String Diagrams for Monoidal Categories, in Rocq

Damien Pous
Plume team, CNRS, LIP, ENS de Lyon, UMR 5668, France

Abstract

We present a Rocq library for monoidal categories, including a decision procedure for proving equality of morphisms as well as notations that make it possible to reason as if these monoidal categories were strict, inferring MacLane isomorphims automatically in the background. Together with an external tool for visualising and editing string diagrams, this make it possible to perform rewriting steps graphically, and to translate them into textual formal proofs which are concise and readable.

2012 ACM Subject Classification Theory of computation → Logic

Keywords and phrases Monoidal categories, string diagrams, formal proofs, graphical proofs, Rocq

Digital Object Identifier 10.4230/LIPIcs.ITP.2026.28

Related Version *Author Version*: `https://hal.science/hal-05522044`

Supplementary Material *Software*: `https://perso.ens-lyon.fr/damien.pous/string-diagrams/`

Acknowledgements We are grateful to Ralph Sarkis for two seminars he gave in 2023 to the Plume team, where he advocated the use of (string) diagrammatic proofs and showed us the proofs for monoid composition we use as a running example in the present paper.

Declaration about GenAI/LLM use No AI has been used for this work: neither to write or proofread the paper, nor to write the accompanying code artifacts, nor to generate illustrations.

Introduction

Modern proof assistants made it possible to formalise impressive results in both mathematics and computer science. Nevertheless, besides situations where the considered proofs involve numerous computations which would hardly be done by a human-being (e.g., Kepler's conjecture or the four-colour theorem), proof assistants often make it harder to write convincing proofs: proofs that we can read and get inspiration from. This is particularly true in domains where scientists have developed notational systems that go beyond traditional one-dimensional text to convey their ideas.

One such domain is category theory: proofs in category theory are often best displayed as commutative diagrams, where we get to see that the whole diagram commutes because all of its inner components do. This notational system has its limitations: when the diagram of interest is not planar, one has to be ingenuous to draw it on paper in an intelligible way. Still, despite recent efforts to design prover interfaces to output proof states and input user instructions graphically [28, 13, 12], it remains difficult to formalise these proofs in such a way that the resulting object faithfully conveys the initial idea.

In this paper we focus on a specific concept of category theory: monoidal categories [9, 29, 23, 17, 10, 32]. Those are categories with an associative bifunctor, thus a fairly general concept, which was subsequently refined into many variations (braided, symmetric, Cartesian, closed). Still, monoidal categories alone already make it possible to study concrete concepts such as monads: "a monad is just a monoid in the monoidal category of endofunctors". Moreover, monoidal categories come with a neat notation system: *string diagrams* [21, 33, 24, 34, 22, 40]. String diagrams are not commutative diagrams as mentioned above: they are pictures that make it possible to represent morphisms in monoidal categories, abstracting over irrelevant details induced by textual presentations of these morphisms. See [35] for a recent introduction.

Contributions. We present a Rocq library for monoidal categories, together with a graphical editor for string diagrams. Combined together, they make it possible to perform reasoning steps graphically, and to translate those graphical steps into clear and concise formal ones.

This is made possible by developing three automation tactics: a first one to infer "MacLane isomorphisms" automatically, so that the user does not need to write them explicitly; a second one to exploit the unicity of these morphisms (which is derived from the coherence conditions of monoidal categories); and a third one to prove general morphism equalities.

Outline. We first briefly recall standard categorical notation (Section 1) before discussing monoidal categories (Section 2) and string diagrams (Section 3). Then we present our diagram editor (Section 4), illustrating its usage on a simple example: the composition of two monoids. We finally describe the Rocq library (Section 5), which makes it possible to follow graphical steps and produce convincing formal proofs. We conclude with related works (Section 6) and directions for future work (Section 7).

