Graphical Rewriting for Diagrammatic Reasoning in Monoidal Categories in Lean4

Sam Ezeh
Durham University, UK

Abstract

We present Untangle, a Lean4 extension for Visual Studio Code that displays string diagrams for morphisms inside monoidal categories, allowing users to rewrite expressions by clicking on natural transformations and morphisms in the string diagram. When the the user manipulates the string diagram by clicking on natural transformations in the Graphical User Interface, it attempts to generate relevant tactics to apply which it then inserts into the editor, allowing the user to prove equalities visually by diagram rewriting.

2012 ACM Subject Classification Theory of computation → Interactive proof systems; Human-centered computing → Graphical user interfaces

Keywords and phrases Interactive theorem proving, Lean4, Graphical User Interface

Digital Object Identifier 10.4230/LIPIcs.ITP.2024.41

Category Short Paper

Supplementary Material *Software (Source Code)*: `https://github.com/dignissimus/Untangle` [7]

1 Introduction

String diagrams are a visual language for reasoning about morphisms in monoidal categories where rewriting rules for expressions using equalities becomes a series of visual transformations on a diagram that appear in a variety of mathematical contexts, both pure and applied. In pure settings string diagrams appear when reasoning about objects whose structure can be modelled by symmetric monoidal categories such as Hopf algebras and braided monoidal categories [1] and in applied settings, string diagrams appear as tools for reasoning about resource-sensitive systems in fields such as quantum mechanics and circuit theory [3]. String diagrams also simplify the exposition of ideas otherwise obscured by terse written notation. In this paper, we contribute a Lean4 extension that renders morphisms and natural transformations inside a monoidal category as string diagrams using the ProofWidgets [13] framework for building Graphical User Interfaces as part of ongoing work.

2 Background and Related Work

2.1 Lean

Lean [12] is a dependently typed programming language and proof assistant based on the Calculus of Inductive Constructions. In Lean, users construct mathematical statements by building types and prove these statements by exhibiting terms that inhabit these types. Lean also provides a "tactic" language that allows users to prove statements via a series of commands called tactics that rewrite the current statement that the user intends to prove by using existing lemmas and hypotheses. Lean exposes the statement that the user would like to prove, referred to as the "goal", alongside the assumptions and hypotheses at hand which are referred to collectively as the "goal state" which can then be accessed and manipulated programmatically through the use of a rich, monadic meta-programming interface. This

extension integrates with the Lean4 proof assistant as an extension that generates tactics in response to user interaction with the rendered string diagram that it presents in the Graphical User Interface.

2.2 Penrose

Penrose [14] is a tool for declaratively constructing mathematical diagrams that allows developers to construct a DSL (Domain Specific Language) that describes the types of mathematical objects that the developer would like to display and the relations between them and a stylesheet to describe how the system should render these mathematical objects. After specifying a DSL, the developer can programmatically construct diagrams containing instances of these mathematical objects by generating scripts written using this DSL. Penrose will then generate images depicting these mathematical objects in the manner described by the user-provided stylesheet. Penrose excels for our use case as it does not rely on a fixed library of visualisation tools and the DSL allows the developer to define visual representations in its constraint-based specification language. We use Penrose to visually render string diagrams.
ProofWidgets [13] is a general-purpose framework for building Graphical User Interfaces in Lean, allowing developers to build visual "widgets" that interact with Lean. ProofWidgets provides an API that allows for effortless construction of interactive widgets and extensions for Lean4 by allowing the developer to create composable components and includes a built-in component that integrates the Penrose system for constructing mathematical diagrams. We build the extension using the ProofWidgets framework, allowing us to write user interface logic in JavaScript and relay information from the user interface to the Lean server and Visual Studio Code using RPCs (Remote Procedure Calls) and also write RPC handlers in Lean.

