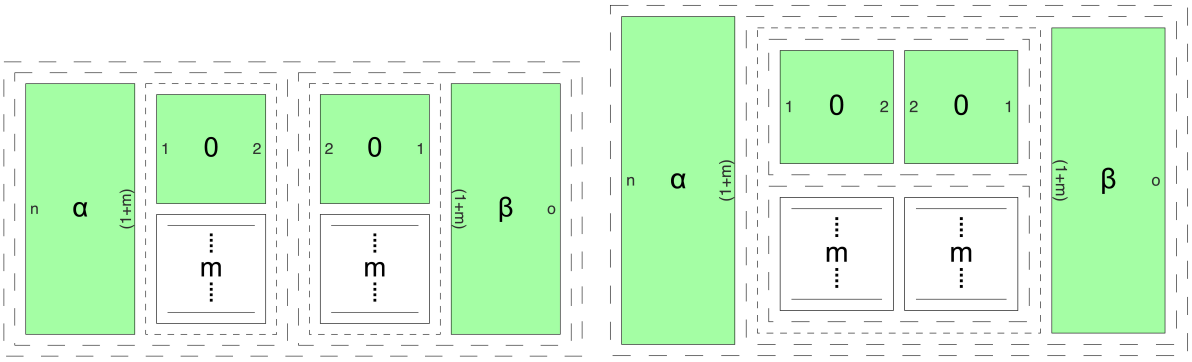
Quantum programs are usually represented using the circuit model popularized by Deutsch [1989], Nielsen and Chuang [2010b]. Hietala et al. [2021a]'s SQIR embeds quantum circuits in Rocq, with `QuantumLib` matrix semantics [INQWIRE Developers, 2022]. Fundamentally, its representation of circuits is an 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 unitary (i.e., 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: ZX-diagrams have n inputs and m outputs for any n or m. Moreover, SQIR circuit components explicitly specify gate locations on numbered qubits instead of stacking diagrams.



(a) `Z_absolute_fusion` lemma

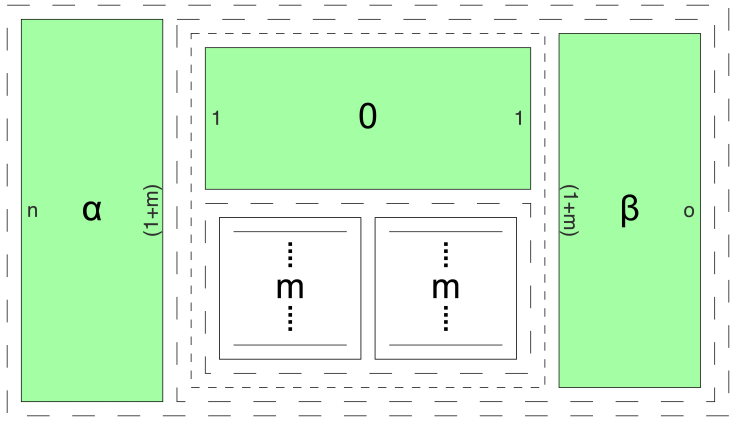
(b) Base case proven separately

(c) Inductive case

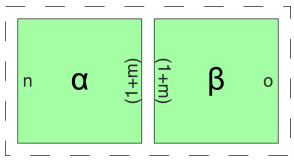


(d) Inductive case after expansion

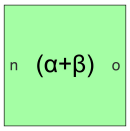
(e) Reassociating subdiagrams



(f) Fusing upper spiders

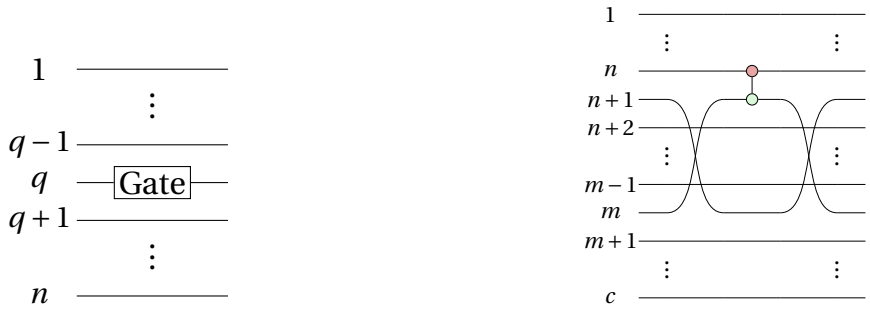


(g) Re-shrinking the graph



(h) Applying the inductive hypothesis

Figure 3.6: Proof of `Z_absolute_fusion`.



(a) Ingestion of a one qubit gate acting on qubit q into a *VyZX* diagram

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

Figure 3.7: Construction of SQIR operations in *VyZX*

Ingestion is the process of converting a SQIR circuit to a *VyZX* diagram. 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 gate application and then compose them appropriately.

We can assume that single-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 single-qubit gates, we define constructions `ZX_H : ZX 1 1`, `ZX_X : ZX 1 1`, and `ZX_Rz : ℝ → ZX 1 1` that translate gates to the corresponding ZX-diagrams. To include explicit circuit size, we define padding functions, that 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 3.7a shows the result of said transformation. To translate CNOT gates, we need to be more thoughtful. Since we cannot 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 3.7a. To allow for

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. [2021c], Nam et al. [2018b].

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 permutation is not trivially accomplished with the basic swap operations, as they merely swap adjacent qubits. 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 3.7b. 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: Rocq can automatically apply rules using `auto` to deal with composition and then singular gate translations until a SQIR circuit has been fully unfolded into a *VyZX* diagram.

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`, *VyZX* 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 *VyZX* by verifying peephole optimizations from Nam et al. [2018b]'s optimizer which are previously verified in the VOQC optimizer by Hietala et al. [2021c].

We use automation to convert each circuit into a ZX-diagram and then proceed to prove equivalences using *VyZX*'s rules. In *VyZX*, 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 *VyZX*'s approach, we are able to use the fact that some of these rules are

corollaries of each other, as they are equivalent up to transposes, complex conjugates, or color swapped versions of each other. This is something that is easy to see diagrammatically, but less obvious in a circuit representation or the textual representation of ZX-diagrams. By implementing these optimizations, we show the flexibility *VyZX* for reasoning about quantum systems and set the foundations for building a verified optimizer in *VyZX*.

3.4 Challenges in graphical proof

*3.4.1 Visualization: ZXV*IZ

Graphical proof is not textual, as the name suggests. Rocq is a fully textual language, however. To illustrate the issue with textual representations of graphical languages, try to parse the following expression: `– ↕ Z n (S (S m)) α ↔ (⊃ ↕ n_wire S m) ↔ (Z 1 2 0 ↕ n_wire m) ∝ – ↕ Z n (S (S (S m))) α ↔ (⊃ ↕ n_wire S (S m))`. We wager you did not find that as easy as reading the ZX-diagrams we showed earlier. In intermediate steps of proof this is a usual level of expression complexity and having to parse these expressions causes significant mental overhead. To bridge this gap, we built the aforementioned visualizer "ZXVIZ".

It is built to integrate with `coq-lsp` [Gallego Arias et al., 2023] as a VSCode plugin. Specifically, it renders all expressions it detects to be *VyZX* expressions in a window, akin to proof view. Every time you update the proof view, the plugin renders the new goal expression. Render times are typically <100ms, meaning it does not slow down the proof workflow. We show what this looks like to a user in Figure 3.9.

ZXVIZ automatically retrieves the expressions including all notations from coq-lsp. Then the expression gets parsed. For each node in the resulting AST, a visual box at the minimum size is created. A second pass then checks that there are no instances of height/width mismatch at compose/stack. For example, `(Z 2 1 0) ↕ (n_wire nIn) ↔ (Z (S`

`nIn`) `nOut` α) is rendered as shown in Figure 3.10, where the Z-spider with rotation α is visually stretched for easier readability. Notice also, that ZXVIZ's diagrams do not look like the canonical ZX-diagrams from Section 2.1.2, as they do not have connected nodes but rather building blocks in boxes representing composition and stacking. The reason we chose this design is to communicate the structure of the diagram effectively. When proving, structure matters, and hence it needs to be communicated. In practice, these diagrams have proven very useful. We do not have a large enough user-base of *Vy*ZX to conduct a formal user study examining the benefit of using ZXVIZ over textual representation. Anecdotally, however, ZXVIZ has sped up proving significantly and reduced the cognitive overhead of rendering the diagrams in our heads. New students on-boarded into *Vy*ZX also find it a lot more intuitive to prove with ZXVIZ. We encourage anyone trying to work with a graphical language in a proof assistant to build a visualization. In general, we claim that visual proofs are easier to comprehend; there is a reason we all sketch in our notebooks while proving on paper. We see more visualization, like in Nawrocki et al. [2023]'s Proof Widgets in the Lean 4 proof assistant by Moura and Ullrich [2021], as a very promising research direction to increase proof assistant adoption.

3.4.2 Associativity

With all these proof tools in place, we show that we can prove various ZX-calculus-based propositions. All the proofs we showed, however, still leave one big pain point: we have to manipulate the structure of the diagrams. Specifically, we often wish to apply lemmas to a sub-diagram deeply embedded in a larger diagram.

Manually, this rewriting would require a significant amount of proof overhead. The proof engineer would be writing a lot of lines of proof code that are semantically irrelevant and merely change the associativity structure of diagrams. To measure how big this issue is in practice, we take basic counts of rewrites we consider to be purely structural:

ones that deal with associativity, commutativity, casting, and other simplification [Lehmann et al., 2024]. We found that 27% of all tactic applications within *VyZX* proofs are to manipulate associativity. Of those, 24% (i.e., 7% of overall statements) are repeated applications (we consider Ltac tactic invocations to be single statements). Given that many *VyZX* proofs are not diagrammatic but instead do direct linear-algebraic reasoning that does not count towards the structural rewrite count, we can start to see the practical costs of having to manually associate diagrams in our proof development.

To bridge this gap, we will spend the further parts of the thesis implementing automation options to bridge this gap. We build a system using e-graphs that automatically completes such proofs for a user.

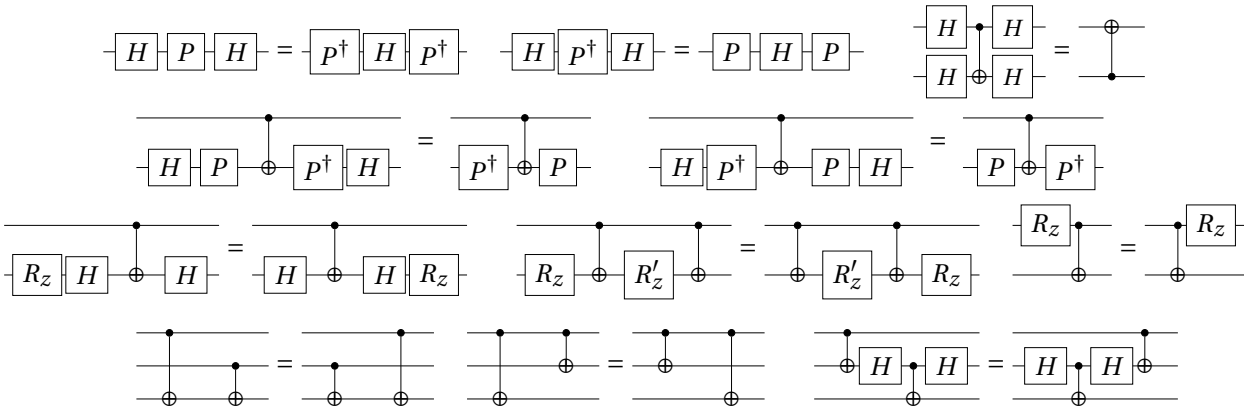


Figure 3.8: Quantum circuit peephole optimizations from Figs 4/5 in Nam et al. [2018b]. The first two lines are Hadamard reductions, where the goal is to reduce single gate count. The last two are to reorganize CNOT gates so that a later optimization pass can detect larger CNOT structures.

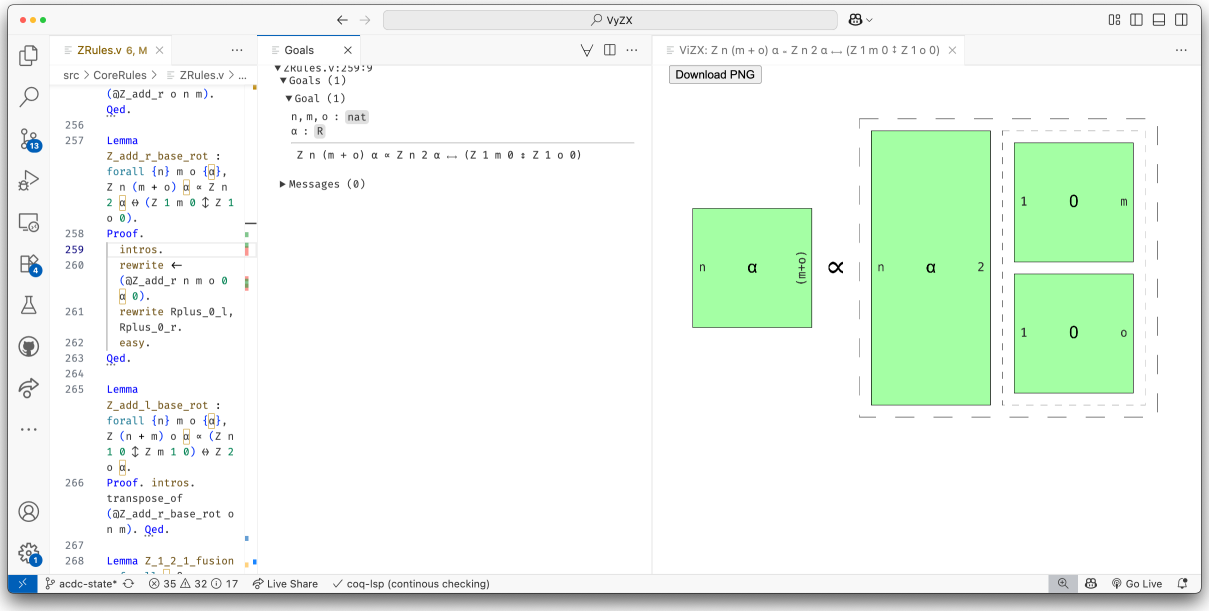


Figure 3.9: VSCode with ZXVIZ enabled.

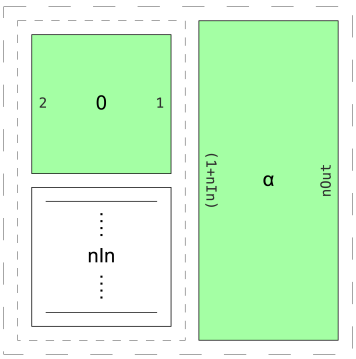


Figure 3.10: (Z 2 1 0) ↕ (n_wire nIn) ↔ (Z (S nIn) nOut α) without height mismatch of elements at the compose boundary.

CHAPTER 4

E-GRAPH-BASED PROOF AUTOMATION

This chapter discusses an extension to the work outlined previously. *VyZX* made it clear to us that automation in Rocq is clearly lacking. Many Rocq projects, including *VyZX*, work on inductive datatypes with an equivalence relation over them. We look further into bridging the automation gap informed by the structural problem we face in *VyZX*. It is our goal to further find practical insight about classes of problems that can be automated and the interplay of modern e-graph system together with Rocq. We build a system that can reach beyond solving associativity and distributivity problems by building a full domain-specific automatic rewrite engine.

4.1 Domain-Specific Proof Generation for *VyZX*

In this section, we discuss how we solve the issue of automatically resolving of structural equalities in *VyZX*. We use this domain as a showcase of using external e-graph solvers to solve domain-specific problems within Rocq. To tackle this problem, we need to be able to represent *VyZX* ZX-diagrams within an e-graph system and represent rewrite rules along with preconditions. For this, we make extensive use of the definitions we already established, such as the type definition found in Figure 4.1. Naïvely, we can simply translate the *VyZX* type into an algebraic, non-dependent data type, ignoring dependent type constraints. Unfortunately, this approach is insufficient. One can easily craft a malformed diagram by composing a wire and an empty diagram, as there are no restrictions on composition. So we need to have the constraints imposed by dependent types to prevent such diagrams from occurring. This problem is a little trickier than it seems on the surface, given that none of the current e-graph solvers natively support dependent types. In fact, egg has no notion of types at all. This means we could also stack a natural number

on a spider. Luckily, we can make do without a rich type system. All problems we work with come from a Rocq context. This means that at the time our system starts processing them, they respect all type constraints. Thus, we simply need to preserve the invariant through the computation on the e-graph level. We shall keep this in mind throughout the following discussion.

4.1.1 Representing an inductive language in `egg`

Let us start translating *VyZX* into `egg`.

We do this in two steps. First, we translate to an intermediate representation that faithfully represents all type constraints, except dependent type constraints, shown in Figure 4.1. Effectively, this captures the syntactic meaning of *VyZX* expressions. Henceforth, we call this representation the non-dependent IR (NDIR).

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

$$\frac{}{\texttt{Wire} : \texttt{NDZX}} \qquad \frac{}{\texttt{Box} : \texttt{NDZX}} \qquad \frac{}{\texttt{Swap} : \texttt{NDZX}} \qquad \frac{}{\texttt{Empty} : \texttt{NDZX}}$$

$$\frac{\texttt{zx}_0 : \texttt{NDZX} \qquad \texttt{zx}_1 : \texttt{NDZX}}{\texttt{Compose zx}_0\ \texttt{zx}_1 : \texttt{NDZX}} \qquad \frac{\texttt{zx}_0 : \texttt{NDZX} \qquad \texttt{zx}_1 : \texttt{NDZX}}{\texttt{Stack zx}_0\ \texttt{zx}_1 : \texttt{NDZX}}$$

Figure 4.1: Non-dependent IR

Second, we have an IR that is directly used by `egg` that is weakly typed. We shall henceforth call this IR weakly-typed IR (WTIR). Here, we include all constructs NDIR terms along with all constructs building the dependent type arguments of an original ZX diagram, just all into one type. We also convert from *VyZX* expressions to NDIR to WTIR and back to translate to an from e-graph level. Converting from WTIR to NDIR is inherently not always possible as we could for example compose a spider with the natural number zero. In addition to the constructs we show, there are some additional constructs we would like to add in our datatype that aren't yet naturally captured.

Formally, we define translation functions τ_N, τ_W, and τ.

$$\tau_N : \texttt{ZX n m} \to \texttt{NDIR} = \text{id}$$

$$\tau_W : \texttt{NDIR} \to \texttt{WTIR} = \text{id}$$

$$\tau : \texttt{ZX n m} \to \texttt{WTIR} = \tau_W \circ \tau_N$$

Pseudo-inductive constructs & functions In addition to the core inductive constructors, as shown in Figure 4.1, we have pseudo-inductive constructs in `egg` that behave like they are part of the basic inductive datatype but technically are not. In Rocq most of these constructs are – in fact – functions. One such example is `cast`; if you recall, `cast` in *VyZX* takes a ZX diagram of certain dimensions and changes it to a new set of dimensions. So far, we effectively treated it as if it were part of the core inductive constructs as one could conceivably have included it as an inductive constructor.[1] We include `cast` as a "pseudo-inductive" construct in our `egg` representation. For the sake of the e-graphs `cast` follows all rules of an inductive construct. This is in so far as that it behaves like an inductive constructor for rewrite rules and composition, and has a parametric morphism, allowing for the inner ZX-diagram being rewritten.

Similar to pseudo-inductive constructs, we need to deal with functions. In the most trivial case, we consider a function to be a black-box. For example, if we have a function from type $\texttt{t}_1$ to ZX-diagrams, where $\texttt{t}_1$ is neither a ZX-diagram nor natural number, we consider each argument of the function a variable of type `nat`. Since there are no simplification constructs, it will be treated as effectively a constant throughout the entire computation. We can also have lemmas that rewrite them. `egg` has no understanding of simplification of $\texttt{t}_1$, so this is a conservative approximation. Often we want a semantic understanding of functions to simplify them. For example, many functions exist that de-

1. In Rocq this would have not been a fruitful approach as keeping it a function allows us to reason over it more effectively when it comes to destructuring the ZX type.

fine some kind of structure of natural numbers, specifically many fixpoints. For example, we often use `n_wire` : $n \to$ `ZX` $n\ n$, where

$$\texttt{n\_wire}\ n = \begin{cases} \texttt{Empty}, & n = 0 \\ \texttt{Wire}\ \updownarrow\ \texttt{n\_wire}\ n', & n = S\ n' \end{cases}$$

Effectively we can see `n_wire` as a pseudo-inductive structure, too. We can destructure this function over the natural number arguments. The user provides the simplification instructions over the arguments. This is important, as Rocq sometimes triggers rewrites with implicit simplifications, as a function based on a match is simplified. As of such, we want users to input any match functions as pseudo-inductives where the e-graph can simplify structures. For all functions without simplification rules that we encounter, we set them to `Opaque` for the duration of the proof to avoid any incorrect simplification.

Similarly, there are some functions that have ZX-diagrams as their parameters. For example, `transpose` in Rocq has type `ZX` $n\ m \to$ `ZX` $m\ n$. To mimic these functions working like an inductive constructor, we must assign dependent types to them. For this, we consider the arguments that natural numbers and ZX-diagrams. If the output type dependent depends directly on a natural number parameter, we create a rewrite for `egg` from that dependent parameter (we discuss this idea in more detail in the following).

4.1.2 Converting Rocq lemmas to `egg`

We implicitly discussed simplification lemmas or using other *VyZX* lemmas in `egg`. So how do we convert a lemma from Rocq into a format that works with our WTIR while keeping all constraints? Let us first discuss the easy case, where dependent types are not relevant. Say we have the lemma that says stacking an empty diagram on any ZX-diagram does not change semantics: $\forall$ `n m (zx : ZX n m), zx` $\propto$ `⦰` $\updownarrow$ `zx`. The way we handle an expression

is we split it into hypotheses and statements. In our case, `n, m`, and `zx` are our hypotheses and `zx ∝ ⦰ ↕ zx` is our statement. In practice, we view hypotheses as binders, specifically binding variables to types. So in our case, we take note that `zx` is bound to the type `ZX n m`. We then look at our statement and translate it down into NDIR. While in NDIR, we look for any variables that are bound by one of our hypotheses. Now in this case we only have `zx`, but we notice there is no `Compose` involved, and hence we don't need to create any constraints. We shall discuss building these constraints after we understand the general translation.

While in NDIR, we take all variables bound by type hypotheses and mark them as variables for `egg` and WTIR. `egg` has one more restriction we must contend with when translating rewrite: Any variables in a WTIR statement must have all variables bound on the left side of a rewrite. This is the case for most lemmas naturally. For example, to be able to rewrite a lemma `∀ (zx : ZX 0 0), ⦰ ∝ zx` in Rocq in the forward direction, we would instantiate an e-variable for `zx`, as we would need to oracle the ZX-diagram we would want to add onto the empty diagram. `egg` cannot do this, therefore we skip any of these rewrite altogether. In our entire project, there are only a handful of erewrites in the context of ZX-diagrams. Then we simply strip out any typing information and lower to WTIR.

Handling dependent types To handle dependent types, we must augment our NDIR and WTIR. We introduce two constructs `Dep1` and `Dep2`, which take an argument (a ZX-diagram term in case of NDIR). These constructs are responsible for calculating the dependent type information for a given ZX-diagram. We then replace any references to the dependent type arguments with the respective `Dep1`/`Dep2` constructs. We then also generate constraints based on `Composes`. For example, for the lemma following lemma, we require that `Dep2(zx1) = Dep1(zx2 ↔ zx3) ∧ Dep2(zx2) = Dep1(zx3)` for the forward rewrite.

`compose_assoc : ∀ n m0 m1 o (zx1 : ZX n m0) (zx2 : ZX m0 m1) (zx3 : ZX m1 o), zx1 ↔ zx2 ↔ zx3 ∝ zx1 ↔`

(zx2 ↔ zx3) Now given that the left side of the rewrite maintains all dependent type constraints, we know that the rewrite is sound, given that we ingest proven lemmas from Rocq. In a way, this flips the problem on its head: We compute the dependent type information and check that it is maintained as a result of any rewrite in egg.

Further, if we have a lemma like `n_wire_removal_l : ∀ n nOut (zx : ZX n nOut), n_wire n ↔ zx ∝ zx`, we change the lemma to be represented as `n_wire (Dep1 zx) ∝ zx` in egg. In this case we recognize, similar to the function constructs we discussed that n is bound to zx and appears in the actual equality statement. Therefore, we replace it with the appropriate dependent type computation.