1 Categorical notation

We assume some familiarity with the basic concepts of category theory: morphisms, isomorphisms, functors, and natural transformations [32, 7]. We use letters $A, B \dots$ to range over the objects of categories. Given two such objects, we write $f \colon A \rightsquigarrow B$ when f is a morphism from A to B, and $f \colon A \simeq B$ when f is an isomorphism from A to B. Given two morphisms $f \colon A \rightsquigarrow B$ and $f \colon B \rightsquigarrow C$, we write $f \mathbin{;} g$ for their composition, which is a morphism from A to C. We prefer this notation to the reversed one, $g \circ f$, as it makes the back and forth translations with diagrams more natural. In contexts where a morphism is expected, we use A to denote the identity morphism on an object A.

2 Monoidal categories

A *monoidal category* is a category $\mathcal{C}$ together with

- a bifunctor $\otimes \colon \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ called the *tensor*, whose application to two morphisms f, g is written $f \cdot g$;
- an object $1 \colon \mathcal{C}$ called the *unit*;
- and three natural families of isomorphisms

$$\alpha_{A,B,C} \colon (A \otimes B) \otimes C \simeq A \otimes (B \otimes C) \qquad \text{(associator)}$$
$$\lambda_A \colon 1 \otimes A \simeq A \qquad \text{(left unitor)}$$
$$\rho_A \colon A \otimes 1 \simeq A \qquad \text{(right unitor)}$$

- such that the following *coherence* diagrams commute:

$$\begin{array}{ccc} (A \otimes 1) \otimes B & \xrightarrow{\alpha_{A,1,B}} & A \otimes (1 \otimes B) \\ & {\scriptstyle \rho_A \cdot B} \searrow \quad \swarrow {\scriptstyle A \cdot \lambda_B} & \\ & A \otimes B & \end{array} \qquad \text{(triangle)}$$

$$\begin{array}{ccccc} ((A \otimes B) \otimes C) \otimes D & & \xrightarrow{\alpha_{A\otimes B,C,D}} & & (A \otimes B) \otimes (C \otimes D) \\ {\scriptstyle \alpha_{A,B,C} \otimes D} \downarrow & & & & \downarrow {\scriptstyle \alpha_{A,B,C\otimes D}} \\ (A \otimes (B \otimes C)) \otimes D & \xrightarrow[\alpha_{A,B\otimes C,D}]{} & A \otimes ((B \otimes C) \otimes D) & \xrightarrow[A \otimes \alpha_{B,C,D}]{} & A \otimes (B \otimes (C \otimes D)) \end{array} \qquad \text{(pentagon)}$$

Every category with finite products is monoidal, by taking the Cartesian product for the tensor and the terminal object for the unit. Endofunctors on a category form a monoidal category, where morphisms are natural transformations, tensor product is functor composition and unit is the identity functor. The latter monoidal category is *strict*, in the sense that the associators, left unitors, and right unitors consist of identity morphisms (in which case the two coherence diagrams hold trivially). The former is not strict in general: products are defined only up to isomorphisms.

2.1 Laws

Besides the last two coherence diagrams, whose nature is rather bureaucratic, the above definition implicitly provides some important laws in monoidal categories.

First, we have bifunctoriality of tensor product: for all morphisms f, g, f', g' of appropriate types, and all objects A, B, we have

$$(f \mathbin{;} g) \cdot (f' \mathbin{;} g') = f \cdot f' \mathbin{;} g \cdot g' \qquad\qquad A \cdot B = A \otimes B \ .$$

(Note that $(\cdot)$ binds stronger than $(;)$ in expressions, and that the second equation is about identity morphisms: $A \cdot B$ is the tensor product of the identity morphisms on A and B, and $A \otimes B$ is the identity morphism on that object.) As a consequence, for all morphisms $i \colon I \rightsquigarrow I'$ and $i \colon J \rightsquigarrow J'$ we have the following exchange law:

$$I \cdot j \mathbin{;} i \cdot J' = i \cdot j = i \cdot J \mathbin{;} I' \cdot j \ .$$

The above laws are used frequently, typically to bring together subexpressions which are far apart in some textual writing (e.g., i and j in the right-hand side of the last one), so that we can use some further knowledge about these subexpressions together. We shall see in the sequel that while the one dimensional textual writing of morphisms makes this tedious, string diagrams, which are two dimensional, make it possible to detect such potential rewritings visually and much more easily.