2.3 Graphical proof assistants

There are a handful of stand-alone visual proof assistants that allow the user to work by manipulating string diagrams, such as Globular [2], homotopy.io [5] and Quantomatic [9]. Graphical proof assistants represent terms as diagrams and allow users to manipulate diagrams according to rewrite rules that correspond to equalities between two terms. In these proof assistants, users interact with graphical proof assistants by clicking on diagram elements that represent mathematical objects in order to apply rewrite rules to them and the neighbouring elements. These proof assistants typically exist either as online web applications or downloadable executables as opposed to integrating with pre-existing proof assistants. In addition to these tools, there is an in-progress Pull Request to Lean's Mathlib for displaying string diagrams authored by Yuma Mizuno [11], although it does not aim to rewrite terms in the goal.

2.4 String diagrams

Commutative diagrams are a graphical representation depicting morphisms inside a category where, typically, objects are represented as points or vertices, morphisms are represented as arrows between these points and 2-morphisms are presented as arrows between arrows. Dualising this representation, one arrives at string diagrams where objects are represented as faces, morphisms as lines and 2-morphisms as points or vertices.

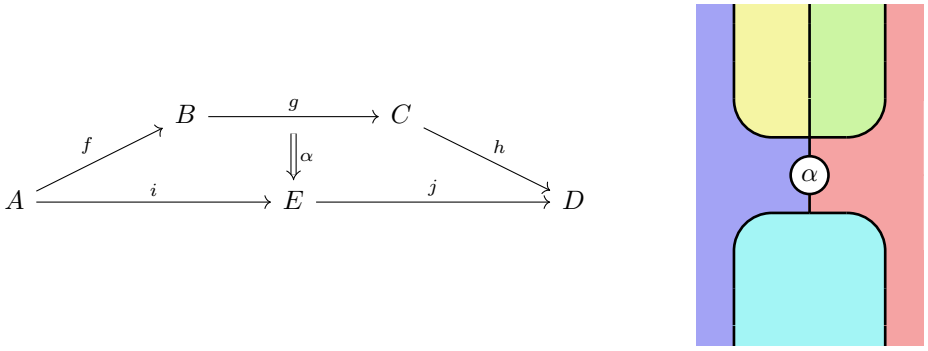

Figure 1 The corresponding commutative diagram and string diagram.

For example, let A, B, C, D and E be objects inside a category, let $f : A \to B, g : B \to C, h : C \to D, i : A \to E, j : E \to D$ be morphisms inside this category and let $\alpha : h \circ g \circ f \Rightarrow j \circ i$ be a 2-morphism between them. As a commutative diagram and as a string diagram, this would appear as in Figure 1 where every object is represented by a coloured planar face.

String diagrams allow for reasoning about morphism equalities with soundness guaranteed by coherence theorems [8] and provide a graphical representation for certain structures inherent to monoidal categories. For example, the associativity of morphism composition is inherent to the string diagram representation as the composition of morphisms in string diagrams is represented by positioning morphisms such that they are adjacent. More generally, string diagrams are for bicategories of which monoidal categories are a special case.

3 Proof of Concept

We implement "Untangle" as a proof of concept and use it to prove example statements about monads: monoid objects in the monoidal category of endofunctors and natural transformations between them. This work is accessible online at `https://github.com/dignissimus/Untangle`. When the current proof goal is an equality, untangle renders string diagrams for the morphisms on both sides of the equality as seen in Figure 2. The user may then prove the equality of the two morphisms by manipulating the diagram by clicking on morphisms on either side of the diagram. When the user has selected the section of the diagram that they would like to rewrite, the extension constructs a tactic to apply a relevant theorem to rewrite the goal statement in the sub-expressions that the diagram components represent. After the goal statement updates in the Lean info view, the Graphical User Interface re-renders the diagram so that it represents the updated goal statement.

3.1 Example workflow

If, for example, one were to prove the equality between $\mu_X \circ f \circ \mu_X$ and $\mu_X \circ T\mu_X \circ TTf$ for some monad T equipped with natural transformations $\mu : T^2 \to T$ and $\eta : \mathbf{1} \to T$ satisfying the monad laws and a morphism $f : X \to TX$ for some object X. We may begin by using the naturality of μ to argue that $\mu_X \circ f \circ \mu_X$ is equal to $\mu_X \circ \mu_{TX} \circ TTf$, we could then proceed to use the associativity law for the μ natural transformation to argue that this equals $\mu_X \circ T\mu_X \circ TTf$, completing the proof.