Overall, when converting a lemma we

1. Strips dependent types, as discussed in WTIR.
2. Consider if any parameters are ZX-diagrams.
3. For all parameters that are ZX-diagrams, replace the bound dimension variables for such diagrams with the dependent type computation in the statement.
4. If two ZX parameters share dependent type variables, we introduce conditions for the lemma to respect the implicit dependent type equalities.

With this system, we can ensure only well-typed statements occur in our computation.

Formally, this means for the parameter set $\mathscr{P} = (p_1, ..., p_n)$, where each $p_i = (n_i, t_i)$ with n_i being the name and t_i being the type. With this parameter set, we build the following substitution σ_{dep} for replacing dependent type arguments:

$$\sigma_{\text{dep}}(P) = \begin{cases} [], & P = \emptyset \\ [n \mapsto \texttt{Dep1}(zx), m \mapsto \texttt{Dep2}(zx)] \cup \sigma_{\text{dep}}(P'), & P = \{(zx, \texttt{ZX n m})\} \cup P' \\ \sigma_{\text{dep}}(P'), & P = \{(n, t)\} \cup P' \end{cases}$$

To build constraints, we build the following functions: $\phi_{\text{dep}}(P)$ the overall constraint function. First, we define a helper function $\mathscr{C}(P,n)$ that finds all instances of parameters P that have the dependent type variable n, along with an index $i \in \{1,2\}$ to indicate if it is the first or second argument:

$$\mathscr{C}(P,n) = \begin{cases} \emptyset, & P = \emptyset \\ \{(zx,1)\} \cup \mathscr{C}(P',n), & P = \{(zx, \texttt{ZX n m})\} \cup P' \\ \{(zx,2)\} \cup \mathscr{C}(P',n), & P = \{(zx, \texttt{ZX m n})\} \cup P' \\ \mathscr{C}(P',n), & P = \{(n,t)\} \cup P' \end{cases}$$

We then build the constraint set $\phi_{\text{dep}}(P)$ as follows:

$$\phi_{\text{dep}}(P) = \begin{cases} \emptyset, & P = \emptyset \\ \{\texttt{Dep1}(zx) = \texttt{Dep}i(c) | \forall (c,i) \in \mathscr{C}(P',n)\} \cup & \\ \{\texttt{Dep2}(zx) = \texttt{Dep}i(c) | \forall (c,i) \in \mathscr{C}(P',m)\} \cup & \\ \phi_{\text{dep}}(P'), & P = \{(zx, \texttt{ZX n m})\} \cup P' \\ \phi_{\text{dep}}(P'), & P = \{(n,t)\} \cup P' \end{cases}$$

If we then have a lemma `l : ∀ p1 p2 ... pn, l ∝ r`, when generate the rewrite with $P = \{p_1, \ldots p_n\}$ as follows:

$$\sigma_{\text{dep}}(P)(\tau(l)) \rightarrow \sigma_{\text{dep}}(P)(\tau(r)) \texttt{ if } \phi_{\text{dep}}(P)$$

4.1.3 Converting Rocq problems to `egg`

Converting a *VyZX* proportionality problem to an `egg` problem basically works by treating it as a *VyZX* lemma and converting it into WTIR. So we convert the Rocq expression into a NDIR and later WTIR term. Then we create `egg` nodes for both the left and right side

using τ. With our loaded rule set, we will have egg run equality saturation. One addition we have to make, however, is adding the dependent type information from the context. This is needed to be able to properly compute dependent type expressions. Further, it immediately relates expressions that use dependent types to each other. For example, if there is a problem $\forall$ n :$\mathbb{N}$ (zx : ZX n n), n_wire n $\leftrightarrow$ zx $\propto$ zx, we create rewrites Dep1(zx) $\mapsto$ n, Dep2(zx) $\mapsto$ n. This allows us to create sufficient dependent type information in egg for the rewrite from the lemma n_wire_removal_l : n_wire (Dep1 zx) $\leftrightarrow$ zx $\propto$ zx to trigger within egg, as it has the implicit precondition that Dep1(zx) = n. Formally, we create the following set of rewrites to create the appropriate dependent type hypotheses:

$$\{\texttt{Dep}i(zx_1) \rightarrow \texttt{Dep}j(zx_2) | \texttt{Dep}i(zx_1) = \texttt{Dep}j(zx_2) \in \phi_{\text{dep}}(P)\}$$

If we can saturate successfully, we move on to create a proof explanation and convert it to a Rocq proof.

4.1.4 *Converting egg proofs to Rocq proofs*

A proof explanation in egg is structurally quite different from a Rocq proof. In Rocq, a proof rewrite line typically has a lemma with a set of arguments. In egg a proof step has the name of the lemma and the before state. This information is insufficient in Rocq, as if there are multiple possibilities for a lemma to match in a statement, Rocq's heuristic will find the first substructure that matches with the lemma and rewrites all instances of it. egg on the other hand, may have matched a different substructure. For example, if you are solving $(a+b)+(c+d) = (b+a)+(d+c)$, the first two Rocq proof lines could be rewrite (Nat.add_comm a b) and rewrite (Nat.add_comm c d), where egg would tell you add_assoc from $(a+b)+(c+d)$, followed by add_assoc from $(b+a)+(c+d)$, with final state $(b+a)+(d+c)$.

This means we need to find parameters for Rocq! For this, we need to compare the before and after for an egg rewrite. Notice, an egg rewrite does not contain the after-state. But, using the next rewrite's before (or the target state in case of the last rewrite) gives us the after we need. We match on the before and after to find and isolate the substructure that changes. To do so, we walk the before and after tree until we find the first difference. This allows us to only consider the sub-diagram the lemma matched on and to identify parameters. Then we greedily find the first match for the lemma's left side by traversing top down on the before. From there, we can fill in any context holes (i.e., variables) and store which parameter matches with which expression.

Suppose we have the change `(zx0 ↔ n_wire n) ↔ (zx0 ↔ n_wire n) ⇝ (zx0 ↔ n_wire n) ↔ (zx0)` caused by the lemma `n_wire_removal_r : zx ↔ n_wire (Dep2 zx) ∝ zx`. We then compare the before and after and find the smallest change is `(zx0 ↔ n_wire n) → zx0`. Next, we compare the smallest change that we just found to the before side and compare it to the lemma: `(zx0 ↔ n_wire n)` vs `zx ↔ n_wire (Dep2 zx)`. We walk the structure and see that the difference is equal to the lemma with the substitution `zx ↦ zx0`. Note that we only look for substitutions for parts of the lemmas AST that are parameters. Meaning, we now know that `rewrite (n_wire_removal_r zx)` will rewrite the change we want in Rocq. If we look at the original expression, however, we notice that there are two locations, where this rewrite would work in the before expression `(zx0 ↔ n_wire n) ↔ (zx0 ↔ n_wire n)`: on the left and right of the top-level compose. This means, Rocq would rewrite both of these instances unless we specify the match index in our rewrite. To specify the match index, we take the substituted lemma statement and match it left to right (depth first) in the before AST. We then count the numbers of matches before and including when we hit the actual match. This becomes our index.

Formally, we see a lemma (indexed by i) as

$$l_i(p_1, p_2, \ldots, p_n) := L_i(p_1, \ldots, p_n) \Rightarrow R_i(p_1, \ldots, p_n), \text{ if } T_i$$

where L_i, R_i are functions to give us left and right expressions from parameters and T_i are the dependent type conditions. We then formalize that the `egg` proof explanation as follows

$$S_0 \xrightarrow{l_0} S_1 \xrightarrow{l_1} S_2 \ldots \xrightarrow{l_{m-1}} S_m, \text{ where } S_i = \begin{cases} \text{before}(l_i), i < m \\ \text{final state}, i = m \end{cases}$$

. Then we find the differences $\Delta(S_i, S_{i+1})$ for $i \in [0, m)$. We then find the smallest different substructures s_i, s_{i+1}. We know that $s_i = C_i[L_i] \wedge s_{i+1} = C_i[R_i] \wedge T_i(s_i)$, for a context $C_i[\cdot]$. This context then induces a substitution σ_i, such that $\sigma_i(s_i) = s_{i+1}$. We then walk S_i with the substitution to find the index a of where the actual difference between S_i and S_{i+1} occurs. This substitution $\sigma = [p_1 \mapsto e_1, \ldots, p_n \mapsto e_n]$ is the exact mapping we need to for Rocq: Outputting `rewrite (`$l_1\ e_1\ \ldots\ e_n$`) at` a gives us the exact format we need for a correct Rocq proof.

Lemma directionality One last note that we have glossed over is that Rocq's lemmas are often bidirectional (assuming all variables are on both the left and right side). When this is the case, we generate two lemmas

$$l_i(p_1, p_2, \ldots, p_n) := L_i(p_1, \ldots, p_n) \Rightarrow R_i(p_1, \ldots, p_n), \text{ if } T_i \text{ and}$$
$$\hat{l}_i(p_1, p_2, \ldots, p_n) := R_i(p_1, \ldots, p_n) \Rightarrow L_i(p_1, \ldots, p_n), \text{ if } T_i$$

4.1.5 Interfacing with Rocq

Now we need to get our system to work within Rocq. There are a few parts we need to work in interactive proof mode:

1. Create and manipulate hint databases to convert lemmas
2. Automatically simplify dependent type arguments based on rules `simpl` uses

3. Attempt to solve an equality with a provided hint database

4. Reduce a term to the syntactically smallest form

To solve equalities, we need to convert the left and right term into `egg` and use the hint database to perform equality saturation.

We then check equality and export the proof via proof explanation (See Section 2.2 and Nieuwenhuis and Oliveras [2005]). Once the proof is produced, we adapt the format to fit Rocq proofs and have it checked by the engine.

4.1.6 Plugin vs. Language server

Since we want to use `egg`, which is a Rust library, our solver based on `egg` needs to be invoked from Rocq's interactive proof mode. Generally Rocq is not designed to work with other external tools, so built-in automation methods like `Ltac` are unable to call out to other tools.[2] Hence, there are two main alternative ways we could achieve this:

1. Via a Rocq plugin

2. Via a language server

In the following, we compare these solutions.

Rocq plugin Rocq has a "plugin API" that allows developers to augment the proof assistant with new tactics that can execute arbitrary OCaml code. In general, these new tactics should be stateless; this is a soft limitation, however, and we could run `egg` proof generation, which would be side-effect. The reason we put "plugin API" in quotes is that the Rocq plugin API differs from what one would usually understand from a plugin API: In Rocq, a plugin is really an augmentation to the kernel. There is no encapsulation and

2. A tactic `external` with which you could invoke other processes for proof generation, but was removed in 2016

the developer works directly with the Rocq internals. This route would enable us to build a system where a user simply calls a tactic and then the plugin generates a proof by invoking egg and has Rocq check it. In practice, we would build a tactic that a proof engineer calls and is invoked on every compile. The great benefit to this approach is that the developer experience is very idiomatic: Have a problem, call a tactic, (hopefully) QED. Unfortunately, this solution comes with some major drawbacks: Firstly, given the Rocq plugin API is really just an extension to the Rocq kernel, plugins break at almost every release of Rocq (so roughly every 6-12 months).[3] Further, despite being an extension to the kernel, there is no (documented) way to access the internal rewrite databases. Another issue is that there are many plugins out here where the authors have simply moved on, and they are now stuck on old versions of Rocq. An example would be PumpkinPatch by Ringer et al. [2018], which is a wonderful tool that we wanted to use during the development of *Vy*ZX, only to discover it does not work with any modern version of Rocq. Given the plugin API is very unstable and changes with (almost) every version of Rocq, this is also not a criticism of the authors but rather acknowledge the difficulty of plugin development. As we have to also acknowledge the nature of the academic process, we see issues like this with great worry, as we do not want the work we produce to fall to the wayside simply due to updates breaking compatibility. The other major disadvantage is the poor documentation of plugin development. While there are some domain experts, there are no (non-basic) tutorials, and writing a plugin is trial by fire. This greatly increases development time and likely means the plugin is outdated by the time it is completed.

Language Server Protocol (LSP) The language server protocol (Gunasinghe and Marcus [2022]) is a language and editor agnostic protocol to give context information of about code to the editor. It is currently in use for most well-known IDEs and text editors, such

3. The reader less familiar with Rocq will see that Rocq did not have its major version change in years; however, Rocq almost always breaks backwards compatibility in minor version bumps.

as Visual Studio Code, vim, and Emacs. Technically speaking, a language server is an Remote procedure call (RPC) server that uses a well-defined API that can be, among other things, called from code editors to provide context to the current position of the cursor. Context can mean anything from formatting, syntax highlighting, and code completion to live proof states in Rocq. Rocq has a language server (Rocq LSP) by Gallego Arias et al. [2023] that provides live proof states in the expected way (comparable to older Rocq editor plugins). There are editor plugins using Rocq LSP for Visual Studio Code, vim, and Emacs. The Rocq LSP itself provides a real API that developers can hook into Rocq to get proof states to custom extensions and use these to provide auxiliary information back to the proof engineer. For example, ZXViz, as discussed in Section 3.4.1, uses this API to render ZX-diagrams live for *Vy*ZX. This project would use the language server approach to provide code editor commands that can be invoked, use the proof state and hypotheses, to return a sequence of commands as a code completion. In tandem with Rocq LSP, Pétanque by Gallego Arias et al. [2023] is a JSON RPC server that allows interactive queries of proof state. This server allows arbitrary Rocq command execution in the current Rocq state, allowing for, for example, hint lemma extraction. Pétanque is part of the Rocq LSP project and shipped with Rocq LSP. It is specifically designed for use cases like ours. Meaning, the proof will be visible to the user as generated in the code editor, and the language server is completely invisible to the proof checker. While this appears to be more convoluted, it means the project code will not be broken if our tool becomes outdated or a new user does not have it installed. The great advantage of this approach is that the Rocq LSP (and by extension Pétanque) has shown to maintain the same API over various versions of Rocq. Given that it is actively being developed, we expected it to deal with the internal Rocq changes going forward, too. We opt for a solution based on the language server protocol, as the small cost in developer experience is outweighed by the much better documentation and longevity.

Extracting lemmas from Rocq To extract lemmas at a given point and send them to our Rust implementation, we call Pétanque and query the user's specified database by sending the commands `Set Printing All.` and `Print Rewrite HintDb [NAME].` The latter will print all lemmas in the rewrite database with the given name. This is the same system that Rocq's `autorewrite` uses, meaning no changes to the Rocq installation is necessary. With setting all printing ensures there are no difficult to parse notations that potentially obfuscate details in our way, and we can examine the raw lemma types. Pétanque allows us to perform these operations at the user's cursor position's state but completely transparently to the user. We take the lemmas, parse them into NDIR and send them to the external code that actually performs the computation using `egg`.

Running problems from Rocq Using Rocq lsp we query the current goal state and context, parse it and send it off to our tool via JSON RPC. Once the system finds a solution, it provides a serialized form, which will then be converted into proof code, plus comments that document the exact proof state changes. We wrap such a proof in `idtac` to make it obvious which parts of source code is generated to the user and the plugin.

Stale proofs This wrapping exists so the plugin checks for errors in the current file. Our system can find generated sections and automatically check them for incorrect proofs. Such errors are often a symptom of changing the proof before the generated code in the file, or from changing some dependency. The system then prompts the user to regenerate the solution. As the non-automatic proof change is the biggest pain point compared to a Rocq plugin, we believe this solution gives us the benefits of not having a plugin along with quicker builds whenever the proof isn't broken while taking the burden of finding broken proofs from the user.

4.1.7 Complexity and decidability limitations

Independent of our choice of Rocq interface, we will be using e-graph systems and will be constrained by their limitations. It is clear that our e-graph based approach cannot solve any ZX-diagram equivalence in non-exponential time, as this problem reduces to graph isomorphism. Solving the problem of rewriting between different, semantically equivalent, structural diagrams, however, is decidable.

Recent work by Zhang and Flatt [2023] tries to outline some decidability constraints. Skimming this paper, one would assume associativity and commutativity normalization is not solvable in e-graphs. However, their counterexamples rely on either commutativity axioms, which are not present in *VyZX* or the zero element for multiplication, which also does not have an analogy in *VyZX*. Hence, we find their claim that associative theories are not decidable in equality saturation is too strong of a statement. Rather, it should focus on commutativity with identity insertion, as this is the actual problem this paper outlines.

e-graph blow-up As alluded to before, e-graphs run into troubled when having rewrites that add identities, as this process can be repeated indefinitely. For example, consider that $\texttt{zx} \propto \texttt{zx} \leftrightarrow \texttt{n\_wire n}$. From this lemma it follows that we can add an arbitrary amount of identities to the right of a given diagram. In equality saturation, not only would this happen, but it would create an infinite amount of new terms that need checking for rewrites. Hence, avoiding such situations is paramount. In our default rule set, there is no such rule, except adding casts in the stack associativity rule. However, we have eager cast contraction (meaning that multiple casts can be folded into one) to avoid this issue of infinite blow-up.

One other source of e-graph blow up are long associativity chains. An associativity chain *ac* of length *n* is defined as

$$ac_{c\in\{\texttt{Stack}\mid\texttt{Compose}\}}(n,\texttt{zx}) = \begin{cases} \texttt{zx}, n = 0 \\ c\ \ \texttt{zx}_n\ ac_c(n-1), \text{ otherwise} \end{cases}$$

For $ac_c(n)$ there are $\mathcal{O}(2^n)$ possibilities of associating the n terms. We inherently can construct proof where we might need any one of these possibilities, so this problem will always exist, given that we also have distributivity that might occur in places. So the largest associativity chain $ac_c(n')$ determines the level of exponential growth. We study the practical implications in Section 5.3, but in short, problems up to $n' = 26$ are quickly tractable. Since we could not find an example of $n' > 8$ in the full *VyZX* development, these chains should not be a problem in practice.

Proof generation limitations Generating proof in e-graphs uses the Flatt et al. [2022] algorithm. In general, extracting an optimally sized (i.e., minimum number of instructions) proof is an NP-hard problem, but, the authors make the claim they have a good approximation. In the e-graph itself, proofs are stored as trees. These trees represent proofs in a way that reduces proof step duplication by representing dependencies of a proof (i.e., subproofs) as children. These tree proofs take a very localized view of the problem and reuse explanations. An example for `egg` by Willsey et al. [2021] is shown in Listing 4.1. Linearizing this proof would require us to replicate each let statements as many times as it was used. This is why this process is exponential in general. Given that linear proof extraction is an NP-hard problem, and the proof extraction algorithm by Flatt et al. [2022] has a polynomial complexity, it follows that linearizing is NP-hard. In our later evaluation (Chapter 5) we will analyze to what extent this exponential nature is a problem in practice. We also see that a proof in this format is not conducive to being represented in Rocq, where rewrites operate on one segment of a larger term.

```
(let
    (v_0 (Rewrite⇒ mul-one (* x 1)))
    (let
      (v_1 (+ (Explanation x v_0) (Explanation x v_0)))
      (let
        (v_2 (+ 1 1))
        (let
          (v_3 (Rewrite⇒ factor (* x v_2)))
          (Explanation
            (+ x (+ x (+ x x)))
            (Rewrite⇒ assoc-add (+ (+ x x) (+ x x)))
            (+ (Explanation (+ x x) v_1 v_3) (Explanation (+ x x) v_1 v_3))
            (Rewrite⇒ factor (* x (+ (+ 1 1) (+ 1 1))))
            (Rewrite⇒ comm-mul (* (+ (+ 1 1) (+ 1 1)) x))
            (*
              (Explanation
                (+ (+ 1 1) (+ 1 1))
                (+
                  (Explanation (+ 1 1) (Rewrite⇒ constant_fold 2))
                  (Explanation (+ 1 1) (Rewrite⇒ constant_fold 2)))
                (Rewrite⇒ constant_fold 4))
              x))))))
```

Listing 4.1: egg proof explanation of (+ x (+ x (+ x x))) = (* 4 x) in s-expression style

4.1.8 Complexity analysis

For this section, let us define a few variables / functions:

$\mid t \mid$	Size of a syntax tree t
c	Number of lemmas
p_l	Number of parameters in lemma l
n	Number of proof steps

Lemma conversion We consider a lemma l to have p_l parameters and a left-hand side term t_0 and an right-hand side term t_1. Due to one traversal per parameter of the syntax tree to extract conditions and find instances of dependent type arguments in the term, we have an overall complexity of $\mathcal{O}((|t_0| + |t_1|)(p_l \log p_l)$ per lemma.

Equality saturation, proof production, proof flattening Equality saturation in general is an NP-hard problem. As we will see in the results (Chapter 5), we manage to usually stay in sub-exponential cases and we avoid e-graph blow-up. Therefore, for the problems we care about, we can consider equality saturation sub-exponential.

Proof production is a different story. The algorithm by Flatt et al. [2022] produces tree proofs in $\mathcal{O}(n^5)$. Aas we discussed before, however, we must flatten proofs to make them work with Rocq. This process is again NP-hard [Nelson and Oppen, 1979].

Proof conversion When converting proofs, we go through each proof step and compute the direction, parameters, and at value. These processes all happen per proof in $\mathcal{O}(|t| \cdot c(p \log p))$, where $t = \max(|t_{\text{before}}|, |t_{\text{after}}|)$, and p is the largest number of parameters for any lemma used. Therefore, proof conversion is overall polynomial in t, c, p, and n.

Front-end/transport layer The parsing, encoding, decoding, and decoding is all polynomial in the input/output size: For our parsing we use parsec which is an LL(k) parser that runs polynomially in input size (see Aho and Ullman [1972]). We then convert the input into NDIR which happens in squared time relative to input size. When getting the proof back, we linearly iterate over the proof steps and syntax tree to convert them into valid Rocq proofs.

Overall While in general we are working with multiple NP-hard problems, we mostly only run into NP-hardness in the process of flattening proofs. This step will likely be the cause of any significant slowdown. In Chapter 5 we see that the step is not a problem in our benchmarks.

CHAPTER 5

EVALUATION

To evaluate the e-graph-based rewriting system described in this dissertation, let us review the goal we had for e-graph-based automation. We want to solve *practical* problems and investigate the limitations of our approach. Further, the idea for the e-graph-based rewriting grew out of the specific problem of associativity and distributivity. We want to make sure to evaluate associativity/distributivity (AD) specifically, but not limit ourselves to this domain. Hence, we will try replacing common instances AD but also common instances of Rocq's `autorewrite` tactic within *VyZX* with our solution.

While evaluating these practical problems shows us how useful the system in practice, we do want to explore limitations. For this, we devise synthetic benchmarks that show us how the system behaves under worst-case scenarios and larger than practical problems.

Note that for this section, all tests were performed on using the release build of our rewriting tool on regular consumer hardware. Specifically, all practical tests involving real Rocq problems are performed on an Intel 8890HK with 32 GB RAM macOS system and all synthetic tests are performed on an Intel 12400 with 32 GB RAM on WSL.

5.1 Custom databases

Next to our original problem, we want to see how well our system performs in other common `autorewrite` settings. The most common `autorewrite` we use in *VyZX* is `simpl_casts`, which uses a database in charge of simplifying casts (Appendix A.1.2 has a list of lemmas for the database). For our benchmark, we take all (non-trivial) cast simplifications in *VyZX*'s rule development and compare them against the `autorewrite` performance. Not only do we want to compare speed, but we also want to try tasks that create circular prob-

lems that make `autorewrite` run in infinite loops. The exact lemmas we use and problems we try can be found in Appendix A.1.

We find three examples of the longest proof terms generated by `simpl_casts`. Further, we find the seven longest cast proof sequences that may or may not include `simpl_casts`. For the cases that are not just an invocation of `simpl_casts`, we would need to add lemmas that cause `autorewrite` to go into an infinite loop. However, we show that e-graphs can handle these cases. Specifically, the reason that `autorewrite` goes in an infinite loop is that adding the lemmas we need for the more complex problems already have their inverse in the original database. We call a lemma l_2 an inverse to l_1, if $\forall\, t$, where l_1 rewrites to t', l_2 rewrites t' to t. The e-graph data structure is perfectly equipped to deal with such circular rewrites as during equality saturation upon the first rewrite t and t' are in the same e-class. `autorewrite` on the other hand, will infinitely apply l_1 and l_2 in succession, hence creating an infinite loop.

Those are only a few problems in the development to that `autorewrite` fails on, although, these are typically the complicated problems: Most invocations of `simpl_casts` can be replaced by a single proof line. The proofs where `autorewrite` fails show us that our e-graph based solution is more powerful than `autorewrite`.

To see if our e-graph tool is a replacement to `autorewrite` in *VyZX* is to compare performance. The goal is to be as fast or faster. Note that we chose the problems to be the problems with unique proofs, such that each problem needs a different set of lemmas.

