

# Automated Proof Generation for Associative and Distributive Rewriting with E-Graphs

ADRIAN LEHMANN\*, University of Chicago, USA

BEN CALDWELL\*, University of Chicago, USA

JOHN REPPY, University of Chicago, USA

ROBERT RAND, University of Chicago, USA

**Abstract.** We present a strategy for encoding a dependently-typed inductive language originally designed for a proof assistant within e-graphs. This language necessitates automated reasoning about distributivity and associativity. We encode our domain-specific language into `egglog`. Since `egglog` currently lacks proof extraction, we discuss strategies for building proof trees within `egglog` and approaches to implementing proof extraction in `Metatheory.jl`. Once extraction exists, we plan on interfacing with `Coq` to automate proof generation and checking. Having such a tool would drastically reduce the overhead in using our `Coq` library and enable reasoning about distributive and associative structures more broadly.

## 1 MOTIVATION

Proof assistants are necessarily pedantic. Every proof detail must be elaborated. This can lead to unnecessary overhead even when proving simple statements. For example, to show  $x + (y - x) = y$  (for  $x, y \in \mathbb{Z}$ ) we cannot immediately cancel  $x$  and  $-x$ . Instead, we must first commute  $y$  and  $-x$ , then reassociate the term to  $(x - x) + y = y$ . There are more complicated problems of the same form and there has been work on automatically handling specifically the associative and commutative cases, allowing one to immediately cancel the terms  $x$  and  $-x$  [Braibant and Pous 2011]. This automation is powerful, but it is restricted to dealing only with associativity and commutativity. This means that even slightly more complicated structural relations, such as distributivity, cannot be handled. Going beyond distributivity, we would like to be able to handle additional structural equivalences that are relevant to grammars of our construction.

This is the issue we faced during the development of `VyZX` [Lehmann et al. 2023].<sup>1</sup> `VyZX` uses an inductive datatype to describe `ZX`-diagrams [Coecke and Duncan 2008] in the `Coq` proof assistant [Coq Development Team 2012]; `ZX`-diagrams are string diagrams [Selinger 2010] for describing quantum computation. We briefly present the structure of `VyZX` diagrams (Figure 1) to illuminate this problem. `VyZX` encodes diagrams as an inductive datatype that explicitly encodes its dimensions at the type level: The type `ZX` has two natural number arguments, where the first and second represent the number of inputs and outputs, respectively. Composite constructors, `Stack` and `Compose`, induce structure that is irrelevant to the diagrammatic interpretation but necessary to express terms within `Coq` and to give those terms semantic interpretations. `Stack` and `Compose` together form the structural level of `VyZX`, they tell us how parts of a diagram are positioned relative to one another. When writing diagrams, we use notations to make the structure easier to understand. We use  $zx_0 \Downarrow zx_1$  for `Stack`  $zx_0\ zx_1$  and  $zx_0 \leftrightarrow zx_1$  for `Compose`  $zx_0\ zx_1$ . `Stack` and `compose` interact in a way that means prior work on rewriting modulo associativity and commutativity fails, as this work focuses on simplifying with a single operator. One equation which highlights this is called `stack-compose distribution`. It tells us that  $(zx_0 \Downarrow zx_1) \leftrightarrow (zx_2 \Downarrow zx_3) \propto (zx_0 \leftrightarrow zx_2) \Downarrow (zx_1 \leftrightarrow zx_3)$ , given  $\text{output}(zx_0) = \text{input}(zx_2)$  and  $\text{output}(zx_1) = \text{input}(zx_3)$ , where  $\propto$  (proportional to) is our equivalence relation on diagrams.