Second, we have naturality of the associators and unitors: for all morphisms $f \colon A \rightsquigarrow A'$, $g \colon B \rightsquigarrow B'$, and $h \colon C \rightsquigarrow C'$, we have

$$\begin{aligned} (f \cdot g) \cdot h &= \alpha_{A,B,C} \mathbin{;} f \cdot (g \cdot h) \mathbin{;} \alpha^{-1}_{A',B',C'} \\ 1 \cdot f &= \lambda_A \mathbin{;} f \mathbin{;} \lambda^{-1}_{A'} \\ f \cdot 1 &= \rho_A \mathbin{;} f \mathbin{;} \rho^{-1}_{A'} \ . \end{aligned}$$

2.2 Strict monoidal categories?

Strict monoidal categories are much easier to work with, as we do not have to care about associators and unitors. For instance, the last three laws just become

$$\begin{aligned} (f \cdot g) \cdot h &= f \cdot (g \cdot h) \\ 1 \cdot f &= f \\ f \cdot 1 &= f \ . \end{aligned}$$

On paper, when proving that a diagram holds, we may "pretend" that the considered monoidal category is strict, thanks to *strictification*: every monoidal category is (monoidally) equivalent to a strict one [32]. This technique is frequently used in the literature, when monoidal categories are not just assumed to be strict, "for the sake of simplicity".

Unfortunately, these two approaches are not helpful in a type-theory based formalisation. Indeed, strict monoidal categories are not easier than the general ones in such a setting. Imagine for instance that we replace the associator by a family of propositional equalities:

$$a \colon \forall A, B, C,\ (A \otimes B) \otimes C = A \otimes (B \otimes C)\ .$$

Given three morphisms f, g, h, we expect $(f \cdot g) \cdot h = f \cdot (g \cdot h)$ to hold in a strict monoidal category. However, this equation is not even well-typed: it relates morphisms whose sources and targets are not definitionally equal. They are propositionally equal, though, thanks to a, and we can use an explicit "casting" operation to state this equation:

$$(f \cdot g) \cdot h = \mathit{cast}\,(a\_\_\_)\,(a\_\_\_)\,(f \cdot (g \cdot h))\ .$$

Such an equation actually corresponds to naturality of α, and following this path, we reach very quickly the point where we need coherence conditions on cast operations that correspond to the coherence diagrams of non-strict monoidal categories. All in all, the induced overhead is equivalent to working with a general monoidal category in the first place.

Note however that some monoidal categories are inherently strict, even in type theory. For instance, assuming extensionality, the category of endofunctors is strict monoidal, definitionally (e.g., given three functors F, G, H, the compositions $(F \circ G) \circ H$ and $F \circ (G \circ H)$ are definitionally equal.) While this makes it easier to work in such concrete monoidal categories, our point is that formalising strict yet abstract monoidal categories in type theory would not be simpler than formalising general ones – even in a context with univalence, where isomorphisms would boil down to propositional equalities: we would still need to deal with explicit coherence conditions on transports [1] (i.e., the previous *cast* operations).

We formalise general monoidal categories in the present work, and one of our main contribution is to provide support to reason as if they were strict, making it possible to elude associators and unitors entirely.