Table 5.1 show the performance comparison. Notice that Problem 3 and Problem 9 cannot be solved by either `autorewrite` or our e-graph system. In the manual solution, we solve the problem with the trick of introducing new casts with user-specified target dimensions. e-graphs cannot solve problems like these, as it can only rewrite based on clear facts about the term, and we do not allow the e-graph to oracle all possible values,

ID	autorewrite	e-graph system	CoqHammer
0	23	202	DNF
1	152	127	DNF
2	148	163	DNF
3*	DNF	DNF	DNF
4	254	167	DNF
5*	DNF	1035	DNF
6	231	20	DNF
7	76	6	5651
8*	DNF	29	DNF
9*	DNF	DNF	DNF

Table 5.1: A comparison of solution time for autorewrite and our system in context of cast proofs. DNF means "did not finish," which we concluded once either a failure message or two minute timeout was reached. The e-graph system's solution time is the sum of the time to unify, explain, and convert proofs. Times are reported in milliseconds, rounded to nearest ms. * indicates that for any solution, additional lemmas must be added to the database that cause autorewrite to become stuck in an infinite loop.

as that would be an immediate cause of e-graph blow-up, since equality saturation will necessarily find all equal expressions.

We also compare our work to CoqHammer by letting it run in both suggested modes – sauto and best created by Czajka [2020] – with a 30-second timeout[1] and our rewrite database, picking the better result.[2] All problems, except Problem 5, failed to solve with the time constraint. We discuss the likely reasons in Section 5.2 after also evaluating on AD problems.

To give us a better understanding of the search space we are covering, Table 5.2 shows the number of e-nodes that the system has created by the end of its run. We see that search spaces vary quite a bit in size, but that even large search spaces can be traversed quickly to find a solution. It also shows us that cast problems are fundamentally hard. There are a lot of ways to rewrite any given problem.

1. In our testing we found a 30 second time set for CoqHammer leads to actual time to completion (or failure) of 200-300s. We have been unable to discover why and suspect a bug.

2. Results of best and sauto typically only differed minimally.

ID	Number of e-nodes
0	20
1	6768
2	32964
3	24349*
4	28806
5	448
6	32011
7	95
8	499
9	26568*

Table 5.2: The number of e-nodes generated for each cast proof. * means no correct solution was found.

5.2 Associativity/Distributivity

To evaluate our performance on associativity/distributivity (AD) problems, we use real problems from the *VyZX* development. We give the system access to associativity, distributivity, and very basic cast simplification lemmas to handle stack associativity. The full list of lemmas and problems is in Appendix A.2.

One may note that our custom database example dealt with casts that are introduced by stack associativity. For these benchmarks, we only provide a small subset of cast lemmas, but in general there is no technical reason our system couldn't handle more lemmas. We choose the longest associativity rewrite sequences we found, plus a basic example labeled case 0 (this is a base case, allowing people setting up the benchmarks to verify the system is working).

In Table 5.3, we show the times for `autorewrite`, CoqHammer and our system. Note that the original proof lines were not intended for shortest length but were written as the developers understood the problem. Producing the shortest proofs possible (aka. Code Golf) is not a core goal of ours, but it shows that we produce reasonably sized proofs and can even beat handwritten proofs. Further, we see that the total time spent by our system to generate proofs is under 250ms, which is about as fast as the `autorewrite` results

in Table 5.1; fast enough not to interrupt flow. Further future executions of the generate proof are a few simple rewrite commands that can be checked in <100ms, instead of having to recompute the expensive search operation, further amortizing the cost. Now `autorewrite` cannot solve any of the benchmarks we provided but if it could every time the file is run a slowdown is experienced, a problem that is becoming very noticeable in build times for *VyZX*. We also notice that our system is unable to find a rewrite for the largest problem (Problem 9) in our benchmark set. This is because here we have casts interact with associativity and distributivity.

We again compare our work to CoqHammer by letting it run in both `sauto` and `best` mode with a 30-second timeout and our rewrite database.[3] We see that CoqHammer fails on most instances and is very slow on the ones it succeeds at. We believe this is because when rewriting CoqHammer tries to induce a lexicographic order to find the smallest terms. Though, in our test cases, we have counteracting lemmas that cause the same problem as in `autorewrite` to show up in CoqHammer. Concretely, we have both directions of `stack_compose_distr` (see Appendix A.2).

Irrespective of the reason, we consider one second to be the maximum time automation can take to not interrupt flow and CoqHammer exceeds that time substantially. Further, unlike our solution, CoqHammer has to run on every compile, blowing up compile time. In our few examples, CoqHammer adds over 6.5 minute of compile time to every single compilation run. Further, this would be on every run, while our system caches proofs.

To give us a better understanding of the search space we are covering, Table 5.4 shows the number of e-nodes that the system has created by the end of its run. This shows us that the actual induced search space from practical problems is at a very reasonable scale. Further, we notice the AD problems carrying significantly fewer possible solutions

3. As in Section 5.1, the 30sec timeout allowed for run times of 200-300 seconds in practice

compared to the cast problems we saw in Table 5.2, showing just how well suited this specific domain is to machine search.

ID	`autorewrite`	e-graph system	CoqHammer
0	78	202	61445
1	DNF	7	DNF
2	DNF	8	DNF
3	DNF	5	123561
4	165	28	60958
5	DNF	24	DNF
6	DNF	26	DNF
7	DNF	22	122831
8	252	9	33378
9	DNF	DNF	DNF
10	DNF	DNF	DNF

Table 5.3: A comparison of solution times of `autorewrite` with our system and CoqHammer for associativity/distributivity proofs. DNF means "did not finish" which we concluded once either a failure message or two minute timeout was reached. The e-graph system's solution time is the sum of the time to unify, explain, and convert proofs. Times in milliseconds, rounded to nearest ms.

5.3 Synthetic Benchmarks

To test the limits of our e-graph rewriting system, we devise a worst-case scenario. We know that solving associativity problems does not necessarily scale well with e-graphs. So we devise a benchmark, where we build a *VyZX* term fully left associated, and we want to rewrite it to a fully right associated term. The reason this is a worst case scenario is that technically there are exponentially many ways to associate an expression. `egg` has the job to find all of them and then sift through to find the shortest proof. So given this exponential case, our experiment wants to determine the following

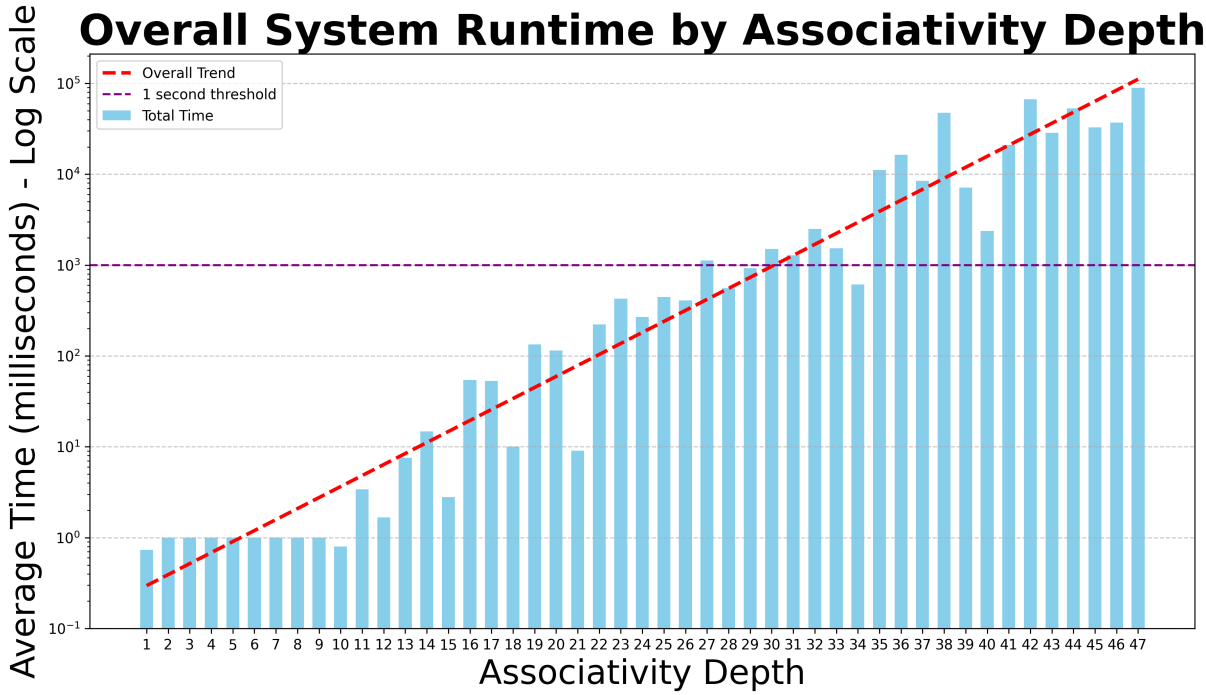
1. What is the maximum depth we can practically compute before `egg` fails?
2. What is the maximum depth we can compute in what a user would find acceptable, even if not instant (<1 min)?

ID	Number of e-nodes
0	41
1	97
2	367
3	1397
4	102
5	596
6	615
7	409
8	113
9	1300*
10	702*

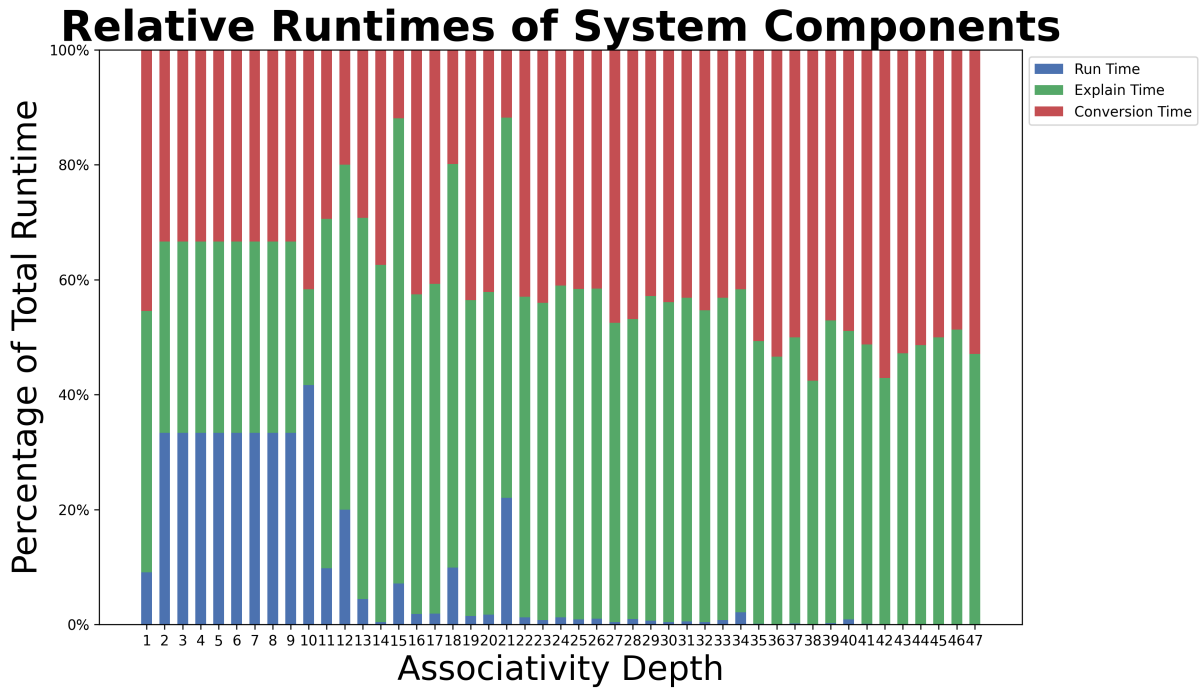
Table 5.4: The number of e-nodes generated for each associativity/distributivity proof. * means no solution was found.

3. In which parts of the pipeline (equality saturation, proof production, proof conversion) do we actually spend the majority of the compute time on?

We present the results in Figure 5.1. We can see the trend confirming our hypothesis of seeing exponential growth in time. Note that this still doesn't mean that we have e-graph blow-up (as the growth is not infinite); the e-graph doesn't grow infinitely in size, only exponentially. We also see that most of the time is spent on explaining and converting proofs. In fact, the actual equality saturation time is almost instant. This is expected, as flattening proofs in egg is an NP-hard problem with high constant factors. Unlike tree-proofs, for which egg optimizes, flat proofs are more akin to Rocq proofs, where you have one rewrite transforming the global expression to the next with a strict order. Therefore, there is no trivial way to optimize. Note, that the actual data-structure created here can and does become exponential in terms of base term size. After that, our polynomial time serialization works on the exponentially sized data-structure to transform it into a Rocq proof Beyond the results shown here, the system uses more than 16GB of memory, the limit assigned for maintaining repeatable results, hence not finishing.



(a) Synthetic benchmark run time by depth. Note that the performance of the smallest benchmark results was clamped at 10^{-3} ms.



(b) The relative runtime of each sub-component for a given benchmark.

Figure 5.1: The synthetic benchmark performance. All benchmarks were run 25 times and we are showing the averages.

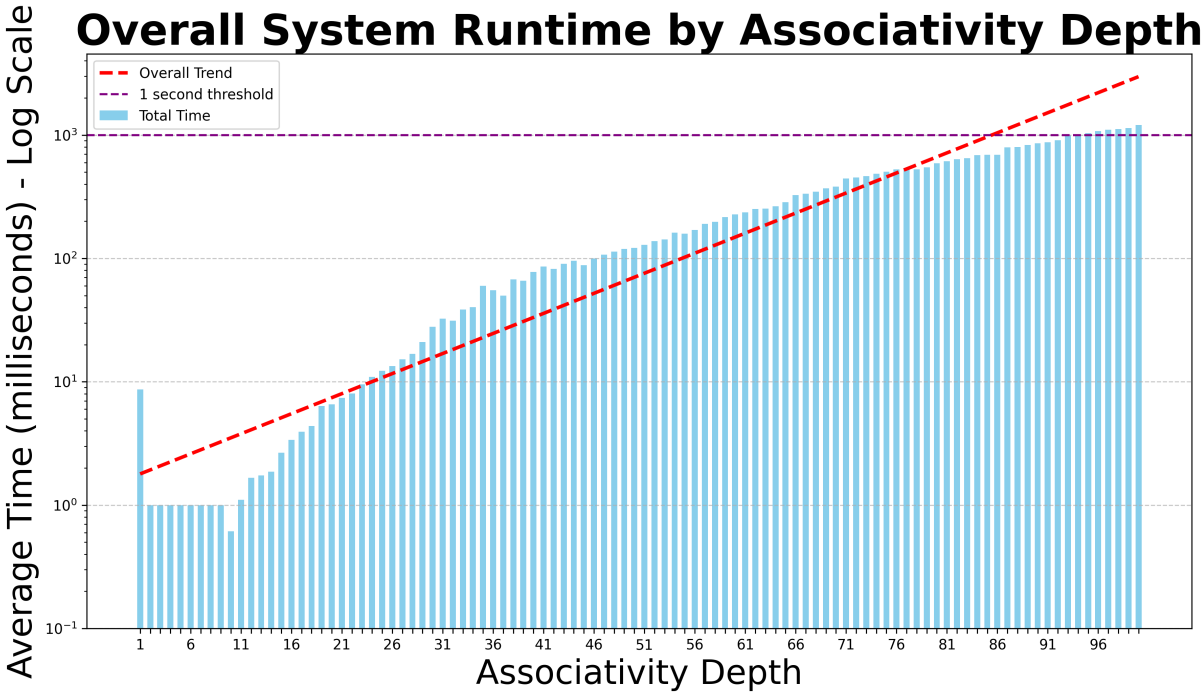
5.4 Synthetic Benchmarks II: Natural Number associativity

Similar to the previous synthetic benchmarks we test solving associativity chains, this time in the realm of a natural numbers instead of *VyZX* diagrams. We define our binary operation as add and our term as variables in these chains: the left associated term as $l(1) = v_1, l(n) = l(n-1) + v_n$ and as $r(1) = v_1, r(n) = v_n + r(n-1)$, where v_i are variables. We the run them like in the set up from Section 5.3 and present the results in Figure 5.2.

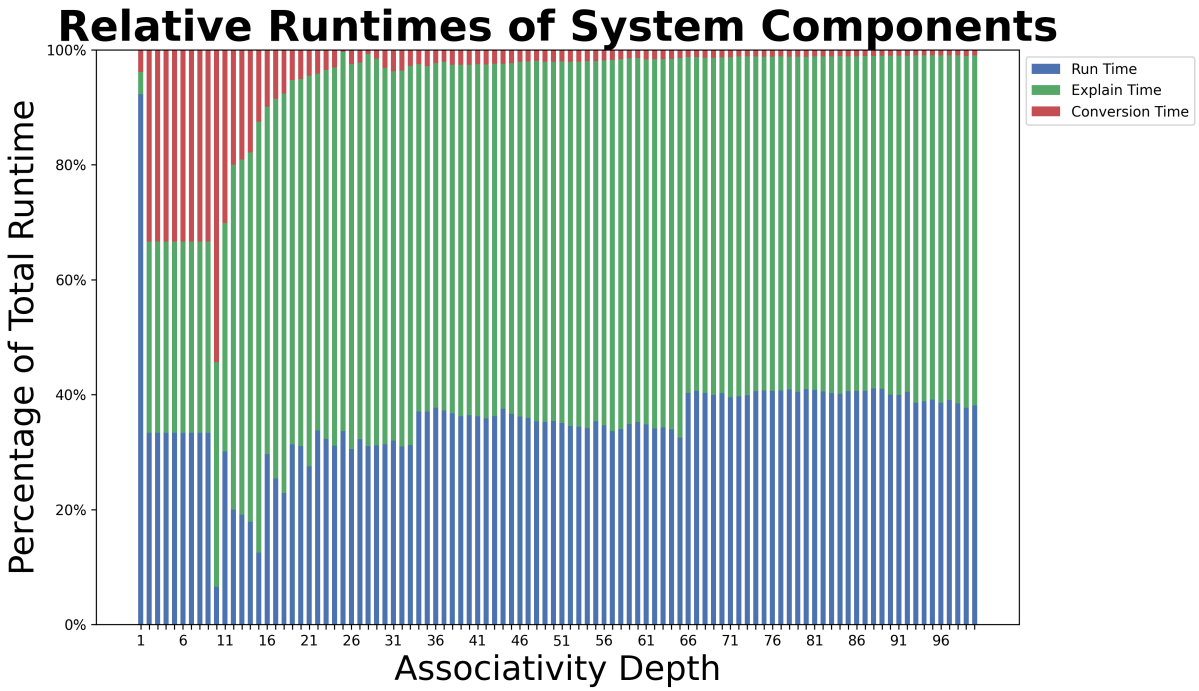
Here we see some clear differences to *VyZX* problems:

1. The time increase is sub-exponential
2. We can run much larger problems
3. The same problem size runs faster
4. The proof conversion is significantly less of the share of the run time

All of these differences come down to the lack of dependent types. It seems the additional e-nodes and additions to the proof that are `Dep_1` and `Dep_2` rewrites cause difficulty for the proof extraction algorithm which leads to larger flat proof which leads to longer proof conversion. In general our dependent type calculation method causes the size of the e-graph to increase by a constant factor. Each of these e-nodes is a candidate for rewrite checking. So even without the worse proof impact, run time for dependently typed systems scales with the number of dependent type arguments. This suggests that `egg` reasons very well about non-dependent problems, making it a promising tool for domain-specific proofs. Furthermore, this capability can also benefit dependent type systems by simplifying the reasoning required for their type arguments.



(a) Synthetic benchmark run time by depth. Note that the performance of the smallest benchmark results was clamped at 10^{-3} ms.



(b) The relative runtime of each sub-component for a given benchmark.

Figure 5.2: The synthetic benchmark performance for natural numbers. All benchmarks were run 25 times and we are showing the averages.

CHAPTER 6

RELATED WORK

In this chapter, we explore work similar to the proposed work and how our work contributes to the overall research landscape.

6.1 Quantum verification

ZXlive, Quantomatic, and Chyp The most notable ones include ZXLIVE by Shaikh et al. [2023], Quantomatic by Kissinger and Zamdzhiev [2015], and Chyp by Kissinger [2023]. All of these tools aim to provide some axioms and allow rewrites with them; a major difference to *Vy*ZX that proves ZX-calculus rules via their underlying semantics. ZXLIVE is, as the name suggests, a live and graphical proof assistant for the ZX-calculus. While the user experience is certainly great, it unfortunately does not have a verified backing, resulting in bugs. In addition, it cannot be used to prove facts about tools, rather it is more of a tool to validate new ZX-calculus rules. Quantomatic is the spiritual predecessor to ZXLIVE that allows custom user defined rules for graphical languages and to rewrite with these rules. Lastly, Chyp is similar to *Vy*ZX in that it is designed to give rules and use them to prove facts about symmetric monoidal categories. Chyp, however, like ZXLIVE, is a visual, axiomatic proof assistant for generic symmetric monoidal categories.

CoqQ, QuantumLib CoqQ by Zhou et al. [2023] and `QuantumLib` by the INQWIRE Developers [2022] are mathematical formalizations of linear algebra for quantum reasoning in Rocq. At their core they solve very similar goals, however, unlike `QuantumLib`, CoqQ relies on Rocq's Mathematical Components library [Roux et al., 2008]. This gives CoqQ a lot of reasoning capability, but also drastically increases complexity for users who are not already experienced with the Mathematical Components library. Furthermore, CoqQ defines simple program semantics, using a quantum-while language.

sqir & voqc As we just mentioned, `QuantumLib` does not formalize quantum circuits or any quantum programs. This is where SQIR by Hietala et al. [2021b] comes in: SQIR uses `QuantumLib` to define quantum circuits within Rocq. This definition is fundamentally quantum gate and qubit-based (as usual in the circuit model). SQIR's goal is to give proof engineers sufficient reasoning capabilities to prove facts about quantum circuits and measurement.

VOQC by Hietala et al. [2021d] is built on SQIR as a verified circuit optimizer. VOQC has the full feature set of a quantum optimizer: optimization passes, mapping, and routing for heterogeneous architectures Hietala [2022]. Using SQIR and VOQC Shor's algorithm [Shor, 1999] was verified 2022.

While it is not in scope to build a verified circuit optimizer for the dissertation, it is still an important work in the realm of quantum verification and *VyZX* uses the translation to the RzQ-gate-set provided in the VOQC package as part of circuit ingestion.

6.2 Proof automation approaches

Large language model (LLM) based automation Recent work has focused on the use of large language models to generate proofs. Perhaps most widely circulated, AlphaProof by AlphaProof-Team and AlphaGeometry-Team [2024] automatically proved Math Olympiad problems in Lean to "win" a silver medal. Note that propositions still had to be manually translated by humans. While this is not a classic view of proof automation, we argue it is still a proof automation tool at its core driven by an LLM. Unfortunately, not many technical details are known, except that it relies mainly on a combination of Gemini (Team et al. [2024]) and AlphaZero (Silver et al. [2017]) machine learning models.

Some other work focused on providing a GitHub copilot-like experience for proof assistants, most notably the work by Song et al. [2024] for lean and coqpilot for Rocq by Solovev et al. [2023]. Instead of generating proofs under the hood and only exposing a

tactic invocation to their users, these tools generate explicit proof steps for the user to see. This is a very sensible design choice given the nondeterminism introduced by LLM generation. The lean tool found great success over existing search tools; however, it does not replace the need for larger automation. At the time of writing, there is no empirical evaluation of coqpilot or Lean copilot.

Congruence in Rocq Rocq has a tactic called `congruence`. This tactic, by Corbineau [2007], uses a congruence closure algorithm by Nelson and Oppen [1979] to identify equality between ground truth and terms equal up to constructor theory ((in-)equalities that the tactics `injectivity` and `discriminate` can handle: where constructors are either not equal, or have arguments of non-equal constructors). In general, Corbineau defines congruence problems as follows over an equivalence relation $\propto$ and n-ary function f:

$$\frac{s_i = t_i, \forall i \in \{1, \ldots, n\}}{f(s_1, \ldots, s_n) \propto f(t_1, \ldots, t_n)} \quad (\text{CONGR}_f)$$

They add the following constraints for constructor theory, where the i^{th} constructor of sort α denoted C_i^α:

$$\frac{C_i^\alpha\, s_1 \ldots s_n \propto C_i^\alpha\, t_1 \ldots t_n}{s_i \propto t_i} \text{ if } C_i^\alpha\, s_1 \ldots s_n : \alpha \wedge C_i^\alpha\, t_1 \ldots t_n : \alpha \quad (\text{INJ}_i)$$

$$\frac{i \neq j}{C_i^\alpha\, s_1 \ldots s_n \not\propto C_j^\alpha\, t_1 \ldots t_n} \quad (\text{DISJ}_{i,j})$$

The congruence closure algorithm used can be seen as a very similar process to what e-graphs are using: A union-find data-structure is used to reason about equivalent terms

as one using the rules shown above and using $\text{DISJ}_{i,j}$ for fast failure. Given the publication is from [2007], this long predates modern equality-saturation based approaches, such as the ones found in `egg` and `egglog`. With the union-find data-structure, it solves the key problems of ordering and exhaustion, too. Unlike modern approaches, however, the algorithm is much slower and hence practically limited to a smaller set of cases.