\* Equal contribution

<sup>1</sup>GitHub: <http://github.com/inQWIRE/VyZX/>

$$\begin{array}{c}
\frac{\text{in out} : \mathbb{N} \quad \alpha : \mathbb{R}}{\text{Z in out } \alpha : \text{ZX in out}} \quad \frac{}{\text{Cap} : \text{ZX } 0 \ 2} \quad \frac{}{\text{Cup} : \text{ZX } 2 \ 0} \quad \frac{\text{in out} : \mathbb{N} \quad \alpha : \mathbb{R}}{\text{X in out } \alpha : \text{ZX in out}} \\
\frac{}{\text{Wire} : \text{ZX } 1 \ 1} \quad \frac{}{\text{Box} : \text{ZX } 1 \ 1} \quad \frac{}{\text{Swap} : \text{ZX } 2 \ 2} \quad \frac{}{\text{Empty} : \text{ZX } 0 \ 0} \\
\frac{\text{zx}_0 : \text{ZX in mid} \quad \text{zx}_1 : \text{ZX mid out}}{\text{Compose } \text{zx}_0 \ \text{zx}_1 : \text{ZX in out}} \quad \frac{\text{zx}_0 : \text{ZX in}_0 \ \text{out}_0 \quad \text{zx}_1 : \text{ZX in}_1 \ \text{out}_1}{\text{Stack } \text{zx}_0 \ \text{zx}_1 : \text{ZX (in}_0 + \text{in}_1) \ (\text{out}_0 + \text{out}_1)}
\end{array}$$

Fig. 1. The inductive constructors for block representation ZX-diagrams

Our goal in this paper is build an automatic rewrite system that can handle most, if not all, of the diagram structure rewriting required for  $\text{VyZX}$  proofs. Coq’s autorewrite tactic greedily rewrites terms using a provided set of equivalences, but this tactic is slow and prone to looping. We can, however, encode both the operators and the conditions for rewriting using e-graphs in a straightforward way. This makes e-graphs a more scalable solution to solve structural rewrite problems. We believe that by using e-graphs, we can accomplish the goal of having a more general form of rewriting modulo associativity and distributivity. We aim to automatically build proof terms for such problems for Coq to check.

To measure how this problem could improve our code, we took basic measurements of rewrites we consider to be purely structural, ones that deal with associativity, commutativity, casting, and other simplification. We found that 24.72% of all tactic applications within  $\text{VyZX}$  proofs are to manipulate associativity. Of those, 26% (i.e., 6.55% of overall statements) are repeated applications. Given that many  $\text{VyZX}$  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.

## 2 IMPLEMENTATION & CHALLENGES

We aim to implement an automatic rewrite system. The goal is that given two valid  $\text{VyZX}$ -diagrams (i.e., one respecting dependent type constraints at compositions), the system outputs a proof path between the diagrams, if such a path exists. We then have Coq check the resulting proof. To accomplish this we need to represent the datatype from [Figure 1](#) in an e-graph solver, such as egg [Willsey et al. 2021], egglog [Zhang et al. 2023], or Metatheory.jl [Cheli 2021]. To represent our datatype, we need to represent both the structure of the string diagram and its dimensions, which are given by natural numbers. In  $\text{VyZX}$  diagrams, often dimensional arguments will be given as variables or equations over variables. Coq natural numbers have simplification rules that we need to encode within the e-graph such as constant folding and left identity elimination. To reason about diagrams with their associated dimensions, we choose to use egglog, which supports multiple types natively. Systems that do not accommodate multiple types natively, such as egg, could represent our types by merging them into a single large type and repeatedly performing checks, but this would be impractical.

In egglog, we build the  $\text{Dim}$  type shown in [Figure 2](#) (corresponding to  $\text{VyZX}$ ’s dimensions) and  $\text{ZX}$  shown in [Figure 1](#) type with the dependent type information removed. In the egglog  $\text{ZX}$  type there are some additional “constructors” such as `cast`, `nWire`, and `nStack1`. These are commonly used

$$\frac{x : \text{str}}{\text{Var } x : \text{Dim}} \quad \frac{n > 0 : \text{int}}{\text{Const } n : \text{Dim}} \quad \frac{n_1 : \text{Dim} \quad n_2 : \text{Dim}}{\{\text{Plus}, \text{Minus}, \text{Mult}\} n_1 \ n_2 : \text{Dim}}$$

Fig. 2. The constructors for  $\text{Dim}$  type

functions on  $VyZX$  diagrams that we would like our rewriting system to handle. In our encoding there is no functional difference between a constructor and a function that produces a  $ZX$  diagram. We show this construction in Listing 1. To then implement dependent types, we create functions that map any given object of type  $ZX$  to its dependent arguments<sup>2</sup>. We then have to force `egglog` to generate e-graph nodes for all dependent types so we can operate on them. Listing 2 shows parts of the `egglog` code implementing dependent types.