2.3 Formalisation

We started from the Rocq formalisation of categories from [3]. This library implements $\mathcal{E}$-categories [42, 25], where each homset is equipped with a user-defined equivalence relation ($\equiv$), so that we do not need axioms such as functional extensionality or proof-irrelevance. It is structured using *Hierarchy Builder* [14], which provides a high-level language to define hierarchies of structures, implementing *packed classes* [19] in the background. Besides Hierarchy Builder, we heavily rely on *setoid rewriting* [38], as well as the underlying *typeclass* system [39]. At a few places, we also use *Elpi* [16, 41] to improve notations. The development is axiom-free.

We slightly extended the aforementioned library to encompass basic properties of natural isomorphisms and products of categories, so that we can easily obtain bifunctors. We also added some basic automation tactics, by rewriting and reflection, to develop subsequent proofs more easily.

The system of notations and coercions makes it possible to keep compact statements that follow closely the conventions we use in the present paper.

3 String diagrams

String diagrams are the right abstraction for morphism expressions in monoidal categories: they abstract away the notational clutter induced by terms. We will not need a formal definition of string diagrams, even in Rocq, so that we only provide an intuitive one here.

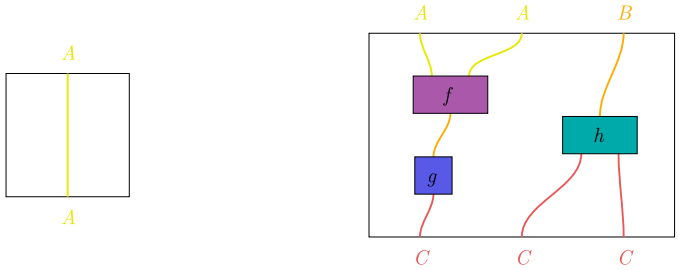


Figure 1 String diagrams corresponding to the identity on A and to the expression $(f \mathbin{;} g) \cdot h$ with $f \colon A \otimes A \rightsquigarrow B$, $g \colon B \rightsquigarrow C$ and $h \colon B \rightsquigarrow C \otimes C$.

A *string diagram* in a monoidal category is a graph, which consists of:

- a sequence of *outer input ports*, labelled by objects;
- a sequence of *outer output ports*, labelled by objects;
- a set of *nodes*, labelled by morphisms and placed in the plane;
- a set of directed *edges*.

When a node is labelled by a morphism $f \colon A_1 \otimes \cdots \otimes A_n \rightsquigarrow B_1 \otimes \cdots \otimes B_m$ (whichever the concrete parenthesising), it defines n *inner output ports* and m *inner input ports*, labelled by $A_1, \dots, A_n$ and $B_1, \dots, B_m$, respectively. Edges always connect an input port with a given label to an output port with the same label. In addition, all ports should be incident to exactly one edge, the graph should be acyclic, and its placement in the plane be a planar embedding: edges cannot cross. String diagrams are considered up to continuous deformations of their embedding in the plane: nodes and edges can be moved along as long as they do not cross.

We provide two examples in Figure 1: we depict string diagrams within a box, with outer input ports listed above and outer output ports listed below; nodes are also represented with boxes, with inner output ports above and inner input ports below (note the reversal). The string diagram on the left is the identity on an object A, it has a single outer input port and a single outer output port, both labelled A, no node, and a single edge. The string diagram on the right has three outer input ports labelled A, A, B, three outer output ports each labelled C, and three nodes with various interfaces.

String diagrams can be composed vertically and horizontally. Vertical composition corresponds to categorical composition ($;$): the outer output ports of the first diagram should match the outer input ports of the second one, which is plugged below it. Horizontal composition corresponds to tensor product ($\cdot$): there are no constraints, the two diagrams are simply put next to each other, with their outer ports concatenated.

Using those operations, we may associate a string diagram to every morphism expression. In particular, the diagram on the right of Figure 1 corresponds to the expression $(f \mathbin{;} g) \cdot h$, with $f \colon A \otimes A \rightsquigarrow B$, $g \colon B \rightsquigarrow A$ and $h \colon B \rightsquigarrow C \otimes C$ (a leaf in the expression is mapped to the string diagram with exactly one node labelled by that leaf).