CoqHammer CoqHammer by Czajka and Kaliszyk [2018], Czajka [2020] is an automation library for Rocq based the previous project Sledgehammer [Böhme and Nipkow, 2010b] for Isabelle/HOL [Nipkow et al., 2002]. Unlike other automation techniques, it uses machine learning (ML) to predict which theorems or parameters to choose. Fundamentally, CoqHammer works in four stages, as described by Ringer et al. [2019]:

- Select premises to use. In our case of the results (see Chapter 5), we specifically told CoqHammer to use our databases for this. This section is aided by ML in CoqHammer.
- The translator then translates the Calculus of Inductive Constructions with the extensions Rocq has to first order logic problems.
- It then uses a first-order logic solver to produce a proof.
- A proof reconstructor that translates the first-order solution to a Rocq proof. This happens through an algorithm similar to Rocq's `eauto` but calls out to custom tactics to address some implicit Rocq behavior such as simplification.

In effect, these stages seem quite universal for proof automation, as our work could also be categorized into the same format. The appeal of hammers is that they can be used to solve almost any problem at the expense of time. In that vein, CoqHammer aims to supersede Rocq's built-in automation in generality.

6.2.1 e-graph proof assistant automation

Lean-egg lean-egg by Rossel [2024] is a tactic that aims to generally encode Lean's lambda calculus terms within egg. It is likely the most comprehensive approach to a rewriting engine that we have seen so far. Rossel's strategy has shown great efficacy in proofs of rather simple types, such as group theory proofs. In his paper, they explore how they mechanized type theoretic rules and lambda reduction rules with egg to create a matching system that under user guidance can solve problems in Lean. Lean-egg's approach also showed various limitations of the egg rewrite system that stem from a lack of typing, an issue we worked hard on circumventing. This work does come with limitations, most prominently the incomplete handling of dependently typed rewrites. His work boils down to a congruence tactic accelerated by e-graphs. There is no congruence theorem for dependently typed settings, causing a fundamental theoretical problem. We believe that exploring domain-specific rewrite strategies can yield faster and better results because of the significant reduction in term size and rewrite rule count and complexity.

Coquetier Coquetier is a mechanization of Rocq using e-graphs that differs from lean-egg in that it does not use the underlying lambda terms of problems, but rather focuses on high-level Rocq terms [Bourgeat, 2023]. The tool aims to use hypotheses in context to solve a goal (in practice this means introducing many lemmas as hints into context). Like the work, Bourgeat believes that by using higher level terms, they can save themselves from a lot of complexity of handling the most general cases. Again, Coquetier is limited by excluding dependently typed problems. Like our approach, it strips type information but due to the increased generality it carries type information slowing down the reasoning process. Further, Coquetier is not an equivalence prover but rather a simplifier, meaning it uses a cost function to determine a simplest state. While undoubtedly useful this can

also prove restrictive as the simplifier may not find the exact change a proof engineer is looking for.

Relation via egg Relations via egg is a domain-specific rewrite engine for memory model proofs [Kozyrev, 2023]. It is the first proof of concept for integrating modern equality saturation with a CIC proof assistant (Rocq). Although limited in the complexity of terms it can handle, the domain-specific approach is similar to the dissertation work and served as inspiration for our e-graph system.

CHAPTER 7

FUTURE WORK

7.1 ZX-calculus verification

Given the wide range of applications of the ZX-calculus, *Vy*ZX is intended to become a basis to verify a broad spectrum of quantum tooling. Most interestingly, we think, *Vy*ZX lends itself very well for a quantum optimizer/simulator and for quantum error correction.

Quantum Optimizing Compiler/Simulator PyZX by Kissinger and van de Wetering [2019] has shown that the ZX-calculus is an effective tool to build a state-of-the-art quantum optimizing compiler and a basis for simulation, as shown by Kissinger and van de Wetering [2022]. As quantum computing is a domain where debugging is very difficult – as evidenced in the work by Li et al. [2022] – compiler bugs can become not only difficult, to find, but very expensive if a faulty program has to be repeatedly run or simulated. Further, most quantum optimizers use a small set of optimizations, unlike classical compilers that have many optimization passes. This leads us to concluding that quantum optimizing compilers are in a unique spot where the cost of mistakes is very high and the verification scope is small. Hence, we believe it is a natural application of *Vy*ZX that ought to be pursued.

There are some challenges, however. The block-like categorically inspired model that *Vy*ZX uses is orthogonal to the more traditional interpretation of graphs with spiders and edges that PyZX uses. This means there is a semantic gap in need of bridging. Even with this gap, however, we are convinced that *Vy*ZX's model is the better model for fundamental verification without axiomatization (outside the Rocq real numbers). This leaves two possible pathways forward:

1. Build a layer on top of *Vy*ZX that can reason over adjacency, instead of structure.

2. Come up with equivalently effective optimizations that behave better in block form.

The latter is a more theoretical research direction, that I do not have many pointers on. If one chooses to go the former road, however, there is a lot of interesting research of unifying graph theory verification with *Vy*ZX's more categorical reasoning. One has to consider how to represent finite open graphs in such a way that they can be destructured into block form. An approach proposed to us by Peter Selinger, is to put all spiders, i.e., spiders, into a vertical stack and connect them through various crossing caps and cups. This means that to break these problems down into *Vy*ZX one only has to build this stack of spiders, permutations, and multi-cap/cups. The real challenge lies in verifying the translation to the stack representation, a problem that is as-of-yet open. While building this system, one will likely encounter many associativity problems, something we hope to have eased significantly with our e-graph based proof automation.

Converting a *Vy*ZX block diagram into a graph is also possible. We outlined an algorithm in a 2022 preprint (Lehmann et al. [2022]). Before converting into graph representation, we convert ZX diagrams into a restricted yet just as universal form; this form is similar to Duncan et al. [2020] graph-like form, differing only in the existence of self-loops and two or more possible links between spiders. The restrictions are as follows:

1. Only Hadamard edges (this can be accomplished by adding multiple hadamards and identity spiders to edges that do not have them)

2. Only one type of spider (Z-spiders): Each X-spider can be converted by chaning its color and surrounding it by hadamard gates.

3. Every spider has one input and zero to two outputs, or vice versa

Restrictions 1 & 2 are in place to make our graph more conventional (and akin to graph-like diagrams) by having spiders and edges. Restriction 3, however, is in place for

ease of proving. To divide up the proofs into logical steps, there are separate intermediate representations (IR) in old builds of *Vy*ZX that build up all three restrictions. These IRs exist to divide proofs into logical components and are mostly transparent. Developers, however, can choose to use less restricted forms. These translations were non-trivial, but rather quick to verify.

Given our restricted form, our algorithm proceeds as follows: A procedure numbers all ZX diagram components (i.e., the constructors) with a unique integer. Then it proceeds to number edges of components as follows: Each component with n inputs and m outputs produce a pair of lists sized n and m, where every position in the list indicates the closest fundamental component (spider or cap/cup). A non stacking/composing component with n inputs, m outputs, and spider number x return lists of size n and m, each with all entries being x.

When stacking two diagrams, the procedure concatenates respective lists, and when sequencing diagrams, it carries forward the outer lists. Figure 7.1 shows an example of this process. It is important to note that we treat caps and cups like spiders at this stage. Once the edge numbering is complete, we traverse the diagram once more, and at every Compose, we match the output edge numbers of the left diagram with the input edge numbers of the right diagram and mark each of those as an edge. Looking at our example in Figure 7.1, we see wires labeled $(1,2)$, $(3,2)$, and $(3,4)$ bridging the main composition. We use the information from the algorithm to infer which inputs/outputs of the diagram are connected to which spider by looking at the outermost annotation. So in our example, we see by looking at the outermost labels that input 1 is connected to spider 1, input 2 to spider 3, output 1 to spider 2, and output 2 to spider 4. We can then annotate the entire diagram's inputs and outputs by looking at the outermost labels. Algorithm 1 shows a pseudocode description of these processes.

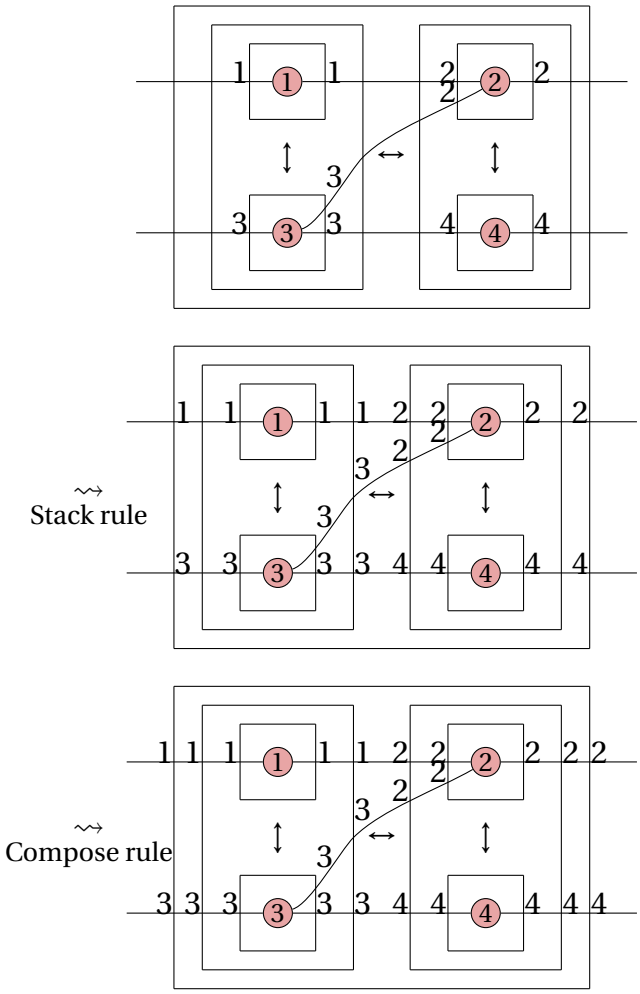


Figure 7.1: An example of the edge annotation algorithm. Each spider is shown with a unique spider number inside of it. In the first step, all wires out of the spiders are annotated with the spider numbers of the spiders. Then when stacking spiders, those numbers are carried forward and applied to the stack constructions wires. Note that the order is preserved. We see that the outer set of numbers is carried forward upon a composition in the last step. As before, $\updownarrow$ denotes stacking and $\leftrightarrow$ composition. Note that we consider vertically stacked numbers within a singular box to form a list. So for example, in the final diagram, the diagram is read to have the tuple $([1,3],[2,4])$ as inputs and outputs.

Algorithm 1 block representation to graph representation conversion

```
function NumberNodes(zx)
    assignFreshNumber(zx)
    if zx = Stack zx1 zx2 OR zx = Compose zx1 zx2 then
        NumberInnerNodes(zx1)
        NumberInnerNodes(zx2)
function NumberEdges(zx)
    if zx = Stack zx1 zx2 then
        (in1, out1) = NumberEdges(zx1)
        (in2, out2) = NumberEdges(zx2)
        return (in1 ++ in2), (out1 ++ out2)
    else if zx = Compose zx1 zx2 then
        (in1, _) = NumberEdges(zx1)
        (_, out2) = NumberEdges(zx2)
        return in1, out2
    else
        return NodeNumber(zx), NodeNumber(zx)
function CreateEdges(zx)
    if zx = Compose zx1 zx2 then
        (_, out1) = GetEdgeNumbers(zx1)
        (in2, _) = GetEdgeNumbers(zx2)
        for ((in, out) ∈ (out1, in2)) do          ▷ Note that out1 and in2 have the same length by construction
            AddEdge(in, out)
    if zx = Compose zx1 zx2 OR Stack zx1 zx2 then
        CreateEdges(zx1)
        CreateEdges(zx2)
```

Quantum Error Correction The noise of quantum computers means errors are inevitable. Quantum error correction (QEC) [Nielsen and Chuang, 2010a] is the idea of achieving fault-tolerance, while addressing the fundamental challenge that quantum states cannot be cloned. The core principle of QEC involves encoding the information of a single "logical qubit" into a highly entangled, redundant state distributed across numerous "physical qubits." This distributed encoding allows for the detection of errors by measuring multiple physical qubits, or syndromes, without destroying the logical state. This idea is similar to classical error correction, where additional bits are used for parity checking. Also like in classical computing there is a wide-range of quantum error correcting codes. In contrast, however, there is less clarity on which error correction code is "best" for which application. There has been a lot of previous work representing error correcting codes in the ZX-calculus. Notable example include the representation of the surface code, a common QEC model, by Kissinger [2022] and initial work by Cao and

Lackey [2025] on representing quantum low-density parity check (QLDPC or also LDPC) codes.

One can use *VyZX* to create a verified implementation of such a code. As a fundamental tool for scaling quantum computing, error correction has to be correct in both theory and, perhaps more importantly, implementation. As the ZX-calculus is an intuitive way to represent quantum error correcting codes, *VyZX* is a natural stepping stone in this regard. Verifying the surface code should also lend itself well to block representation, making verification straightforward in the sense that no new ZX-calculus graph layer is needed.

7.2 e-graph proof automation

With the exciting results we discussed in Chapter 5, we are very optimistic about using e-graphs for proof automation. The described automation proves that problems, including ones the tool was not specifically built for, can be solved efficiently. With these results in mind, in the following, we discuss extensions to the project that we believe would be fruitful.

Premise selector Like in CoqHammer– described in Section 6.2 – premise selection could be an exciting direction. If one would not have to supply lemmas themselves but have a premise selector select them automatically, this could make the e-graph based approach more universal and more ergonomic. Further, it could integrate with the scheduling of rules within `egg`, to choose which rules to run first. As in CoqHammer, this approach could also lead to increased performance, whether a database is explicitly provided or not.

autorewrite replacement In Chapter 5, we compared our e-graph system to Rocq's built-in `autorewrite`. While we managed to solve more problems directly, a major difference is that our system either solves or does not progress, whereas `autorewrite` will rewrite as far as it can. The heuristic `autorewrite` uses is to perform rewrites until dead-ends are found. It then picks the dead-end with the longest chain of rewrites. Note that in the case of infinite rewriting, `autorewrite` does not terminate, so upon termination such a chain exists. Practically, this behavior is often desirable, as it leads to making `autorewrite` a simplification technique. In fact, this is how *VyZX* uses `autorewrite` for cast simplification.

Emulating this behavior with e-graphs is possible. e-graphs allow for searching using a so-called "cost function." Cost-functions allow e-graphs to select a "best" member of an e-class. So after equality saturation runs with the initial term, ideally there should be multiple terms in the same e-class as before, meaning we found equal terms to our original term. Now with the cost function, we select from these terms to find our rewrite target. The question then becomes, which cost function is the best. There likely isn't one answer, but there are a few good starting points:

1. Smallest resulting AST size: This is perhaps the best initial cost function. Finding terms that are as small as possible is a close emulation of `autorewrite`. Though, we know from experience that in the case of associativity and distributivity, this is a bad heuristic.

2. Weighted AST: We assign weights to different AST terms we prefer. For example, in the case of *VyZX*'s associativity/distributivity, if we cannot solve, we typically prefer terms with as few casts as possible, as they make rewriting more difficult. For similar reasons, we prefer composes over stacks. Using these preferences together with AST sizes would likely give us a good heuristic.

3. Custom weighted AST: A problem that remains with the weighted approach is that different problems require different weights. One option is of course to have users specify the weights. Though, this process is manual and a burden on the engineer, though more flexible.

4. Machine Learning for weights: Using machine learning, we could dynamically figure out weights by looking at existing proofs using the same lemmas as in the database that we provide for our e-graph system to see what the proof engineer wants. Though, one needs big projects to have sufficient data for training.

Generalization Our approach being domain specific has shown great efficacy over existing general tools like `autorewrite` and CoqHammer. That leaves the big problem of having to build automation for every domain, which is clearly not a scalable solution. The work on the system suggest an approach, however, to keeping most benefits of the domain specific approach while allowing user developments to benefit: Building a tool that can reason over inductive data types with an equivalence relation.

Specifically, we need to restrict assumptions about the domains we work on:

- The system is designed to work on a single (dependent) inductive data type.
- The inductive data type must be non-empty; i.e., a lemma showing there exists at least one valid element must exist.
- There needs to be a single equivalence relation defined on the data type specified. These uniqueness requirements allow us to state our problem in `egg`.
- All dependent arguments are of inductive nature themselves. Further, the proof engineer can provide a set of simplification lemmas that emulate Rocq's `simpl` tactic (as previously discussed).

- The proof engineer must provide a set of rewrites over the data type and equivalence relation specified.
- It is the proof engineer's responsibility to avoid exponentially complex cases, as this cannot be done automatically.

Once these requirements are met, our procedures can be generalized to deal with inductive constructors and pseudo-inductive constructs. We translate a system that has two additional constructors into `egg` instead of domain specific constructs in WTIR:

$$\frac{\texttt{type\_name} : \texttt{String} \quad \texttt{ind\_constr} : \texttt{String} \quad \texttt{args} : \texttt{WTIR[]}}{\texttt{Inductive(type\_name, ind\_constr, args)} : \texttt{WTIR}}$$

$$\frac{\texttt{type\_name} : \texttt{String} \quad \texttt{ind\_constr} : \texttt{String}}{\texttt{Const(type\_name, ind\_constr)} : \texttt{WTIR}}$$

These rules would allow us to translate all types into this new form and run equality saturation. For example, `Z` spiders would look as follows: `Inductive(''ZX'', ''Z'', [n, m, a])` and we would represent natural numbers using `Inductive(''nat'', ''S'', n)` and `Const(''nat'', ''O'')`. Further, we would need to follow the same model we built for functions to create dependent type rewrites. From there, the existing infrastructure should work on the e-graph side. One complication is that we need a more complete parser of Rocq expressions. Right now, we can parse most expressions created by Rocq, however, not modules or universes. These features would need to be added to fully support all custom developments.

egglog `egglog` by Zhang et al. [2023] is a datalog style DSL that unifies equality saturation and e-graphs with database technology. It has a few distinct advantages over `egg`:

1. The DSL makes it significantly more concise to express terms and rules than using the Rust macros that `egg` provides.

2. Readability is drastically increased.

3. `egglog` supports multiple types at once, making a lot of the fragile untyped system from WTIR obsolete, replacing it with NDIR.

The reason we chose not to use `egglog`, however, that at the time of writing it does not support proof production. There are numerous technical challenges for producing proofs from the underlying database-like representation that are out of scope for this section, but it should be said this is not a quick copy-and-paste conversion from `egg`. Once `egglog` has proof production, translating the core of our system into `egglog` is very straightforward. In Listing 7.1 we show the early prototype implementation to see if associativity/distributivity problems can be solved with e-graphs– the original proof of concept leading us to this work. Similar to our approach, we then build lemmas with dependent type constraints as conditions. Our existing infrastructure could be reused to easily generate these statements. In fact, there is a python API that would make it easy to implement these rewrites programmatically. We show in Listing 7.2 how an example lemma would be implemented in `egglog`. Describing problems would be analogous to the lemma description. Overall, believe that the switch to `egglog` would significantly reduce technical debt and enable quicker development. Further, for generalizing our work, if possible, `egglog` would be a great place to start due to the option of representing multiple types easily.

```
; egglog proof of concept implementation of our e-graph system
; NDIR
(sort Dim)
(function Dim_lit (i64) Dim)
(function Dim_var (String) Dim)
```

```
(function Dim___add__ (Dim Dim) Dim)
(function Dim___mul__ (Dim Dim) Dim)
(sort ZX)
(function ZX_symbol (String Dim Dim) ZX)
(function ZX_cast (ZX Dim Dim) ZX)
(function ZX_compose (ZX ZX) ZX)
(function ZX_stack (ZX ZX) ZX)
(function ZX_Z (Dim Dim Dim) ZX)
(function ZX_X (Dim Dim Dim) ZX)
(function ZX_nStack1 (ZX Dim) ZX)
(function ZX_nWire (Dim) ZX)
; Example symbol definitions
(let ___________int_const__0 (Dim_lit 0))
(let ___________int_const__1 (Dim_lit 1))
(let ___________int_const__2 (Dim_lit 2))
(let Wire (ZX_symbol "-" ___________int_const__1 ___________int_const__1))
(let Empty (ZX_symbol "Empty" ___________int_const__0 ___________int_const__1))
(let Cap (ZX_symbol "Cap" ___________int_const__2 ___________int_const__0))
(let Cup (ZX_symbol "Cup" ___________int_const__0 ___________int_const__2))
; Dependent type functions and rules
(function ZX_n (ZX) Dim)
(function ZX_m (ZX) Dim)
(rule ((= x (ZX_cast a n m)))
      ((ZX_n x) (ZX_m x))) ; Force creation of dependent type node
(rewrite (ZX_n (ZX_cast a n m)) n) ; Dep type rewrite
(rewrite (ZX_m (ZX_cast a n m)) m)
(rule ((= x (ZX_compose a b)))
      ((ZX_n x) (ZX_m x)))
```

```
(rewrite (ZX_n (ZX_compose a b)) (ZX_n a) :when ((= (ZX_m a) (ZX_n b))))
(rewrite (ZX_m (ZX_compose a b)) (ZX_m b) :when ((= (ZX_m a) (ZX_n b))))
(rule ((= x (ZX_stack a b)))
      ((ZX_n x) (ZX_m x)))
(rewrite (ZX_n (ZX_stack a b)) (Dim___add__ (ZX_n a) (ZX_n b)))
(rewrite (ZX_m (ZX_stack a b)) (Dim___add__ (ZX_m a) (ZX_m b)))
(rule ((= x (ZX_Z n m alpha)))
      ((ZX_n x) (ZX_m x)))
(rewrite (ZX_n (ZX_Z n m alpha)) n)
(rewrite (ZX_m (ZX_Z n m alpha)) m)
(rule ((= x (ZX_X n m alpha)))
      ((ZX_n x) (ZX_m x)))
(rewrite (ZX_n (ZX_X n m alpha)) n)
(rewrite (ZX_m (ZX_X n m alpha)) m)
(rule ((= x (ZX_symbol s n m)))
      ((ZX_n x) (ZX_m x)))
(rewrite (ZX_n (ZX_symbol s n m)) n)
(rewrite (ZX_m (ZX_symbol s n m)) m)
(rule ((= x (ZX_nStack1 a n)))
      ((ZX_n x) (ZX_m x)))
```

Listing 7.1: NDIR in egglog with dependent types

```
(rewrite
; Step 1:  Constraint free rewrite
(ZX_compose (ZX_stack zx0 zx1) (ZX_stack zx2 zx3))
(ZX_stack (ZX_compose zx0 zx2) (ZX_compose zx1 zx3))
; Step 2:  Condition on output of functions representing dependent types
```

```
:when ((= (ZX_m zx0) (ZX_n zx2)) (= (ZX_m zx1) (ZX_n zx3))))
```

Listing 7.2: Stack Compose Distributivity

CHAPTER 8

CONCLUSION

In this dissertation, we discussed three problems:

1. Build a formally verified implementation of the ZX-calculus.
2. Automate the common problem of associativity and distributivity for our categorical structure.
3. Build a more general rewriting tool for ZX-calculus problems that can use custom lemmas.

For problem one, we introduced *VyZX*, a Rocq verification that takes a block view to the ZX-calculus. With this block view, we show that we can prove a complete rule set of the ZX-calculus. The structure we introduce creates some new proof challenges. For this, we present ZXViz, a visualizer for the block structure ZX-diagrams. With this, proof engineers can effectively use the inherent visual properties of the ZX-calculus while working in Rocq.

The other large gap was that our structure caused about a quarter of proof lines to deal with reassociation. For this, we built an e-graph system that can automatically rewrite while avoiding recomputation and efficiently finding rewrites. In our testing, we showed that it outperforms not only `autorewrite` that functionally stalls solving these problems but also CoqHammer. Our approach solves more problems than CoqHammer and works on sub-second timescales, where CoqHammer works on the order of minutes. We further discover through synthetic testing that associativity problems much larger than practically occurring still complete in under a second, even if the systems has exponential time growth with the increase in problem size.