The next step is to translate  $VyZX$  lemmas into `egglog`. Given we are working under the promise that any input diagram is valid, we need to ensure that any translation maintains that invariant. For many rules this is immediate, while for others this means checking pre-conditions. We will show examples of both. When showing these examples we will be using visualizations of  $VyZX$  diagrams. In this visualization, squares are variables, and dotted horizontal and vertical rectangles represent `Stack` and `Compose`, respectively. For more details please see our 2023 paper. Let's look at the lemma `compose_assoc` (visualized in Figure 3):

```
Lemma compose_assoc : ∀ {in mid0 mid1 out : ℕ}
  (zx1 : ZX in mid0) (zx2 : ZX mid0 mid1) (zx3 : ZX mid1 out),
  (zx1 ↔ zx2) ↔ zx3 ∝ zx1 ↔ (zx2 ↔ zx3).
```

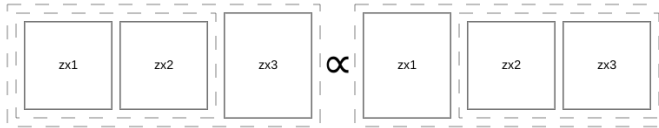


Fig. 3. Visualization of the `compose_assoc` lemma

In this example, we do not alter connection information at any composition point. Hence, given a valid  $VyZX$  diagram, rewriting using this rule will always yield a valid diagram. This means we do not need any additional checks and can simply encode the rule to search for the  $ZX$  substructure in our e-graph, as shown in Listing 3.

By contrast, when we look at `stack_compose_distr` (visualized in Figure 4), as discussed earlier, we need to take care of dependent type information. In the type parameters of the lemma, we see that we need to enforce that the upper and lower diagrams form valid composition points individually (i.e., their inputs and outputs match):

```
Lemma stack_compose_distr : ∀ {in1 mid1 out1 in2 mid2 out2 : ℕ}
  (zx1 : ZX in1 mid1) (zx2 : ZX mid1 out1) (zx3 : ZX in2 mid2) (zx4 : ZX mid2 out2),
  (zx1 ↔ zx2) ↓ (zx3 ↔ zx4) ∝ (zx1 ↓ zx3) ↔ (zx2 ↓ zx4).
```

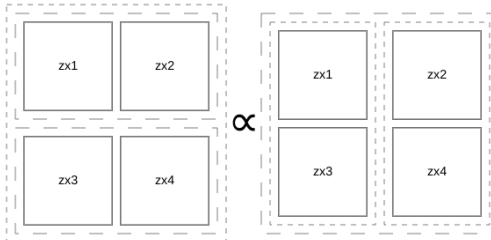


Fig. 4. Visualization of the `stack_compose_distr` lemma

<sup>2</sup>We build on the idea found in the [egglog matrix example](#) on GitHub.

```

148 (sort Dim)
149 (function Dim_lit (i64) Dim)
150 (function Dim_var (String) Dim)
151 (function Dim__add__ (Dim Dim) Dim)
152 (function Dim__mul__ (Dim Dim) Dim)
153 (sort ZX)
154 (function ZX_symbol (String Dim Dim) ZX)
155 (function ZX_cast (ZX Dim Dim) ZX)
156 (function ZX_compose (ZX ZX) ZX)
157 (function ZX_stack (ZX ZX) ZX)
158 (function ZX_Z (Dim Dim Dim) ZX)
159 (function ZX_X (Dim Dim Dim) ZX)
160 (function ZX_nStack1 (ZX Dim) ZX)
161 (function ZX_nWire (Dim) ZX)

```

Listing 1. Defining our type in egglog

```

165 (function ZX_n (ZX) Dim)
166 (function ZX_m (ZX) Dim)
167 ; Force nodes to be created
168 (rule ((= x (ZX_cast a n m))) ((ZX_n x) (ZX_m x)))
169 (rule ((= x (ZX_compose a b))) ((ZX_n x) (ZX_m x)))
170 ; Actual rules to define "dependent types"
171 (rewrite (ZX_n (ZX_cast a n m)) n)
172 (rewrite (ZX_m (ZX_cast a n m)) m)
173 (rewrite (ZX_n (ZX_compose a b)) (ZX_n a) :when ((= (ZX_m a) (ZX_n b))))
174 (rewrite (ZX_m (ZX_compose a b)) (ZX_m b) :when ((= (ZX_m a) (ZX_n b))))
175 ; Rest omitted for brevity