In Lean, this statement would appear as follows:

```
example [Category C] {T : Monad C} {X : C} {f : X → T.obj X}
  : T.μ.app X ≫ T.map f ≫ T.μ.app X
    = T.map (T.map f) ≫ T.map (T.μ.app _) ≫ T.μ.app _
  := by with_panel_widgets [Untangle] {

  }
```

When the user places the cursor inside the braces inside Visual Studio Code, the extension will render string diagrams for both of the morphisms on each side of the equality that appear inside the Lean infoview as in Figure 2. Visually, the proof of the theorem consists of pulling up the point that represents the morphism f as a simple planar deformation then using the monad associativity law to swap the order of the μ natural transformations.

With the extension, clicking on the morphism f and the natural transformation μ generates the following tactic and enters it into the lean editor:

```
conv => {
  enter [1];
  slice 1 2;
  rw [
    ← (Monad.μ T).naturality (f),
     CategoryTheory.Functor.comp_map
  ];
};
try simp only [CategoryTheory.Category.assoc];
```

This tactic does the following:

- Selects the left-hand side of the expression
- Uses the indices 1 and 2 to select a "slice" containing the subexpression to re-write
- Applies the naturality condition for the μ natural transformation to rewrite the goal statement
- Rewrites $(T \circ T)f$ as TTf
- Attempts to simplify the expression using the associativity of morphism composition.

This updates the goal state and the extension renders a new diagram that has the order of f and μ swapped.

After this, we can click on the two μ morphisms to apply the associativity law which inserts the following tactic into the editor:

```
  conv => {
    enter [1];
    slice 2 3;
    rw [← Monad.assoc];
  };
  try simp only [CategoryTheory.Category.assoc];
```

The equality has now been proved and the Lean type-checker accepts the statement and proof.

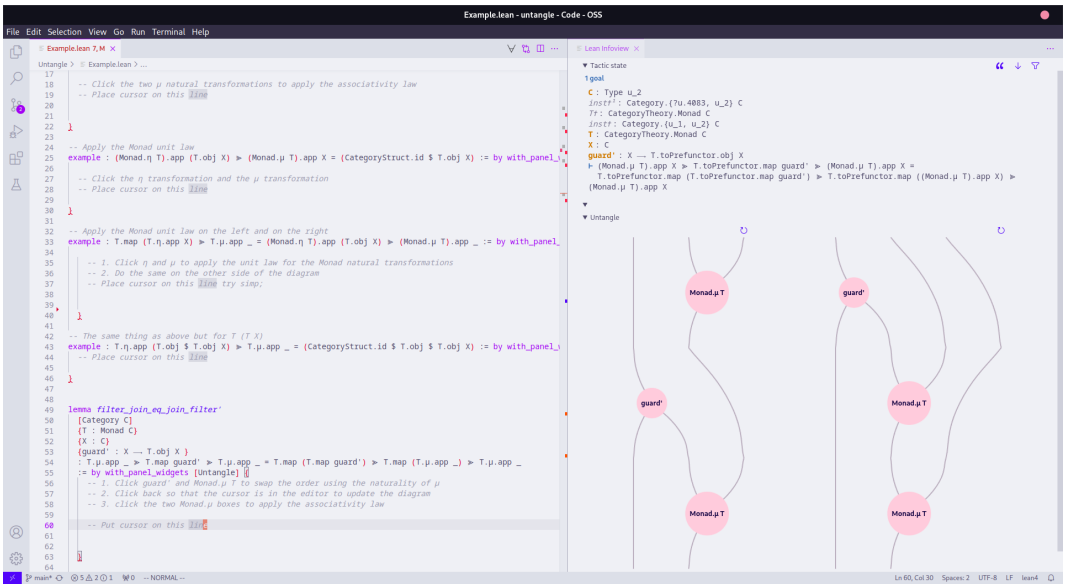

Figure 2 Using the extension to prove morphism equality.