From there, we generalized the e-graph rewriting engine to work with any rewrite database as found Rocq. This allows us to ingest custom lemmas to convert them to e-

graph rewrites and use them to prove equivalences. The results show that the most common use of `autorewrite` to solve `cast` problems can be replaced with our system, where our system can solve strictly more problems. This is because our system can deal with counteracting lemmas that cause `autorewrite` to go into an infinite loop. Further, our system is comparably fast compared to `autorewrite` and outperforms CoqHammer again.

Overall, this leads us to the following takeaways:

- Verifying graphical languages categorically with an eye towards semantics is a fruitful endeavor. Together with visualization, working in such a language is ergonomic and allows us to verify even complex statements.
- Domain-specific e-graph-based automation shows clear efficacy to solve common classes of problems both quickly and effectively.
- e-graphs are an exciting tool that can work well with proof assistants to automate repetitive, yet non-trivial tasks
- Domain specific or other high-level automation can be practically fruitful, even if not theoretically optimal.

APPENDIX A

EXPERIMENTAL SETUP

In the following we show the *Vy*ZX lemmas and problems for the cast and associativity/distributivity problems from Chapter 5. All statements are rendered are directly output using ZXVIZ.

A.1 Cast experiment

A.1.1 Lemmas

We split the lemmas here in two sections: lemmas that are passed to autorewrite and the e-graph system in Appendix A.1.2 and lemmas that cause autorewrite to loop forever and are hence only passed to the e-graph system in Appendix A.1.3.

A.1.2 autorewrite & e-graph

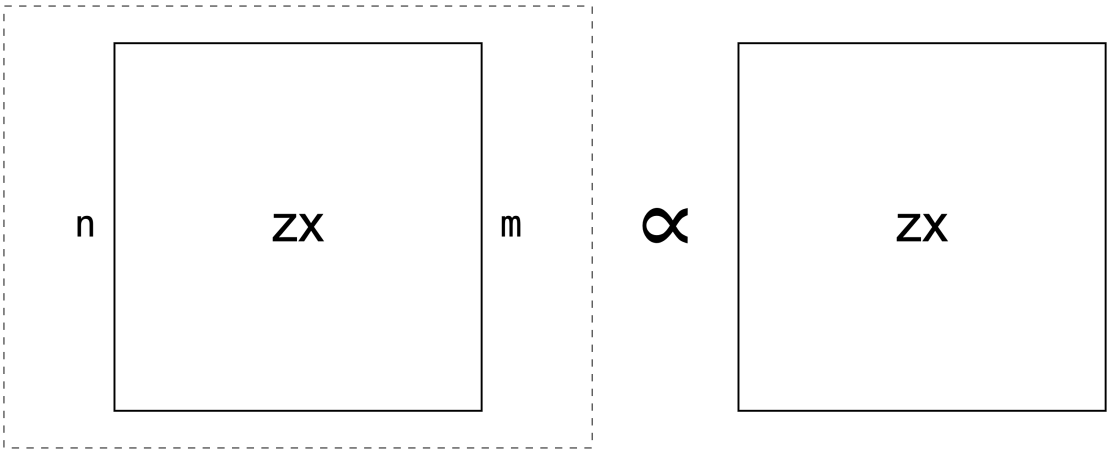


Figure A.1: Cast lemma: `cast_id`. If casting to the underlying dimension

eval/lemmas/castdbthe cast is optional..png

Figure A.2: Cast lemma: `the cast is optional.` . the cast is optional.

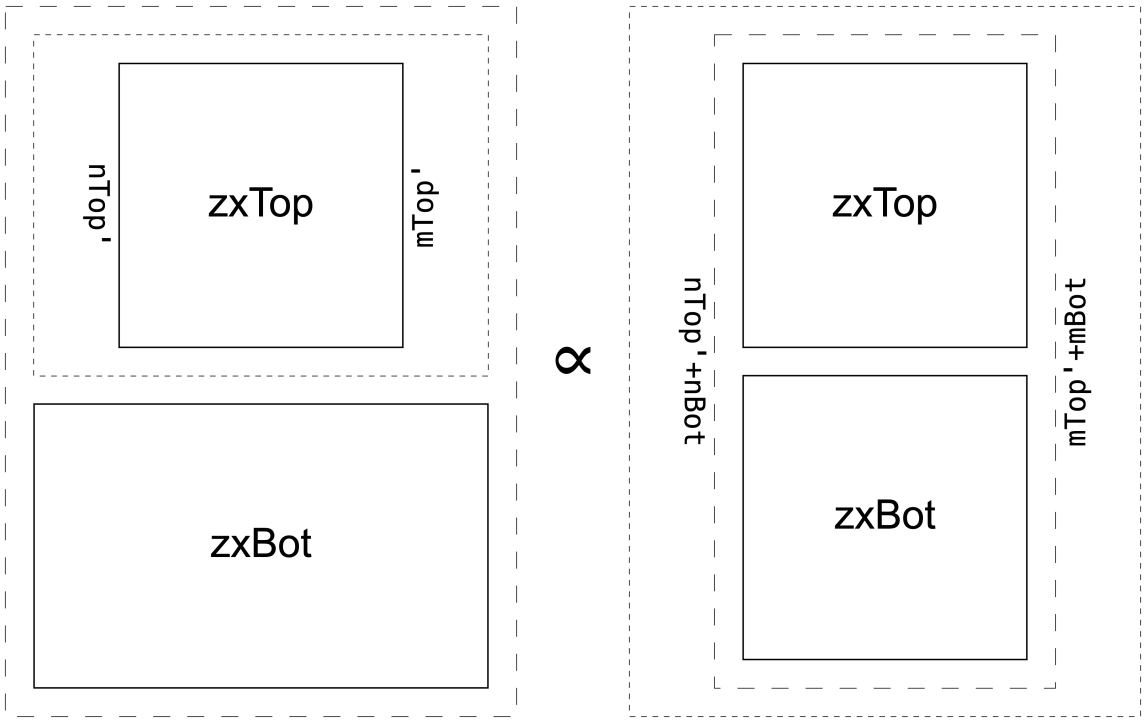


Figure A.3: Cast lemma: `cast_stack_l`. Casting the top of the stack is the same as casting the entire stack.

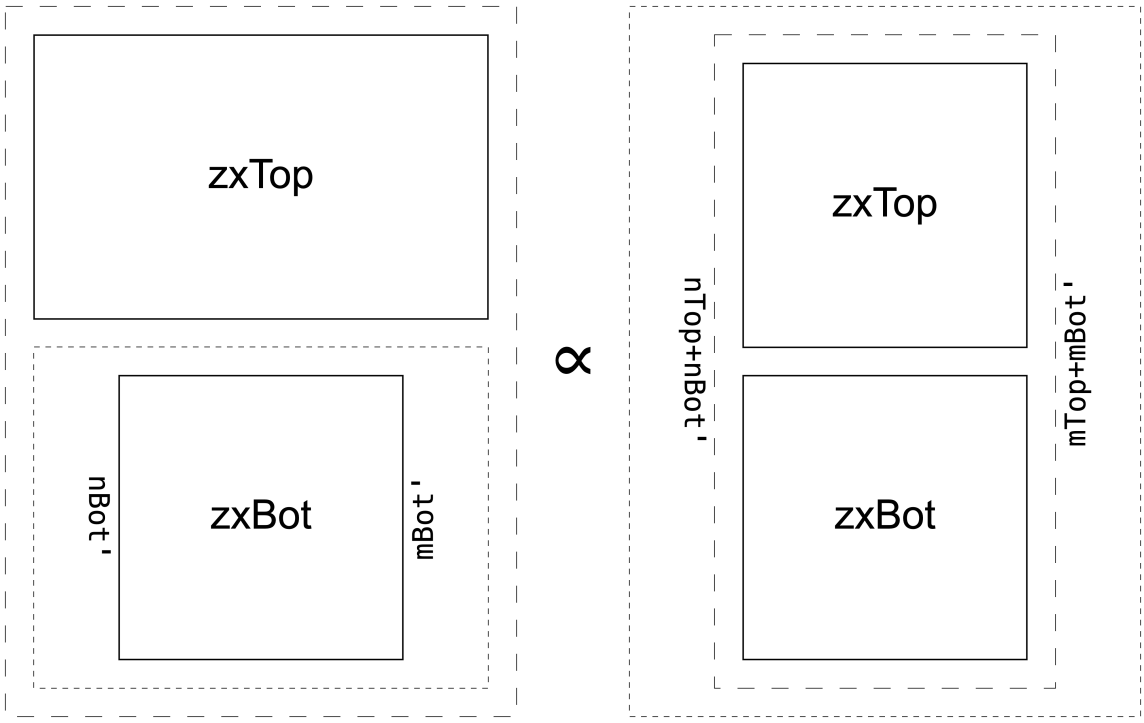


Figure A.4: Cast lemma: `cast_stack_r`. Casting the bottom of the stack is the same as casting the entire stack.

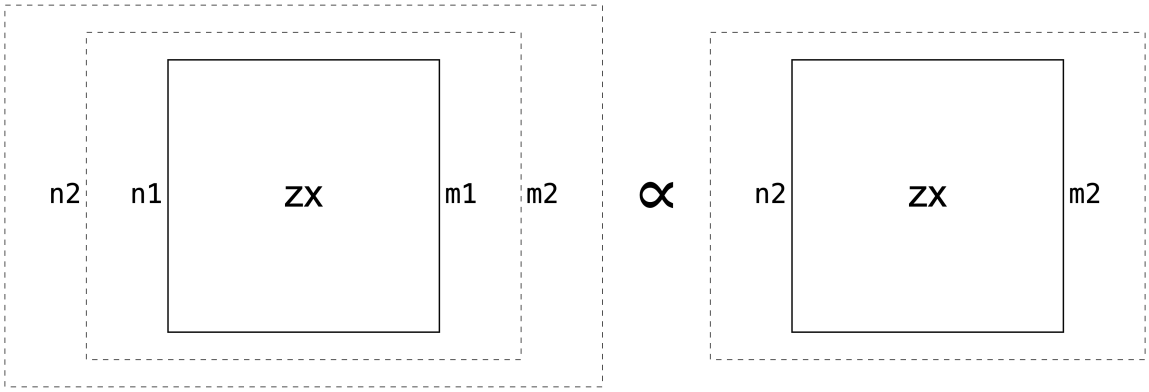


Figure A.5: Cast lemma: `cast_contract_l`. Contracting a cast to the left side.

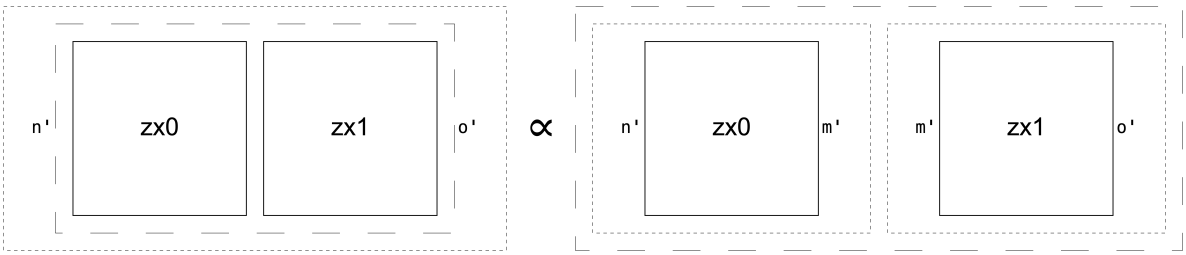


Figure A.6: Cast lemma: `cast_compose_mid_contract`. Casting a compose is equivalent to casting its left and right side.

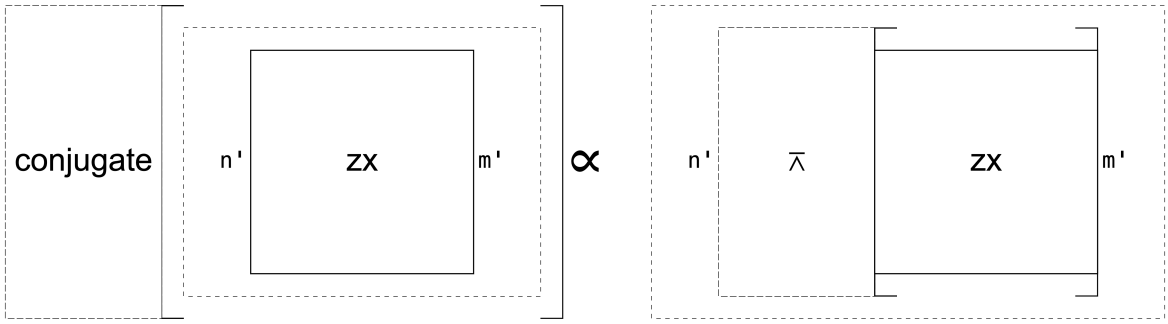


Figure A.7: Cast lemma: `cast_conj`. Casting a conjugation operation is the same as casting the argument.

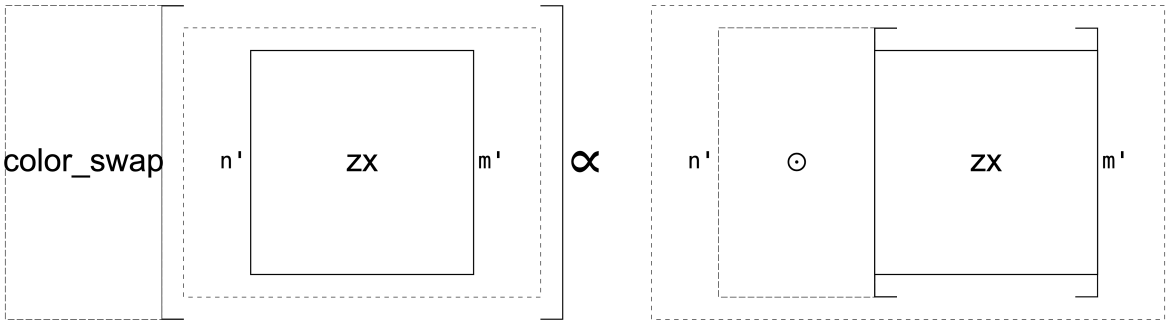


Figure A.8: Cast lemma: `cast_colorswap`. Casting a color swap function is the same as casting the argument.

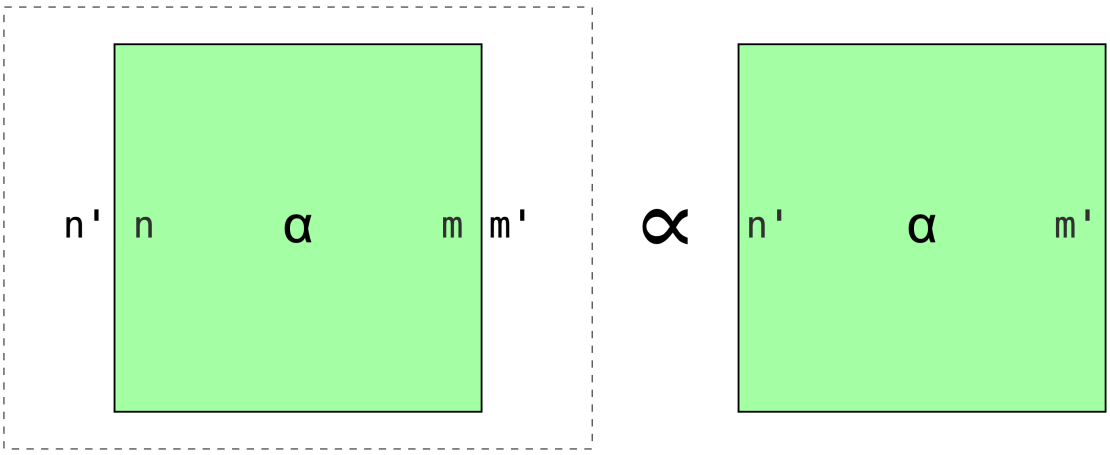


Figure A.9: Cast lemma: `cast_Z`. Casting a Z-spider is equal to changing the arguments.

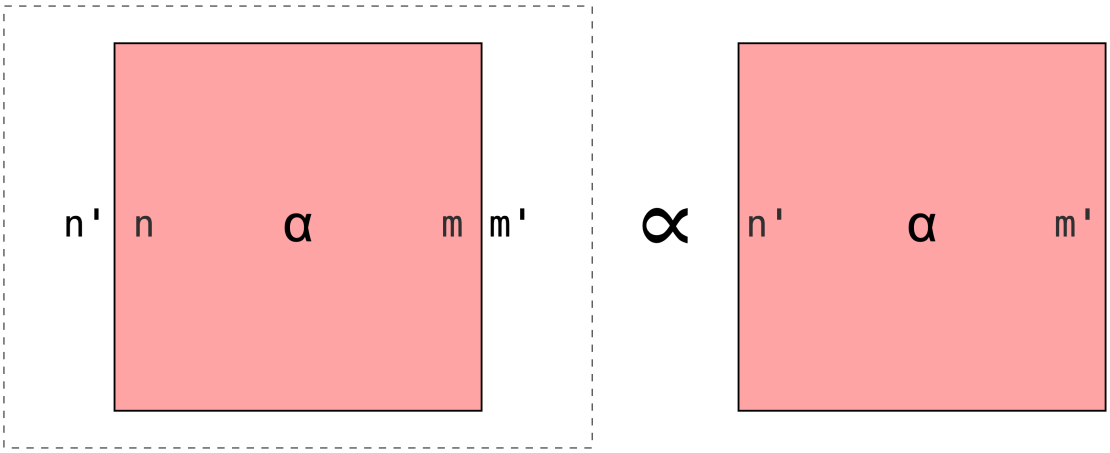


Figure A.10: Cast lemma: `cast_X`. Casting an X-spider is equal to changing the arguments.

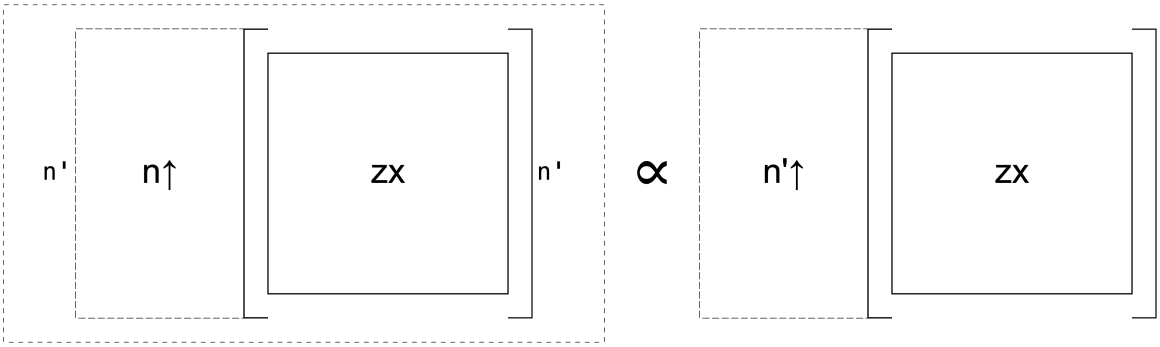


Figure A.11: Cast lemma: `cast_nstack_1`. Casting a `n_stack_1` is equal to changing the argument.

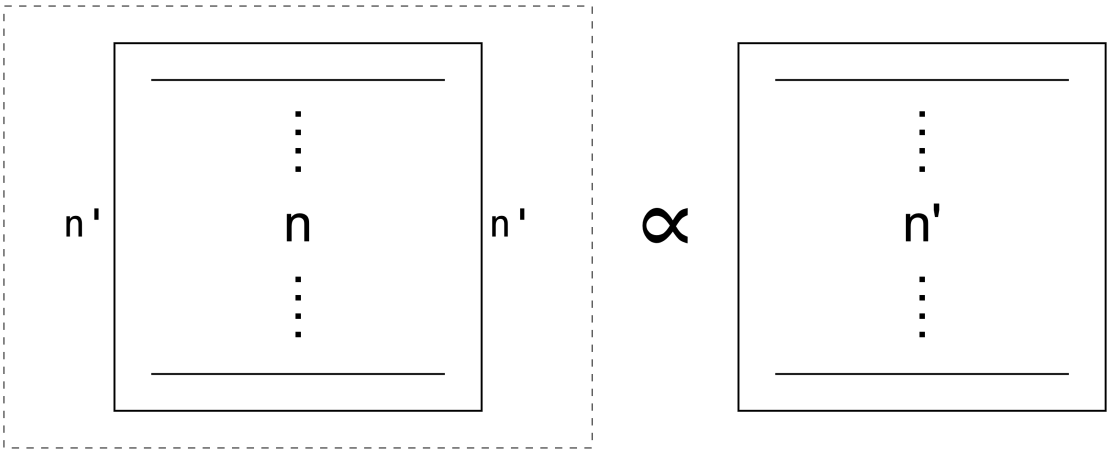


Figure A.12: Cast lemma: `cast_n_wire`. Casting a single n-wire is equal to changing the argument.

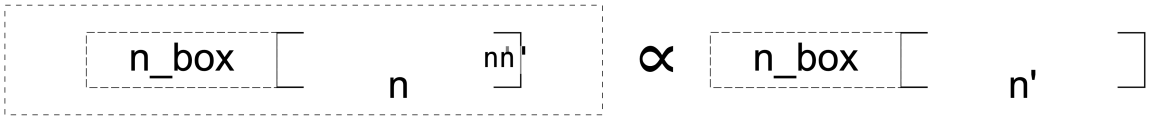


Figure A.13: Cast lemma: `cast_n_box`. Casting a n-box is equal to changing the argument.

A.1.3 e-graph only

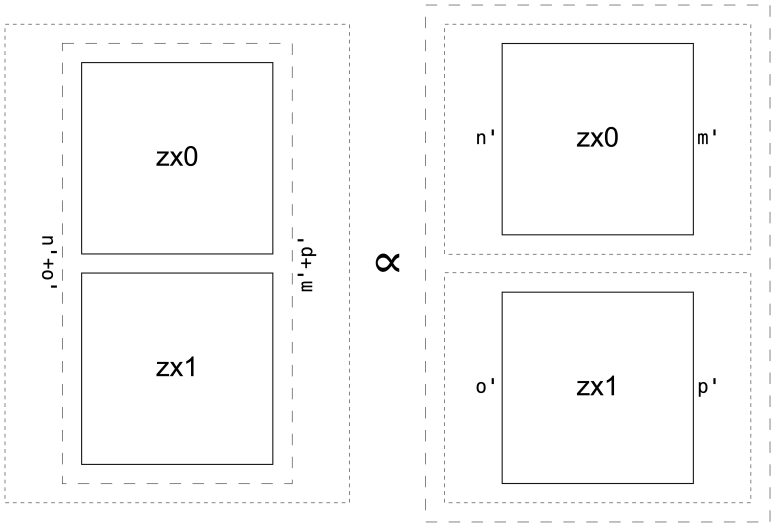


Figure A.14: Extended cast lemma: `cast_stack_distribute`. casting a stack is the same as casting its components if the target dimensions are of the form shown.

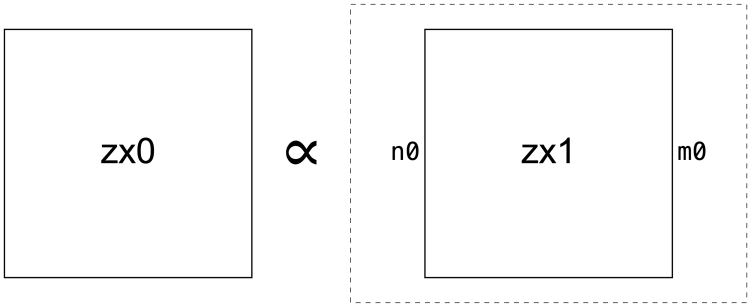


Figure A.15: Extended cast lemma: `cast_contract_r`. Contracting a cast to the right side.

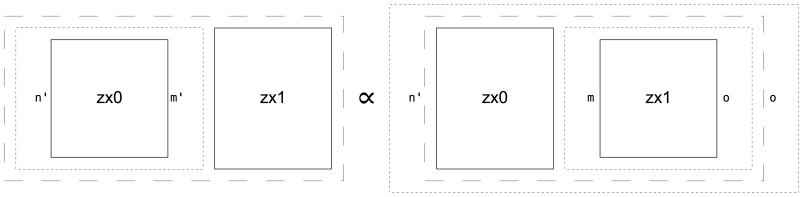


Figure A.16: Extended cast lemma: `cast_compose_l`. We can move the cast of the left side of a composition to the right side

eval/lemmas/castdbeif we also cast the entire composition..png

Figure A.17: Extended cast lemma: `if we also cast the entire composition..` if we also cast the entire composition.