```

Listing 2. Defining dependent types

```

180 (rewrite (ZX_compose zx0 (ZX_compose zx1 zx2)) (ZX_compose (ZX_compose zx0 zx1) zx2))

```

Listing 3. compose\_assoc translated into egglog

```

184 (rewrite (ZX_compose (ZX_stack zx0 zx1) (ZX_stack zx2 zx3))
185         (ZX_stack (ZX_compose zx0 zx2) (ZX_compose zx1 zx3))
186         :when ((= (ZX_m zx0) (ZX_n zx2)) (= (ZX_n zx3) (ZX_m zx1))))

```

Listing 4. stack\_compose\_distr translated into egglog

Given we have the input and output information in the e-graph we can simply add a condition to our rewrite rule, as shown in Listing 4. Here we see the elegance of having multiple types, as this is a very natural encoding.<sup>3</sup>

<sup>3</sup>To see more rules encoded, please see <https://github.com/inQWIRE/vyzz-acdc/blob/main/zx.egg> or <https://github.com/inQWIRE/vyzz-acdc/blob/main/eggvyzz.ipynb> with `proof_tree` set to `False`.

## 2.1 Proof extraction

Once we have our rewriting system we need to extract proofs from egglog to Coq. Unfortunately, unlike egg, egglog does not have a proof extraction system [Flatt et al. 2022; Nieuwenhuis and Oliveras 2005]. This is a planned feature for egglog that should benefit proof assistants broadly, but its current absence is an obstacle to our work. To circumvent this issue, we took inspiration from the pathproof example in egglog,<sup>4</sup> which constructs path objects that carry their proofs. By expanding their example to work for Stack and Compose we gain the ability to generate proof paths between two equivalent diagrams. This necessarily creates up to 3 unique proof paths per binary operator, creating an exponential blowup. We show the resulting construction of proof paths in Figure 5. The exponential nature of this construction costs us many of the performance advantages created by an e-graph based system, as we construct all possible proofs. Therefore, this workaround will not scale beyond small examples.<sup>5</sup>

$$\begin{array}{c}
 \frac{\text{edge}(a, b)}{\text{path}(a, b)} \quad \frac{\text{edge}(b, c)}{\text{path}(a, b)} \quad \frac{\text{path}(a, b) \quad \text{path}(c, d)}{\text{path}(a \oplus c, b \oplus d)} \quad \oplus \text{ is binary operation} \\
 \frac{\text{path}(a, b) \quad \oplus \text{ is binary operation}}{\text{path}(a \oplus c, b \oplus c)} \quad \frac{\text{path}(c, d) \quad \oplus \text{ is binary operation}}{\text{path}(a \oplus c, a \oplus d)}
 \end{array}$$

Fig. 5. Definition of proof paths

## 3 FUTURE DIRECTIONS

*Metatheory.jl.* Given the lack of proof extraction in egglog, we recently started exploring Metatheory.jl [Cheli 2021] as an alternative means of using e-graphs to reason about string diagrams. Metatheory.jl boasts a significant speed advantage over egglog and also allows for a more natural style of programming using the Julia programming language. Unfortunately, it also does not yet have proof extraction, as discussed in an open issue.<sup>6</sup> Given the importance of proof extraction, the clear next step for this line of work is to add extraction into Metatheory.jl.

*Translating proofs into Coq.* Once we have a proof extraction system producing a proof path, we can extract it into a Coq proof via a plugin, similar to the plugin presented in Kozyrev’s 2023 thesis.

*Fundamental research directions.* We see a few exciting directions for this line of work. One key feature of previous projects on rewriting modulo AC is that you can access their rewrite rules by implementing a Coq typeclass and thereby gaining access to modulo AC rewriting tactics. In our work abstracting string diagram operations and automation from VyZX to general monoidal categories [Shah et al. 2024], we have implemented many typeclasses covering different categorical definitions. We could apply a similar approach by implementing e-graph based automation for typeclasses, which can then be accessed without having to interact with e-graphs directly.