4 Implementation

4.1 Data structures

In a similar manner to Comfort et al. [4], we represent diagrams as a series of diagram components without association information where each diagram component is represented as the number of inputs into the natural transformation, its number of outputs and its location or "offset" in its level in the string diagram referring to the number of wires or strings that exist to the left of the component at its level. See Delpeuch and Vicary [6] for more information about this data structure.

4.2 Parsing morphism expressions

We parse morphisms, natural transformation components and objects from the goal statement into an internal representation that closely resembles the syntax tree for the Lean expressions that represent them by using a basic recursive-descent parser. While parsing morphisms, we use type information from Lean to tag expressions with semantic labels such as "functor", "morphism" or "natural transformation component". We use this semantic information to infer which tactics to apply when the user interacts with the diagram in the user interface.

4.3 Displaying diagrams

To display string diagrams to the user, we parse expressions into diagram components and use our representation of diagram components to construct declarative statements in the extension's Penrose DSL which Penrose uses to render a diagram according to the specification in the extension's stylesheet. We then construct a Penrose component using the ProofWidget framework and include it in the extension's section of the info view.

4.4 Handling user input

We handle user input by writing JavaScript to listen to click events in Visual Studio Code. After the user has interacted with the diagram, we send an RPC that contains information describing the interaction. This includes the location of the the elements that the user clicked on in the diagram and information about the state of the editor such as the position of the mouse cursor.

4.5 Pattern matching and tactic selection

We implement a simple rule-based system for deciding which tactic to select by using a combination of syntactic information from Lean's representation of the expression and the inferred semantic tags. For example, if the user selects two μ natural transformations we infer that the user would like to transform the diagram by swapping their order and rewrite the expression using the μ associativity law: $\mu_X \circ T\mu_X = \mu_X \circ \mu_{TX}$. We identify the direction in which we need to perform the rewrite by looking at the syntactic structure of the expression. For the example of the μ associativity law, we determine the direction of the rewrite by comparing the functor lifts of each of the two instances of the μ natural transformation in the diagram.

4.6 Rewriting up to associativity

A graphical proof interface with Lean faces a unique challenge with rewriting and associativity. In string diagrams, two expressions that differ only by association are represented identically however, in Lean, two expressions that differ only by their associations are not necessarily equivalent and we must associate theorems in the correct way before we may apply them to expressions. For example, if the context contains a lemma that states $f \circ g = f'$ we may reduce $(f \circ g) \circ h$ to $f' \circ h$ but we may not immediately reduce $f \circ (g \circ h)$ without re-associating the expression. This poses problems as to translate the user's theorem into a tactic to rewrite the goal statement as we must get the association right in terms of both the hypotheses and the resulting goal state. We tackle this by making use of Lean's "`conv`" tactic and in particular, its "`slice`" command. When parsing the composition of morphisms in the equalities, we calculate the "location" of each morphism in the expression. We then use this as input to Lean's `slice` command which allows us to specify indices to select subexpressions in the goal statement and re-associate them into a normal form by repeatedly applying the `Category.assoc` lemma which asserts the equality of $f \circ (g \circ h)$ and $(f \circ g) \circ h$

4.7 Extensibility

From a Software Engineering perspective, we achieve extensibility in the extension by defining an abstract `GraphicalLanguage` structure which developers can implement. A developer defines graphical languages for new structures by implementing a function that parses Lean expressions and produces a representation of the expression's structure, which the extension uses to render the diagram. Additionally, the developer defines a function to handle click events, producing a list of tactics to insert into the editor, and implementing functions that determine the cosmetic styling of elements in the diagram.

4.8 Entering tactics into the editor

After the RPC handler has received the message describing the user's interaction with the diagram and has constructed the tactic that is to be entered into the user interface, we enter tactics into the Visual Studio Code editor by embedding the generated tactic and information about where in the document we should insert it in the response to the RPC which we then handle in the JavaScript code. The JavaScript then uses an "EditorContext" to relay this information to Visual Studio code.