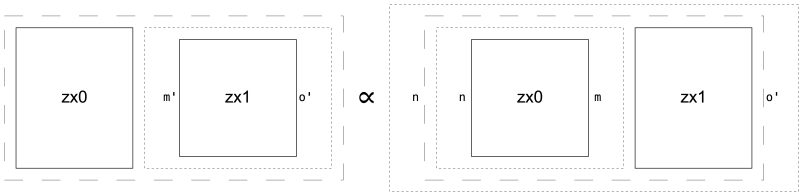


Figure A.18: Extended cast lemma: `cast_compose_r`. We can move the cast of the right side of a composition to the left side

eval/lemmas/castdbeif we also cast the entire composition. .png

Figure A.19: Extended cast lemma: `if we also cast the entire composition.  .` if we also cast the entire composition.

A.1.4 Problems

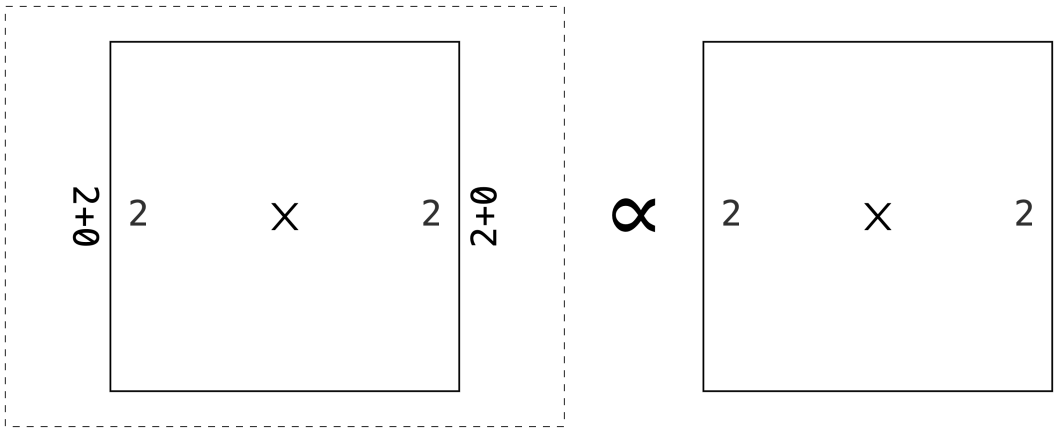


Figure A.20: Cast problem 0 from `a_swap_2_is_swap` (Base case to verify the set up).

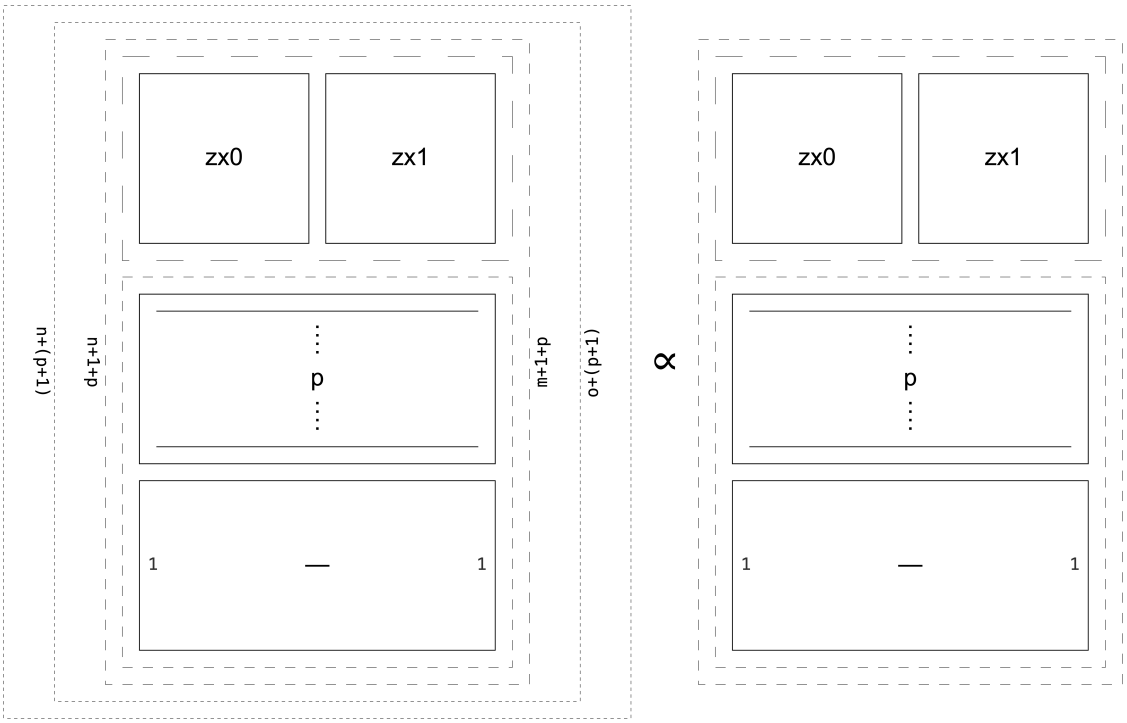


Figure A.21: Cast problem 1 from `Z_copy`.

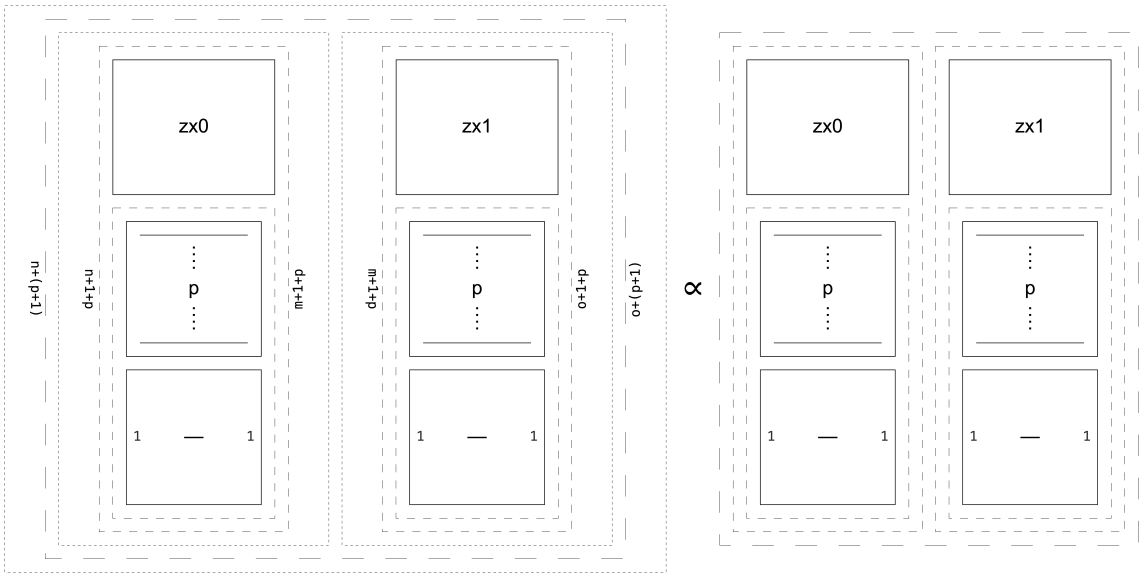


Figure A.22: Cast problem 2 from `X_state_copy`.

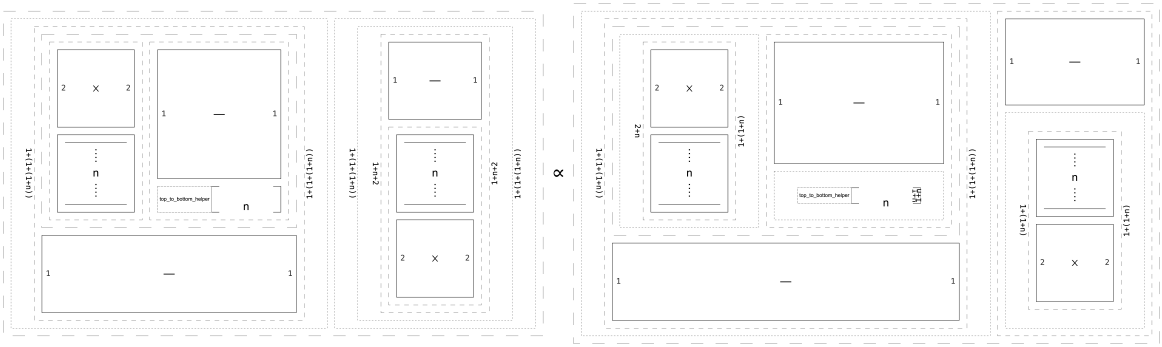


Figure A.23: Cast problem 3 from `top_to_bottom_grow_r`.

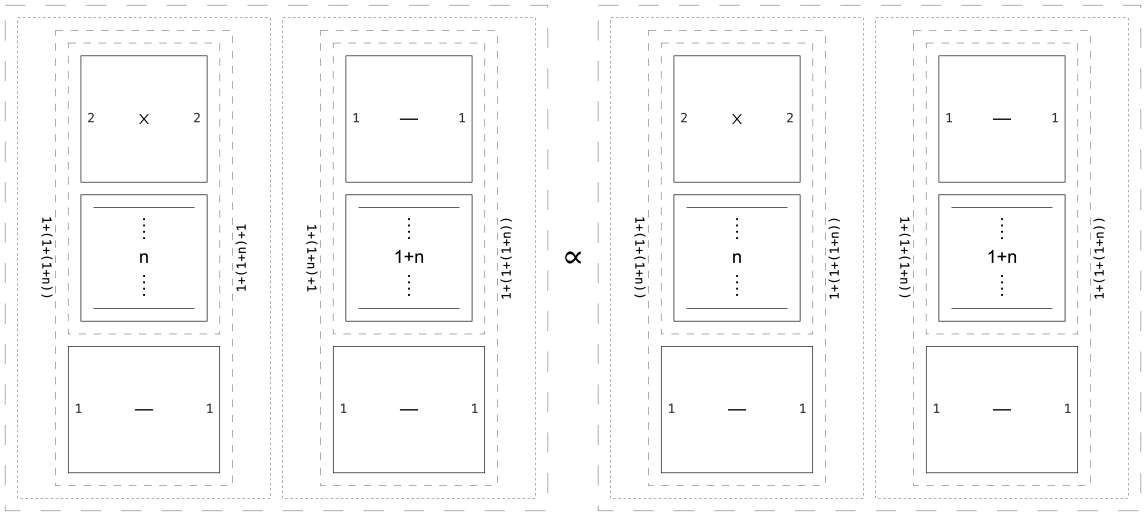


Figure A.24: Cast problem 4 from `top_to_bottom_grow_r` (distinct from Figure A.24).

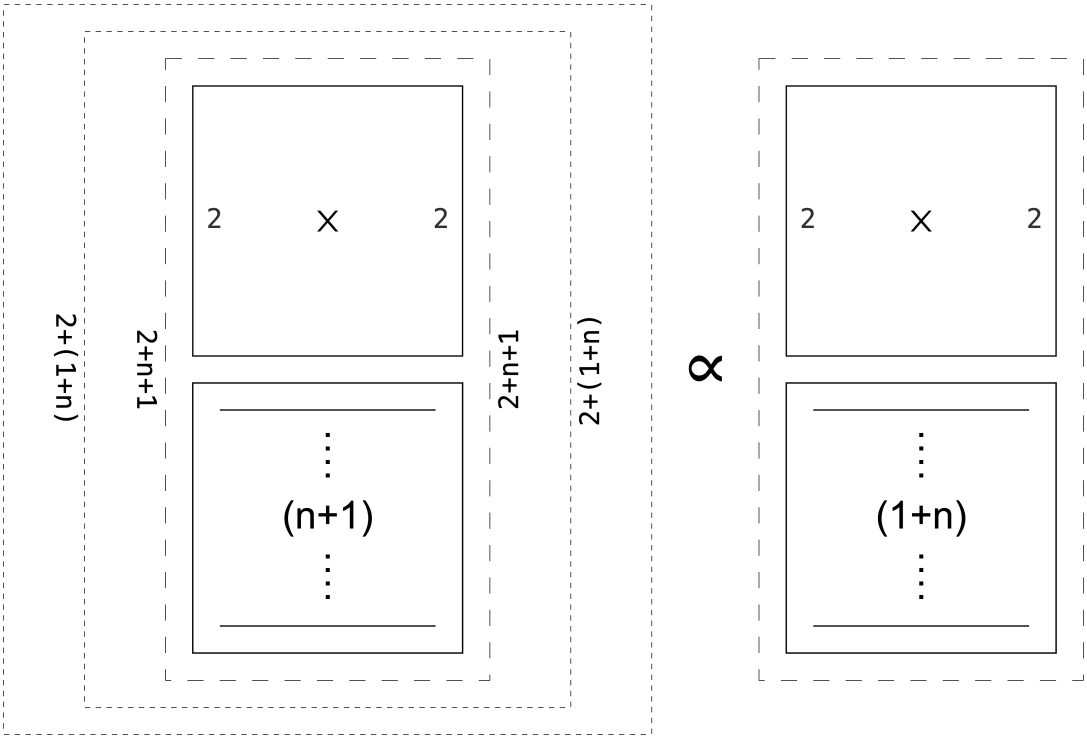


Figure A.25: Cast problem 5 from `a_swap_grow`.

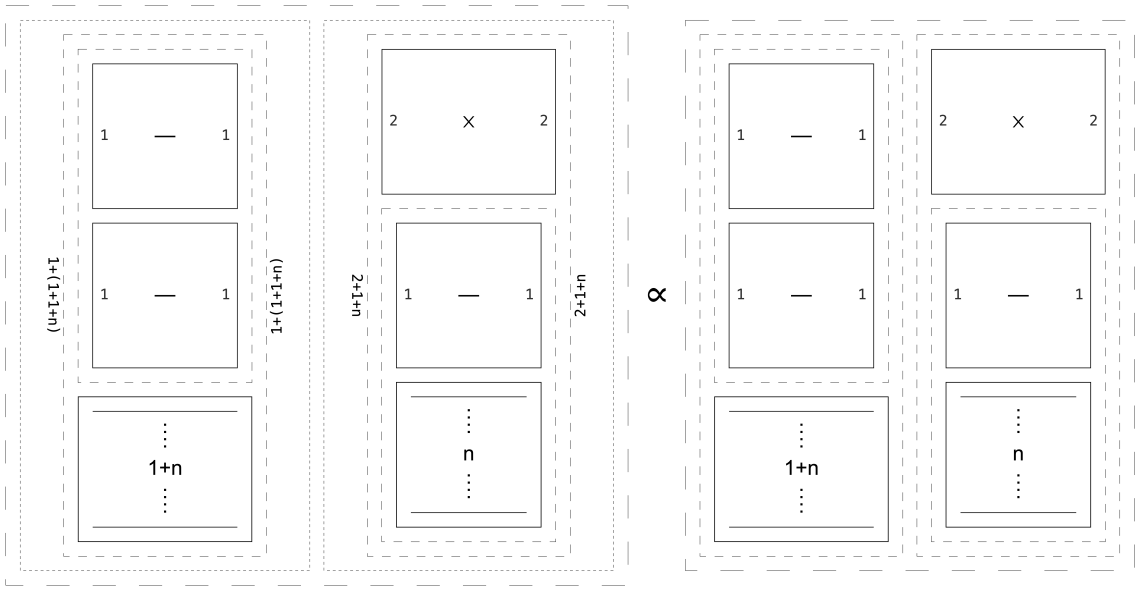


Figure A.26: Cast problem 6 from `Z_self_top_to_bottom_absorbtion_right_base`.

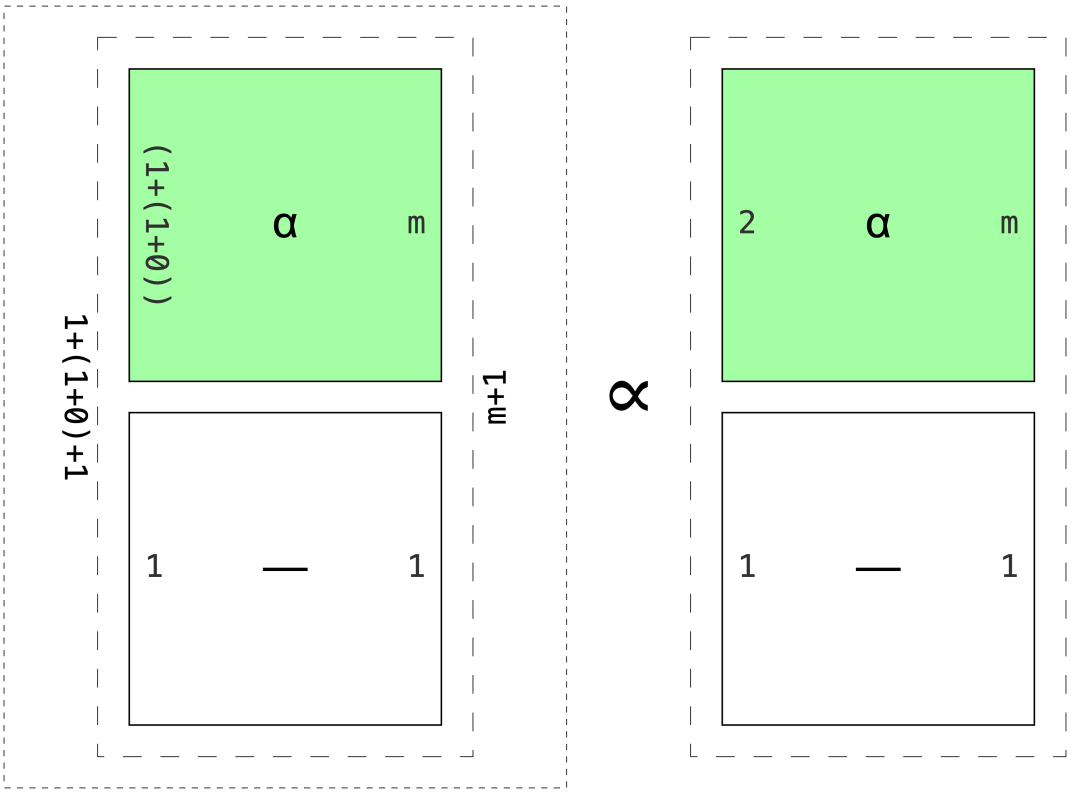


Figure A.27: Cast problem 7 from `Z_spider_fusion_bot_left_top_right`.

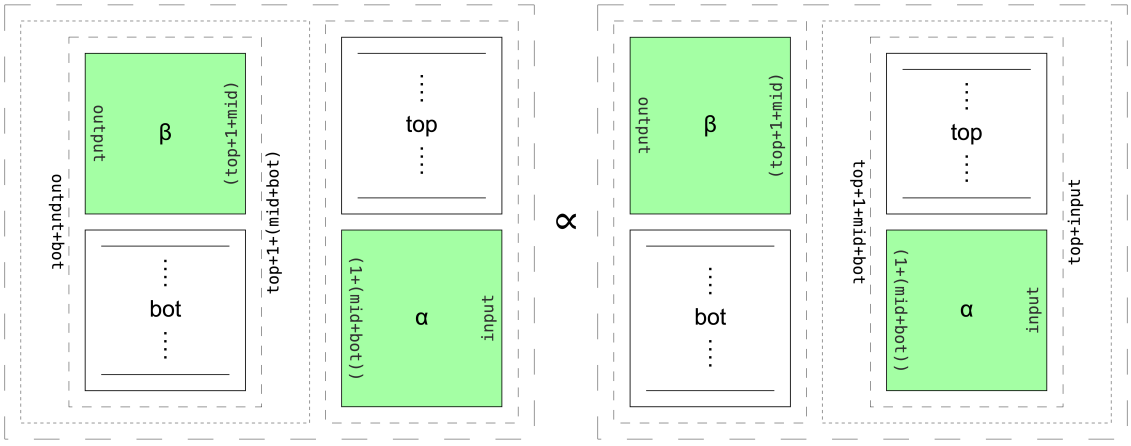


Figure A.28: Cast problem 8 from `Z_wrap_under_bot_left`.

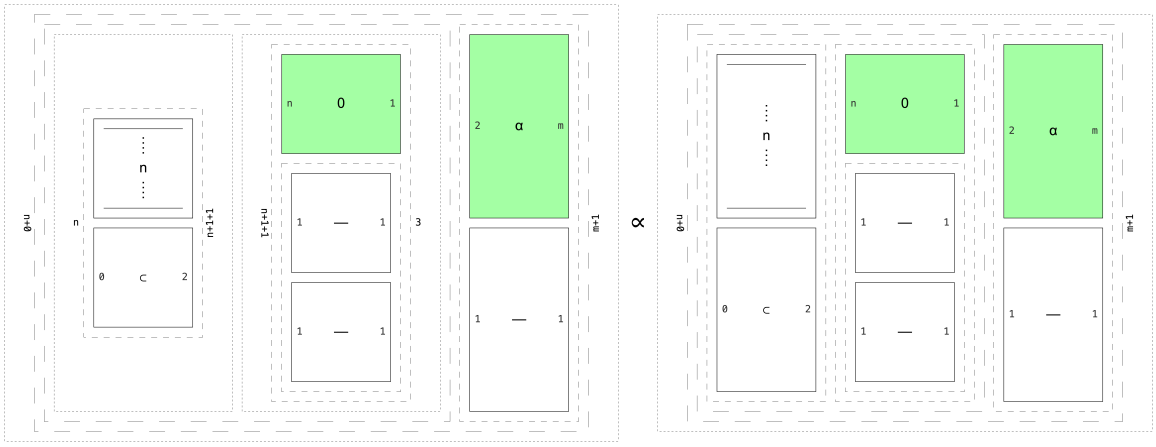


Figure A.29: Cast problem 9 from `Z_self_top_to_bottom_absorbtion_right_base` (distinct from Figure A.27). .

A.2 Associativity/Distributivity (AD)

A.2.1 Lemmas

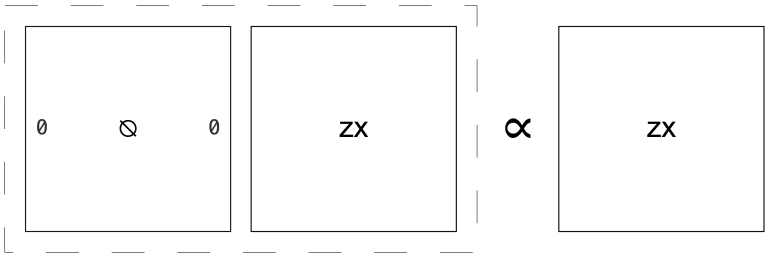


Figure A.30: AD lemma: `compose_empty_l`. Removal of empty diagrams in composition.

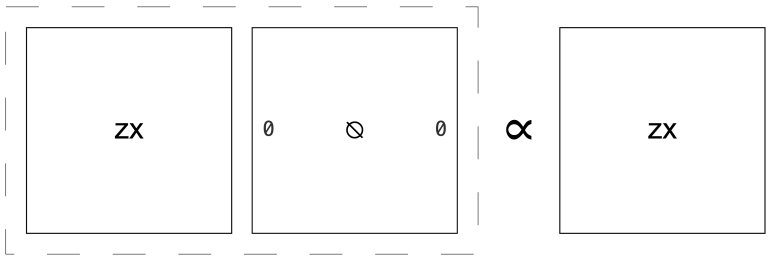


Figure A.31: AD lemma: `compose_empty_r`. Removal of empty diagrams in composition.

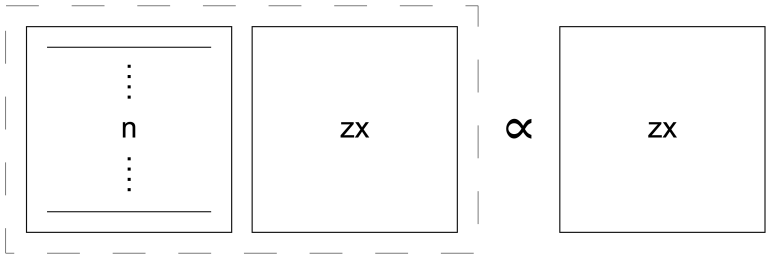


Figure A.32: AD lemma: `n_wire_removal_l`. Removal of wires in composition.