We have three types of automation that are of interest. The first is rewriting modulo some equivalences inspired by prior work on rewriting modulo AC. The second is simplification of terms by using rules which shrink the size of our diagrams to simplify proofs. Finally, we would like to

<sup>4</sup>See <https://egraphs-good.github.io/egglog/?example=pathproof>

<sup>5</sup>See this system of proof construction using <https://github.com/inQWIRE/vyzz-acdc/blob/main/eggvyzz.ipynb> with proof\_tree set to True

<sup>6</sup><https://github.com/JuliaSymbolics/Metatheory.jl/issues/111>

implement hammer-style tactics [Czajka and Kaliszyk 2018] which attempt to automatically solve goals. We believe these types of automation could all be implemented easily using e-graphs.

In line with adding custom tactics using Coq, it would be of great value if Coq lemmas could be turned into e-graph rewrite rules automatically. We want to extend our work to automatically solve proofs by using lemmas specified within a Coq hint database (a user defined collection of lemmas for use with automation), and extracting those to e-graph-based solvers.

## 4 CONCLUSION

We present a system to automatically rewrite a dependently typed domain specific language within Coq. While it still has many challenges to overcome, we have shown how we can encode a dependently typed system and rewrite rules built for such a system. We believe that proof automation is critical to formal verification, and specifically for reasoning modulo associativity and distributivity. We see e-graphs as a promising path towards large scale verification of diagrammatic languages, and structurally complex datatypes in general.

## ACKNOWLEDGEMENTS

We thank Saul Shanabrook, Alessandro Cheli and Max Willsey for their assistance with e-graph related issues and for welcoming us to the [e-graphs Zulip channel](#).

This material is based upon work supported by the Air Force Office of Scientific Research under awards number FA95502310361 and FA95502310406.

## REFERENCES

- Thomas Braibant and Damien Pous. 2011. Tactics for reasoning modulo AC in Coq. In *International Conference on Certified Programs and Proofs*. Springer, 167–182.
- Alessandro Cheli. 2021. Metatheory.jl: Fast and Elegant Algebraic Computation in Julia with Extensible Equality Saturation. *Journal of Open Source Software* 6, 59 (2021), 3078. <https://doi.org/10.21105/joss.03078>
- Bob Coecke and Ross Duncan. 2008. Interacting Quantum Observables. In *Automata, Languages and Programming*, Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfssdóttir, and Igor Walukiewicz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 298–310. [https://doi.org/10.1007/978-3-540-70583-3\\_25](https://doi.org/10.1007/978-3-540-70583-3_25)
- The Coq Development Team. 2012. The Coq Reference Manual, version 8.4. Available electronically at <http://coq.inria.fr/doc>.
- Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for dependent type theory. *Journal of automated reasoning* 61 (2018), 423–453.
- Oliver Flatt, Samuel Coward, Max Willsey, Zachary Tatlock, and Pavel Panchekha. 2022. Small Proofs from Congruence Closure. arXiv:2209.03398 [cs.PL]
- Andrei Kozyrev. 2023. *Equality saturation for solving equalities of relational expressions*. Bachelor’s thesis. Constructor University Bremen.
- Adrian Lehmann, Ben Caldwell, Bhakti Shah, and Robert Rand. 2023. VyZX: Formal Verification of a Graphical Quantum Language. arXiv:2311.11571 [cs.PL]
- Robert Nieuwenhuis and Albert Oliveras. 2005. Proof-producing congruence closure. In *International Conference on Rewriting Techniques and Applications*. Springer, 453–468.
- P. Selinger. 2010. A Survey of Graphical Languages for Monoidal Categories. In *New Structures for Physics*. Springer Berlin Heidelberg, 289–355. [https://doi.org/10.1007/978-3-642-12821-9\\_4](https://doi.org/10.1007/978-3-642-12821-9_4)
- Bhakti Shah, William Spencer, Laura Zielinski, Ben Caldwell, Adrian Lehmann, and Robert Rand. 2024. ViCAR: Visualizing Categories with Automated Rewriting in Coq. <https://rand.cs.uchicago.edu/publication/shah-2024-vicar/> Preprint.
- Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29.
- Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. arXiv:2304.04332 [cs.PL]