5 Conclusion and Future Work

In conclusion, we have created a Lean4 extension capable of rendering morphisms inside a monoidal category as string diagrams to assist users in proving equalities between morphisms in monoidal categories. Untangle is still in the early stages of development and, in the future, we seek to implement rewrite rules for other monoidal structures such as comonoids, bimonoids and Hopf monoids. In addition to supporting more monoidal structures we want to achieve the following:

- At present, the user interface is simple: rewrite rules are applied by clicking on two morphisms and natural transformations in the diagram. This simplicity limits the type of rewrite rules that the extension can support while remaining intuitive. In the future, we want to extend the user interface by implementing a context menu for a more explicit interface over what tactic the extension selects.
- There are simple graphical rules for checking whether certain rewrite rules or transformations can be applied to a diagram and while the Lean4 type-checker will refuse erroneous applications of rewrite rules and output an error message, using these simple diagrammatic checks to alert the user about incorrect rewrites within the graphical interface would provide the user with a better experience.
- While existing Lean tactics continue to work as normal with the assumptions and hypotheses in scope, the extension doesn't provide a way to apply these lemmas graphically. Allowing the user to graphically apply lemmas from the goal state to the diagram via the user interface would be a great addition to the user experience.
- Currently, all rewrite rules are pre-declared for mathematical objects that exhibit monoidal structure. For example, the extension can render diagrams for classes of expressions in the language of HopfAlgebra and Monads and we have written rewrite rules for the monoidal structure of Monads and this would be repeated for new mathematical objects with monoidal structure. While this process is not tedious, it would be much better if there were a more general way of building rewrite rules for mathematical objects with monoidal structures.
- The extension lacks parsing for certain types of expressions in the statement. While we currently parse TTf and $Tf \circ Tg$ into the internal representation of the syntax tree, we do not currently parse $(T \circ T)f$ or $T(f \circ g)$. In the future, we want to parse a wider variety of expressions and provide more comprehensive coverage over the types of expressions the user can work with diagrammatically.

References

1 John Baez and Mike Stay. *Physics, topology, logic and computation: a Rosetta Stone*. Springer, 2011.

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

3 Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. String diagram rewrite theory ii: Rewriting with symmetric monoidal structure. *Mathematical Structures in Computer Science*, 32(4):511–541, 2022.

4 Cole Comfort, Antonin Delpeuch, and Jules Hedges. Sheet diagrams for bimonoidal categories. *arXiv preprint arXiv:2010.13361*, 2020.

5 Nathan Corbyn, Lukas Heidemann, Nick Hu, Chiara Sarti, Calin Tataru, and Jamie Vicary. homotopy.io: a proof assistant for finitely-presented globular n-categories. *arXiv preprint arXiv:2402.13179*, 2024.

6 Antonin Delpeuch and Jamie Vicary. Normalization for planar string diagrams and a quadratic equivalence algorithm. *Logical Methods in Computer Science*, 18, 2022.

7 Sam Ezeh. Dignissimus/untangle: Graphical rewriting for diagrammatic reasoning in monoidal categories in lean4, 2024. Software (visited on 2024-08-19). URL: `https://github.com/dignissimus/Untangle`.

8 André Joyal and Ross Street. The geometry of tensor calculus, i. *Advances in Mathematics*, 88(1):55–112, 1991. `doi:10.1016/0001-8708(91)90003-P`.

9 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.

10 Daniel Marsden. Category theory using string diagrams. *arXiv preprint arXiv:1401.7220*, 2014.

11 Yuma Mizuno. Feat: String diagram widget by yuma-mizuno (pull request #10581) leanprover-community/mathlib4, February 2024. URL: `https://github.com/leanprover-community/mathlib4/pull/10581`.

12 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.

13 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. `doi:10.4230/LIPIcs.ITP.2023.24`.

14 Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Jonathan Sunshine, and Keenan Crane. Penrose: From mathematical notation to beautiful diagrams. *ACM Trans. Graph.*, 39(4), 2020.