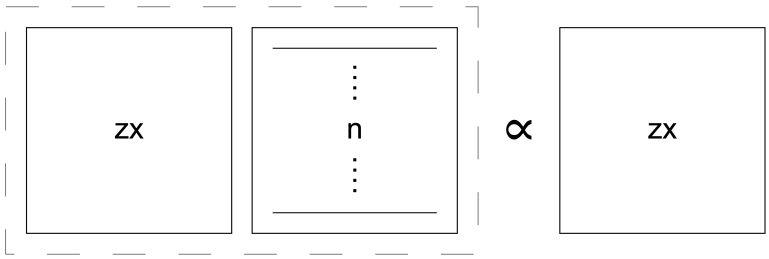


Figure A.33: AD lemma: `n_wire_removal_r`. Removal of wires in composition.

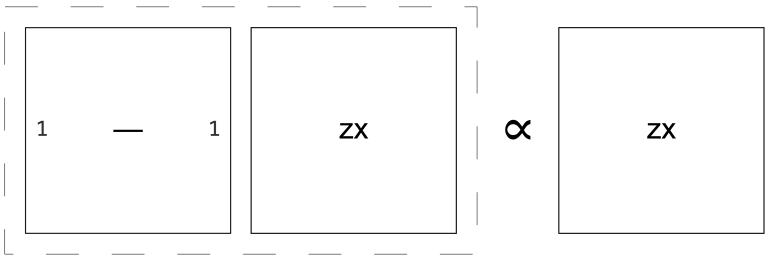


Figure A.34: AD lemma: `wire_removal_l`. Removal of wires in composition.

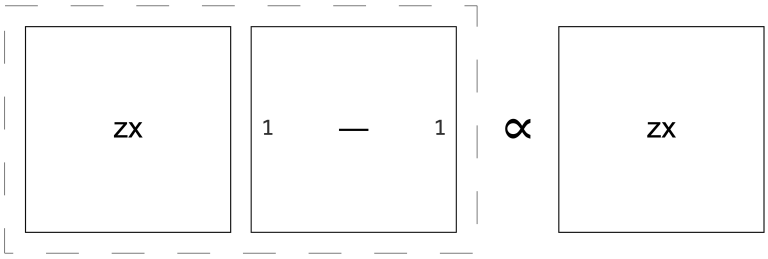


Figure A.35: AD lemma: `wire_removal_r`. Removal of wires in composition.

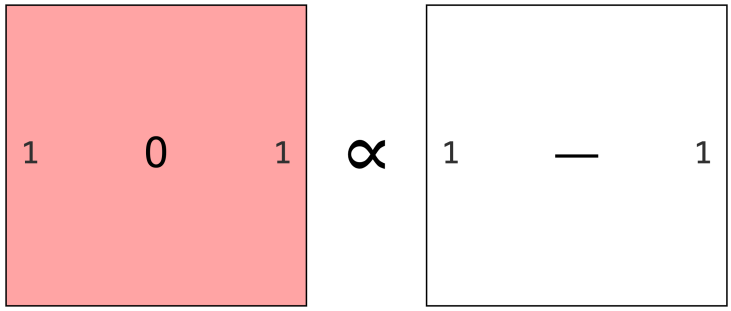


Figure A.36: AD lemma: `X_0_is_wire`. Converting a spider to a wire.

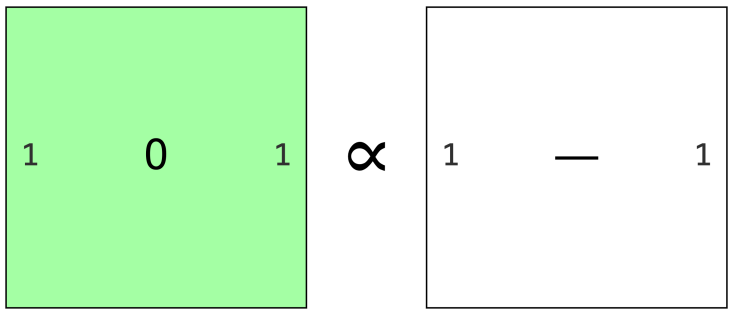


Figure A.37: AD lemma: `Z_0_is_wire`. Converting a spider to a wire.

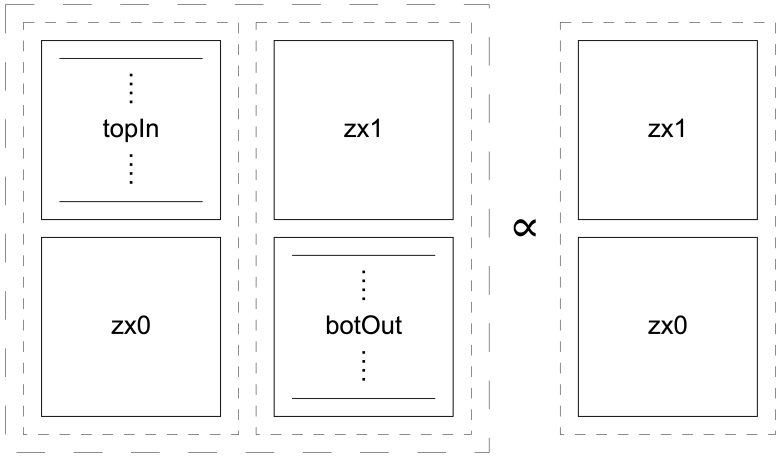


Figure A.38: AD lemma: `nwire_stack_compose_topleft`. Fusing wires diagonally.

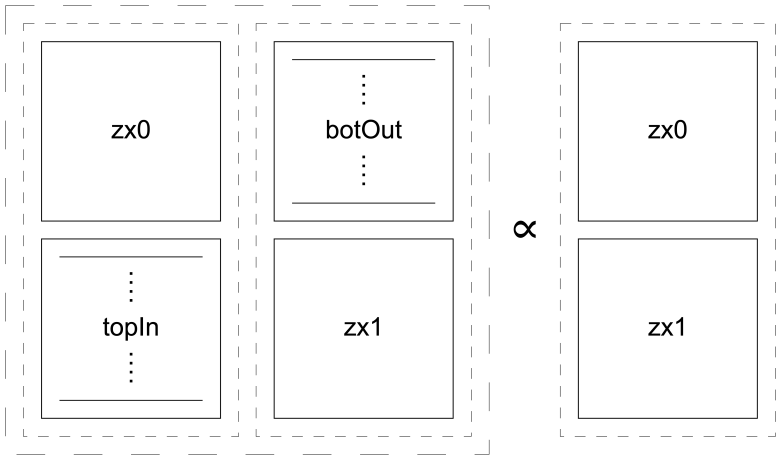


Figure A.39: AD lemma: `nwire_stack_compose_botleft`. Fusing wires diagonally.

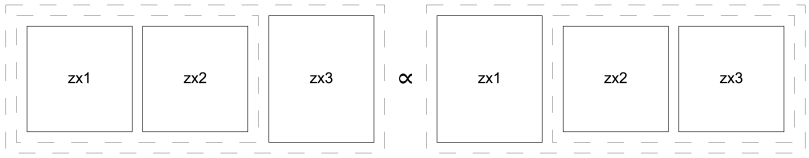


Figure A.40: AD lemma: `compose_assoc`. Compositional associativity.

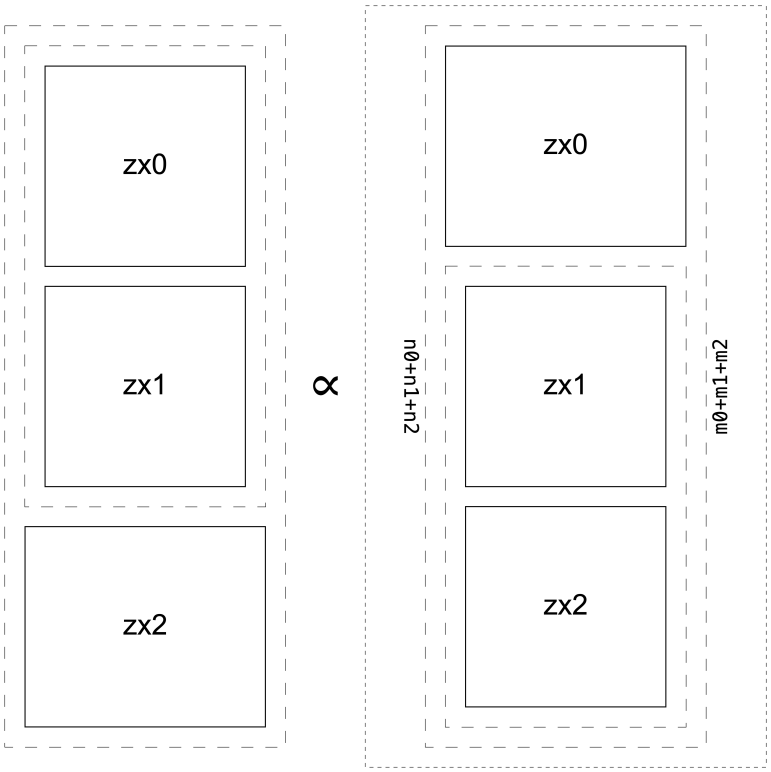


Figure A.41: AD lemma: `stack_assoc`. Stack associativity. Note the cast.

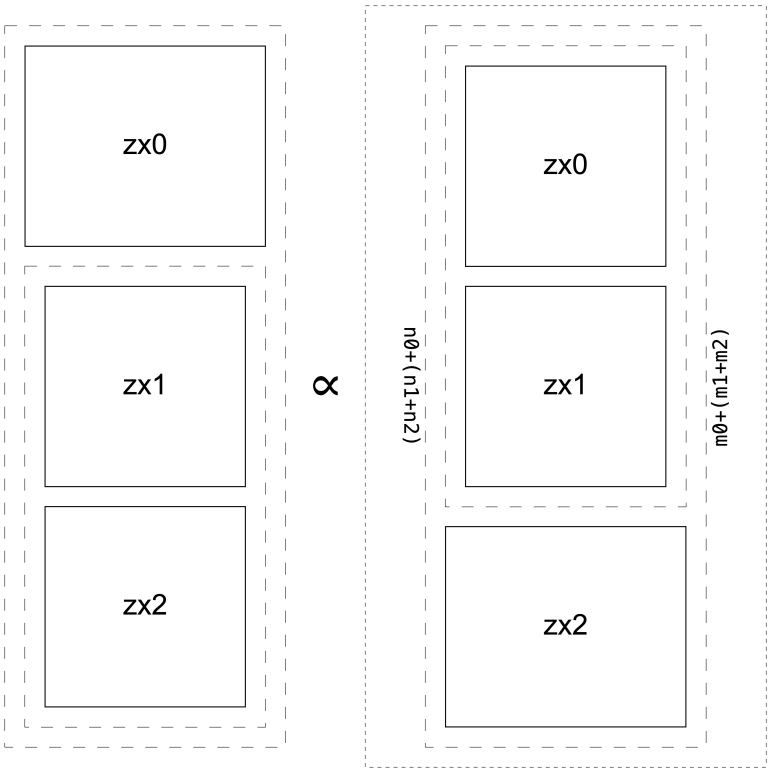


Figure A.42: AD lemma: `stack_assoc_back (Reverse of Figure A.41)`. Stack associativity. Note the cast.

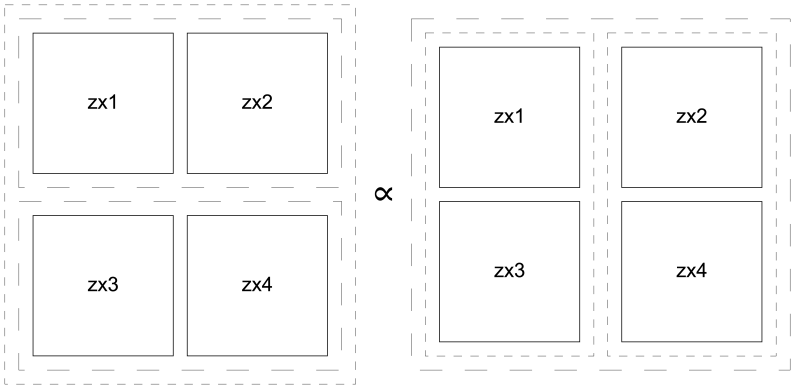


Figure A.43: AD lemma: `stack_compose_distr`. Structural relation of stack and compose.

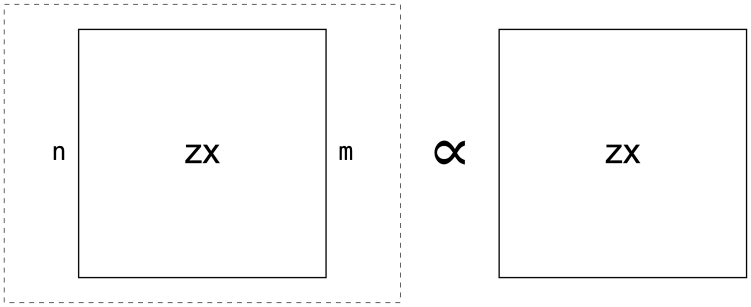


Figure A.44: AD lemma: `cast_id`. Casts that do not change dimensions are not needed.

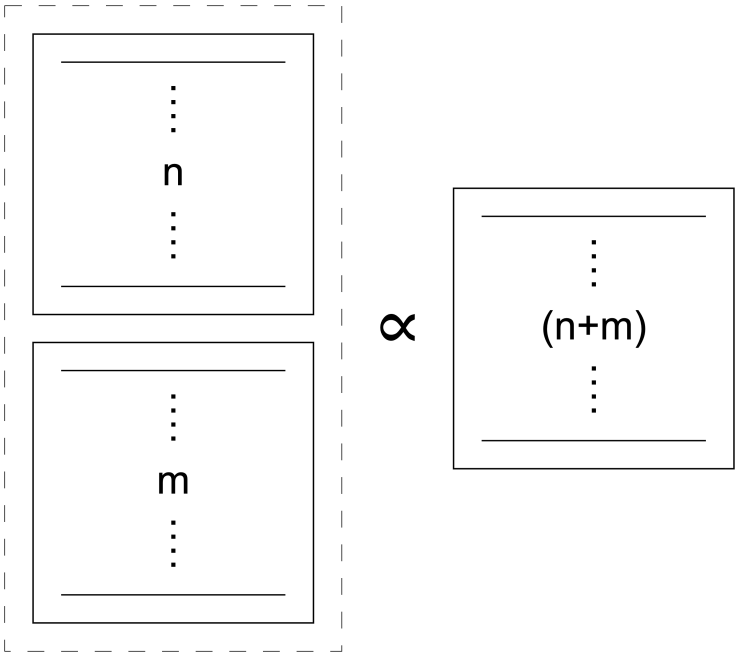


Figure A.45: AD lemma: `n_wire_stack`. Stacked `n_wires` contract.

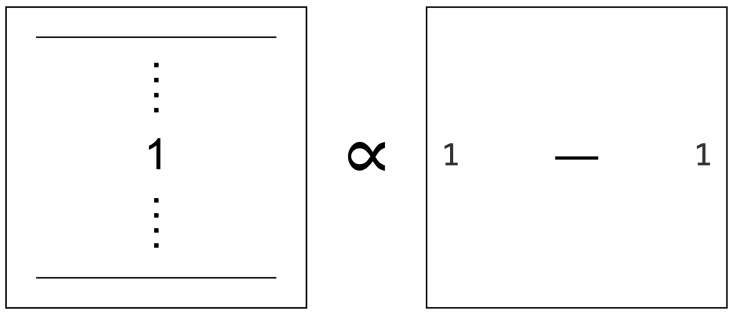


Figure A.46: AD lemma: ← `wire_to_n_wire`. Converts (n_wire 1) to wires.

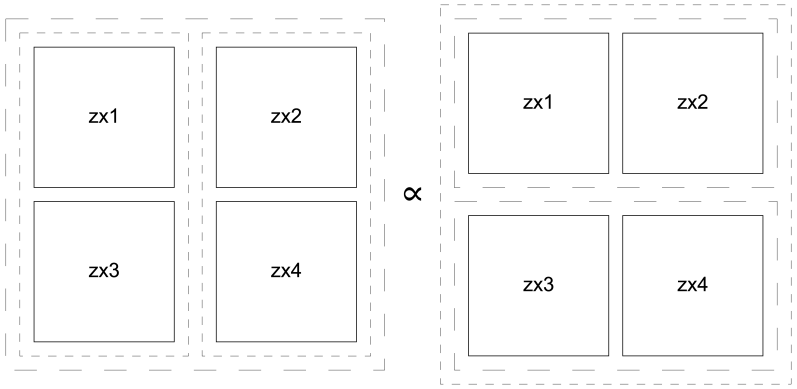


Figure A.47: AD lemma: `stack_compose_distr_back`. Reverse of Figure A.43.

A.2.2 Problems

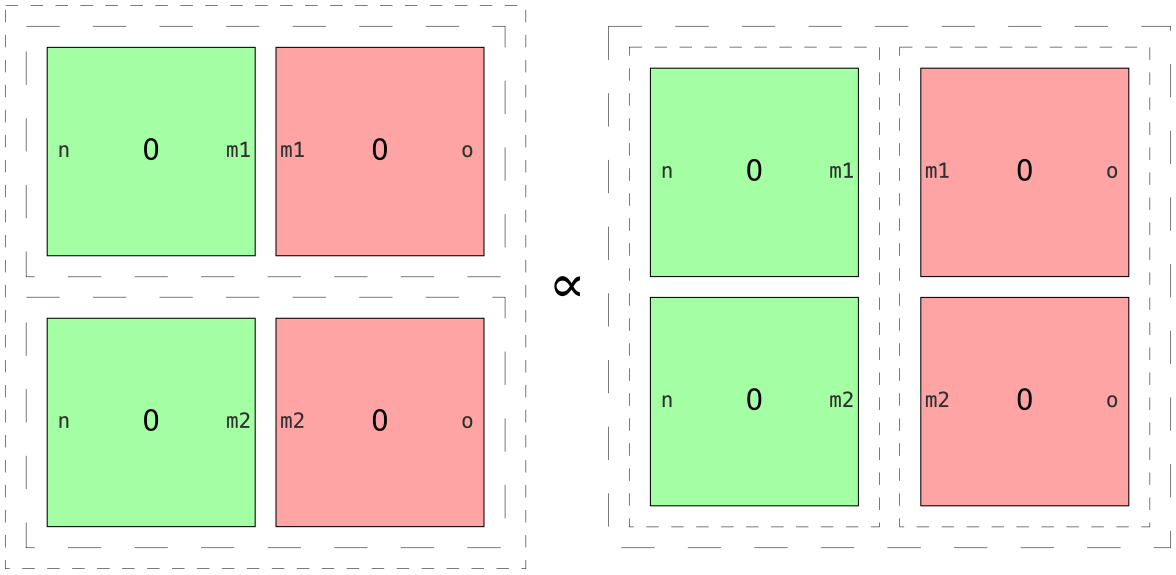


Figure A.48: AD problem 0. Structural rewrites in `a_swap_2_is_swap`.

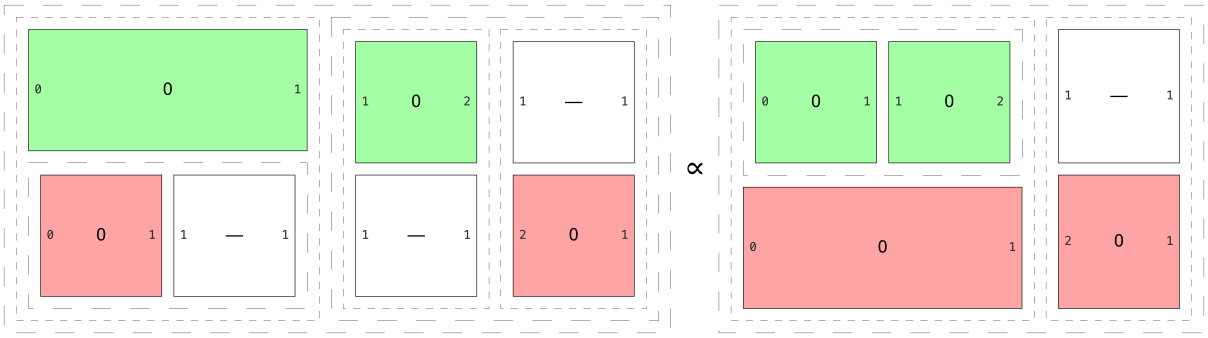


Figure A.49: AD problem 1. Structural rewrites in `bell_state_prep_correct`.

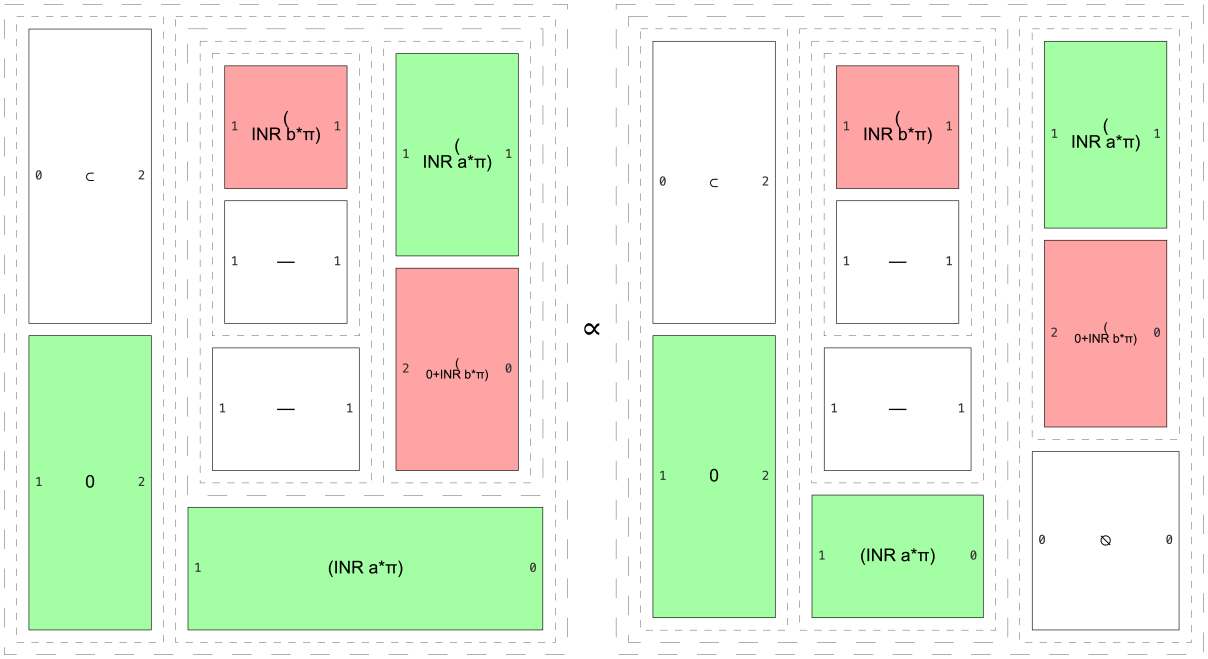


Figure A.50: AD problem 2. Structural rewrites in `teleportation_correct`.

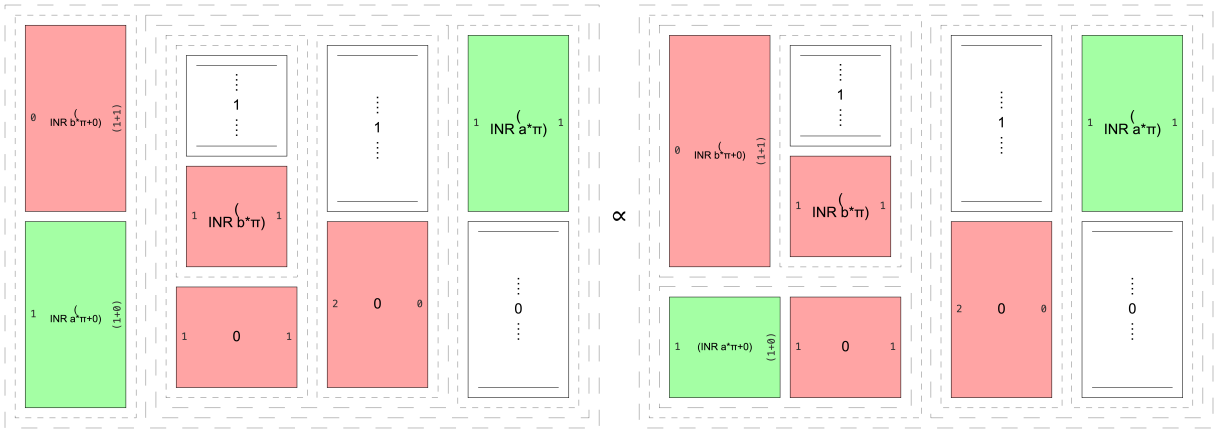


Figure A.51: AD problem 3. Structural rewrites in `teleportation_correct` (separate structural problem to Figure A.51).

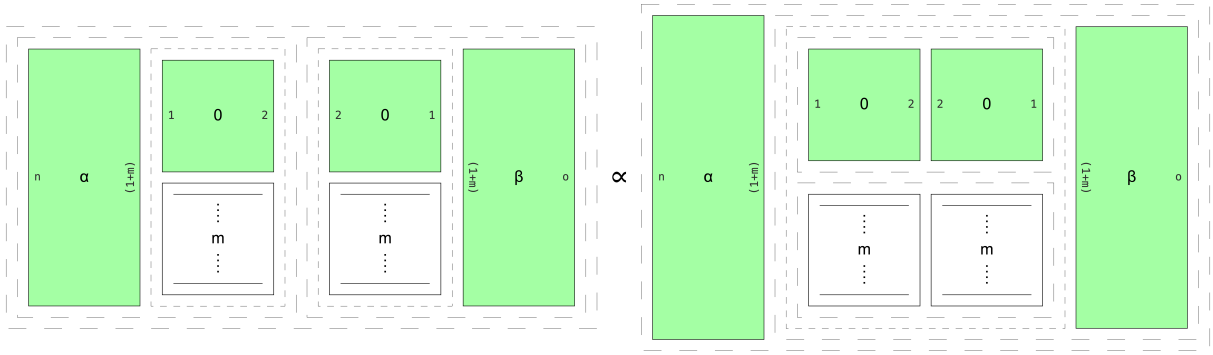


Figure A.52: AD problem 4. Structural rewrites in `Z_absolute_fusion`.

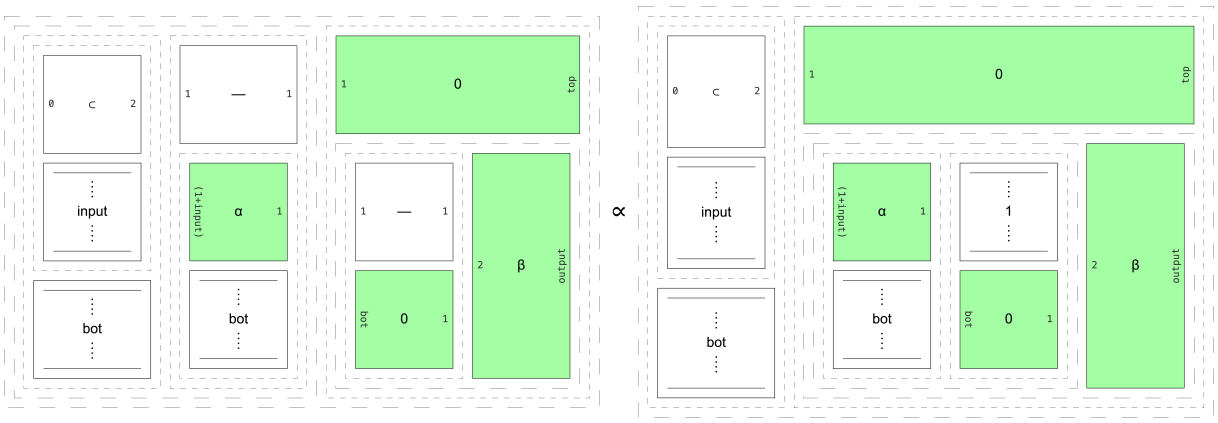


Figure A.53: AD problem 5. Structural rewrites in `Z_spider_fusion_top_left_bot_right`.

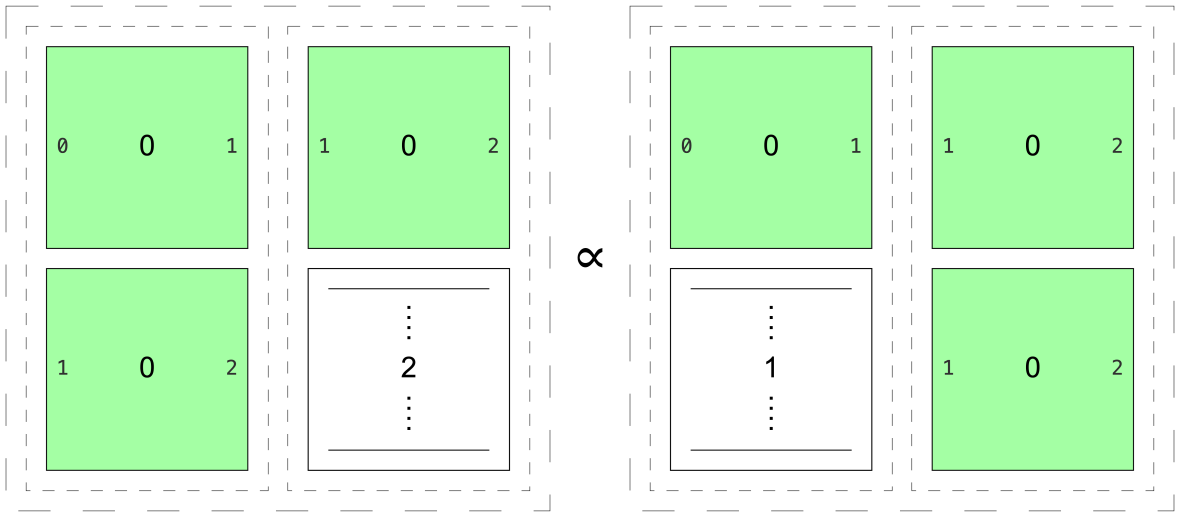


Figure A.54: AD problem 6. Structural rewrites in `hopf_rule_Z_X`.

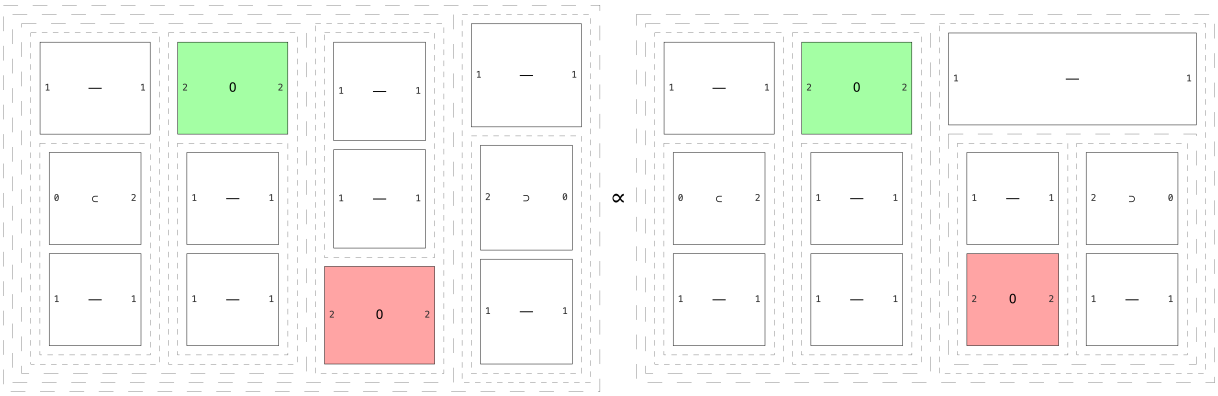


Figure A.55: AD problem 7. Structural rewrites in `HBiAlgAssoc` (A reassociated bi-algebra rule).

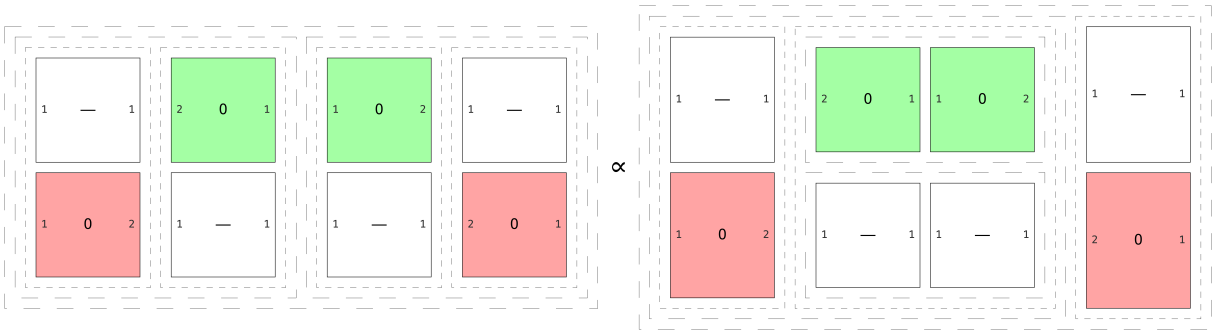


Figure A.56: AD problem 8. Structural rewrites in `cnot_involutive`.

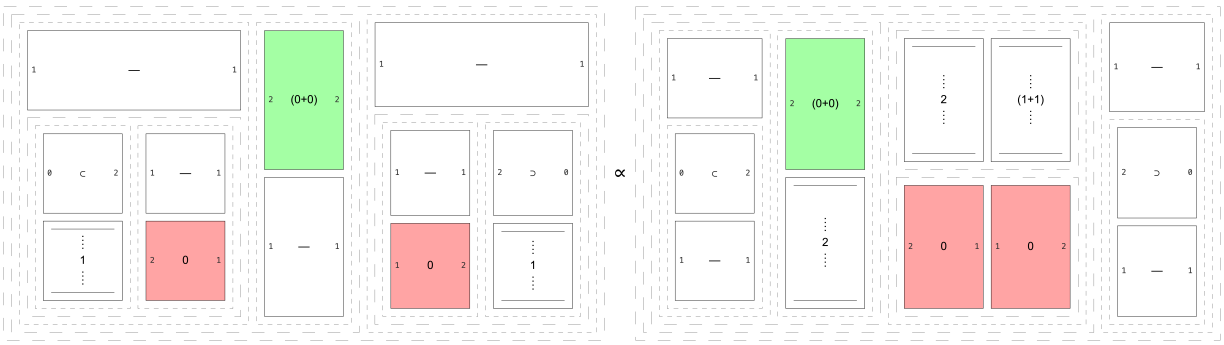


Figure A.57: AD problem 9. Structural rewrites in `cnot_involutive` (separate structural problem to Figure A.56).

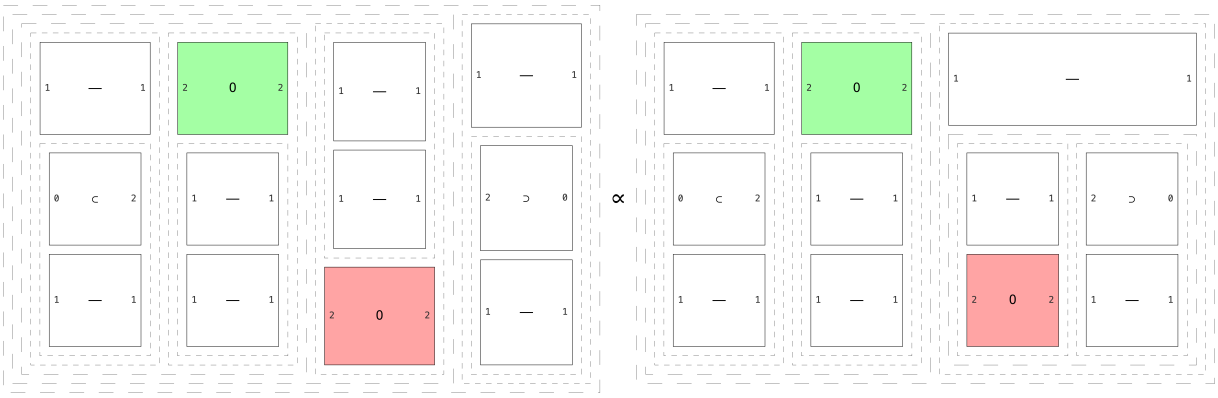


Figure A.58: AD problem 10. Structural rewrites in `cnot_involutive` (separate structural problem to Figures A.56 and A.57). .

REFERENCES

Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., USA, 1972. ISBN 0139145567.

AlphaProof-Team and AlphaGeometry-Team. Ai achieves silver-medal standard solving international mathematical olympiad problems, Jul 2024. URL https://deepmind.google/discover/blog/ai-solves-imo-problems-at-silver-medal-level/.

Kenneth Appel and Wolfgang Haken. The solution of the four-color-map problem. *Scientific American*, 237(4):108–121, 1977. ISSN 00368733, 19467087. URL http://www.jstor.org/stable/24953967.

Miriam Backens and Aleks Kissinger. Zh: A complete graphical calculus for quantum computations involving classical non-linearity. *arXiv preprint arXiv:1805.02175*, 2018.

Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing tls with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy*, pages 445–459, 2013. doi:10.1109/SP.2013.37.

N. L. Biggs. De morgan on map colouring and the separation axiom. *Archive for History of Exact Sciences*, 28(2):165–170, 1983. ISSN 1432-0657. doi:10.1007/bf00327701. URL http://dx.doi.org/10.1007/BF00327701.

F. Bloch. Nuclear induction. *Phys. Rev.*, 70:460–474, Oct 1946. doi:10.1103/PhysRev.70.460. URL https://link.aps.org/doi/10.1103/PhysRev.70.460.

Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement day. In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning*, pages 107–121, Berlin, Heidelberg, 2010a. Springer Berlin Heidelberg. ISBN 978-3-642-14203-1.

Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement day. In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning*, pages 107–121, Berlin, Heidelberg, 2010b. Springer Berlin Heidelberg. ISBN 978-3-642-14203-1.

Thomas Bourgeat. *Specification and verification of sequential machines in rule-based hardware languages*. Massachusetts Institute of Technology, 2023.

ChunJun Cao and Brad Lackey. Growing sparse quantum codes from a seed, 2025. URL https://arxiv.org/abs/2507.13496.

CISA. Openssl "heartbleed" vulnerability (cve-2014-0160): Cisa, Oct 2016. URL https://www.cisa.gov/news-events/alerts/2014/04/08/openssl-heartbleed-vulnerability-cve-2014-0160.

Bob Coecke and Ross Duncan. A graphical calculus for quantum observables. *Preprint*, 2007. URL https://www.cs.ox.ac.uk/people/bob.coecke/GreenRed.pdf.

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

Pierre Corbineau. Deciding equality in the constructor theory. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs*, pages 78–92, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-74464-1.

Łukasz Czajka. Practical proof search for coq by type inhabitation. In *Automated Reasoning: 10th International Joint Conference, IJCAR 2020, Paris, France, July 1–4, 2020, Proceedings, Part II*, page 28–57, Berlin, Heidelberg, 2020. Springer-Verlag. ISBN 978-3-030-51053-4. doi:10.1007/978-3-030-51054-1_3. URL https://doi.org/10.1007/978-3-030-51054-1_3.

Łukasz Czajka and Cezary Kaliszyk. Hammer for coq: Automation for dependent type theory. *Journal of automated reasoning*, 61(1):423–453, 2018.

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

P. A. M. Dirac. A new notation for quantum mechanics. *Mathematical Proceedings of the Cambridge Philosophical Society*, 35(3):416–418, 1939. doi:10.1017/S0305004100021162.

Christian Doczkal and Damien Pous. Completeness of an axiomatization of graph isomorphism via graph rewriting in coq. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, POPL '20, page 325–337. ACM, January 2020. doi:10.1145/3372885.3373831. URL http://dx.doi.org/10.1145/3372885.3373831.

Ross Duncan, Aleks Kissinger, Simon Perdrix, and John van de Wetering. Graph-theoretic simplification of quantum circuits with the zx-calculus. *Quantum*, 4:279, June 2020. ISSN 2521-327X. doi:10.22331/q-2020-06-04-279. URL http://dx.doi.org/10.22331/q-2020-06-04-279.

Oliver Flatt, Samuel Coward, Max Willsey, Zachary Tatlock, and Pavel Panchekha. Small proofs from congruence closure, 2022. URL https://arxiv.org/abs/2209.03398.

Emilio Jesús Gallego Arias, Ali Caglayan, Ramkumar Ramachandra, and 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, 2023.

Antonio García de la Barrera, Ignacio García-Rodríguez de Guzmán, Macario Polo, and Mario Piattini. Quantum software testing: State of the art. *Journal of Software: Evolution and Process*, 35(4):e2419, 2023. doi:https://doi.org/10.1002/smr.2419. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2419.

Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. Quantum random access memory. *Physical Review Letters*, 100(16), April 2008. ISSN 1079-7114. doi:10.1103/physrevlett.100.160501. URL http://dx.doi.org/10.1103/PhysRevLett.100.160501.

Georges Gonthier et al. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11): 1382–1393, 2008.

Nadeeshaan Gunasinghe and Nipuna Marcus. *Language Server Protocol and Implementation*. Apress, 2022. doi:10.1007/978-1-4842-7792-8. URL http://dx.doi.org/10.1007/978-1-4842-7792-8.

Amar Hadzihasanovic. A diagrammatic axiomatisation for qubit entanglement. In *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 573–584. IEEE, 2015.

Amar Hadzihasanovic, Kang Feng Ng, and Quanlong Wang. Two complete axiomatisations of pure-state qubit quantum computing. In *Proceedings of the 33rd annual ACM/IEEE symposium on logic in computer science*, pages 502–511, 2018.

Kesha Hietala. *A Verified Software Toolchain for Quantum Programming*. PhD thesis, University of Maryland, 2022. URL https://drum.lib.umd.edu/handle/1903/29337.

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, 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, 2021b. 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 2021c. doi:10.1145/3434318.

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. A verified optimizer for quantum circuits. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021d.

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

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.

M. M. Kapranov and V. A. Voevodsky. ∞-groupoids and homotopy types. *Cahiers de Topologie et Géométrie Différentielle Catégoriques*, 32(1):29–46, 1991. URL http://eudml.org/doc/91469.

Aleks Kissinger. Phase-free zx diagrams are css codes (...or how to graphically grok the surface code), 2022. URL https://arxiv.org/abs/2204.14038.

Aleks Kissinger and John van de Wetering. Pyzx: Large scale automated diagrammatic reasoning. *arXiv preprint arXiv:1904.04735*, 2019.

Aleks Kissinger and John van de Wetering. Reducing the number of non-clifford gates in quantum circuits. *Physical Review A*, 102(2), August 2020. ISSN 2469-9934. doi:10.1103/physreva.102.022406. URL http://dx.doi.org/10.1103/PhysRevA.102.022406.

Aleks Kissinger and John van de Wetering. Simulating quantum circuits with zx-calculus reduced stabiliser decompositions. *Quantum Science and Technology*, 7(4):044001, July 2022. ISSN 2058-9565. doi:10.1088/2058-9565/ac5d20. URL http://dx.doi.org/10.1088/2058-9565/ac5d20.

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.

Andrei Kozyrev. Equality saturation for solving equalities of relational expressions. Bachelor's thesis, Constructor University Bremen, 2023.

Adrian Lehmann, Ben Caldwell, and Robert Rand. Vyzx : A vision for verifying the zx calculus, 2022. URL https://arxiv.org/abs/2205.05781.

Adrian Lehmann, Ben Caldwell, Bhakti Shah, and Robert Rand. Vyzx: Formal verification of a graphical quantum language, 2024. URL https://arxiv.org/abs/2311.11571.

Peiyi Li, Ji Liu, Yangjia Li, and Huiyang Zhou. Exploiting quantum assertions for error mitigation and quantum program debugging. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pages 124–131, 2022. doi:10.1109/ICCD56317.2022.00028.

Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing. ISBN 978-3-030-79876-5.

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, 2018a. doi:10.1038/s41534-018-0072-4.

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, 2018b. doi:10.1038/s41534-018-0072-4.

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.dagstuhl.de/opus/volltexte/2023/18399.

Charles Gregory Nelson. *Techniques for program verification*. PhD thesis, Stanford University, Stanford, CA, USA, 1980. AAI8011683.

Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, October 1979. ISSN 0164-0925. doi:10.1145/357073.357079. URL https://doi.org/10.1145/357073.357079.

Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010a.

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

Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In *International Conference on Rewriting Techniques and Applications*, 2005. URL https://api.semanticscholar.org/CorpusID:5253917.

Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL*. Springer Berlin Heidelberg, 2002. ISBN 9783540459491. doi:10.1007/3-540-45949-9. URL http://dx.doi.org/10.1007/3-540-45949-9.

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, 2022. URL https://arxiv.org/abs/2204.07112.

Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, pages 209–228, New York, NY, 1990. Springer-Verlag. ISBN 978-0-387-34808-7.

Robert Rand, Jennifer Paykin, and Steve Zdancewic. Phantom Types for Quantum Programs, January 2018. URL https://popl18.sigplan.org/event/coqpl-2018-phantom-types-for-quantum-programs. Talk at The Fourth International Workshop on Coq for Programming Languages (CoqPL '18).

Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Adapting proof automation to adapt proofs. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 115–129, 2018.

Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. Qed at large: A survey of engineering of formally verified software. *Foundations and Trends® in Programming Languages*, 5(2–3):102–281, 2019. ISSN 2325-1131. doi:10.1561/2500000045. URL http://dx.doi.org/10.1561/2500000045.

Marcus Rossel. An equality saturation tactic for lean. Master's thesis, TU Dresden, 2024.

Pierre Roux, Cyril Cohen, Enrico Tassi, Kazuhiko Sakaguchi, affeldt-aist, Laurent Théry, Erik Martin-Dorel, Georges Gonthier, Assia Mahboubi, Christian Doczkal, Florent Hivert, Yves Bertot, Maxime Dénès, Anton Trunov, jouvelot, Emilio Jesús Gallego Arias, Pierre-Marie Pédrot, Evgeniy Moiseenko, Jasper Hugunin, Matej Košík, Pierre-Yves Strub, Alessandro Bruni, antonio nikishaev, KimayaBedarkar, YOSHIHIRO Imai, Pierre PomCo, YAMAMOTO Mitsuharu, Takafumi Saikawa, Julin S, and Jason Gross. math-comp/math-comp. https://github.com/math-comp/math-comp, 2008. URL https://github.com/math-comp/math-comp.

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, William Spencer, Laura Zielinski, Ben Caldwell, Adrian Lehmann, and Robert Rand. Vicar: Visualizing categories with automated rewriting in coq, 2024. URL https://arxiv.org/abs/2404.08163.

Razin Shaikh, Boldizsár Poór, David Yonge-Mallo, Mark Koch, Richie Yeung, Toumas Laakkonen, Kazuki Tsuoka, Lia Yeh, John van de Wetering, and Aleks Kissinger. GitHub - zxcalc/zxlive: A graphical tool for the ZX calculus — github.com. https://github.com/zxcalc/zxlive, 2023.

Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017. URL https://arxiv.org/abs/1712.01815.

Carlos Simpson. Homotopy types of strict 3-groupoids, 1998. URL https://arxiv.org/abs/math/9810059.

Gleb Solovev, Andrei Kozyrev, Nikita Khramov, and Anton Podkopaev. JetBrains Coq Pilot, 3 2023. URL https://coqpilot.net/.

Peiyang Song, Kaiyu Yang, and Anima Anandkumar. Towards large language models as copilots for theorem proving in lean, 2024. URL https://arxiv.org/abs/2404.12534.

Gemini Team, Rohan Anil, Sebastian Borgeaud, and Jean-Baptiste Alayrac et al. Gemini: A family of highly capable multimodal models, 2024. URL https://arxiv.org/abs/2312.11805.

The Coq Development Team. The Coq reference manual – release 8.18.0. https://coq.inria.fr/doc/V8.18.0/refman, 2023.

John van de Wetering. Zx-calculus for the working quantum computer scientist, 2020. URL https://arxiv.org/abs/2012.13966.

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, January 2021. ISSN 2475-1421. doi:10.1145/3434304. URL http://dx.doi.org/10.1145/3434304.

Georg Zehfuss. Ueber eine gewisse determinante. *Zeitschrift für Mathematik und Physik*, 3:298–301, 1858.

Yihong Zhang and Oliver Flatt. Ensuring the termination of equality saturation for terminating term rewriting systems. *Preprint*, 2023. URL https://pldi23.sigplan.org/details/egraphs-2023-papers/9/Ensuring-the-termination-of-equality-saturation-for-terminating-term-rewriting-system.

Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. Better together: Unifying datalog and equality saturation. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023. doi:10.1145/3591239. URL https://doi.org/10.1145/3591239.

Li Zhou, Gilles Barthe, Pierre-Yves Strub, Junyi Liu, and Mingsheng Ying. Coqq: Foundational verification of quantum programs. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571222. URL https://doi.org/10.1145/3571222.