As announced before, the point of string diagrams is that they abstract away irrelevant notational clutter. For instance, the three expression in the exchange law correspond to the three pictures on the left of Figure 2. While we have placed nodes in these three pictures according to the way they were recursively built from the expressions, this placement information is irrelevant: the three string diagrams are equivalent up to continuous deformations in the plane. Similarly, both sides of the bifunctoriality law are mapped to the same diagram on the right of Figure 2. Going back to Figure 1, the diagram on the right also corresponds to the expressions $f \cdot B \mathbin{;} g \cdot h$, $f \cdot h \mathbin{;} g \cdot C \cdot C$, or $f \cdot B \mathbin{;} B \cdot h \mathbin{;} g \cdot C \cdot C$, amongst others. In

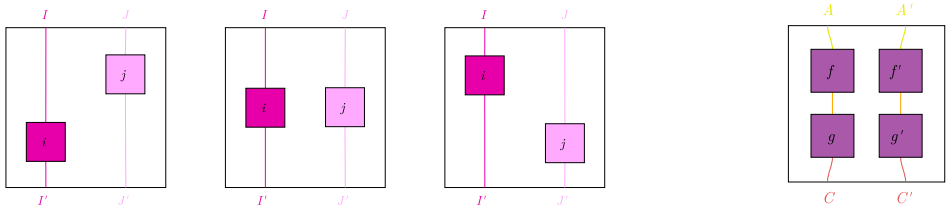


Figure 2 String diagrams for the exchange law $I{\cdot}j \;;\; i{\cdot}J' = i \cdot j = i{\cdot}J \;;\; I'{\cdot}j$ and bifunctoriality law $(f \;;\; g) \cdot (f' \;;\; g') = (f{\cdot}f') \;;\; (g{\cdot}g')$.

particular, the string diagram makes it clear that f and h can be put next to each other, in case we would need to use an assumption of the shape $f \cdot h = \cdots$. This information was less obvious in the initial expression $(f \;;\; g) \cdot h$.

This approach is safe thanks to the following completeness theorem: all expressions denoting a given diagram are provably equal from the axioms of monoidal categories [32].

Finally note that string diagrams are inherently strict: they are indexed by lists of objects rather than trees of objects, and associators and unitors are all mapped to identity string diagrams. Nevertheless, the aforementioned completeness theorem holds for arbitrary monoidal categories: string diagrams abstract away both bifunctoriality of tensor product, and coherence conditions from non-strict monoidal categories.

4 Graphical interface

We developed a standalone graphical editor for string diagrams. This program is written in OCaml, using the *vector graphics* [11] and *store* libraries [2]; it does not need to be trusted; we let it communicate with the proof assistant only via textual inputs and outputs.

This program is designed to perform three distinct tasks:

1. parse equational goals in monoidal categories; compute their string diagrams and render them graphically;
2. let the user delimit a subdiagram, detect whether this subdiagram matches some hypothesis, and substitute the subdiagram with the other member of the equation;
3. extract morphism expressions from string diagrams.

The first two items make it possible display the current state of a proof from the proof assistant, and to perform equational reasoning steps at the level of string diagrams. With appropriate library support (Section 5), the third one makes it possible to translate those graphical steps into textual ones in the proof assistant.

We discuss those three tasks in the next sections, illustrating them on the following running example. Here we present this example in strict monoidal categories for the sake of clarity; we will see in Section 5 how to formalise it in general ones, with no additional cost.

A *monoid* in a strict monoidal category $\mathcal{C}$ is an object M together with two morphisms $\mu \colon M \otimes M \rightsquigarrow M$ (the multiplication) and $\eta \colon 1 \rightsquigarrow M$ (the unit), such that the following equations hold (associativity, left unitality, right unitality):

$$\mu \cdot M \;;\; \mu = M \cdot \mu \;;\; \mu \qquad \eta \cdot M \;;\; \mu = M \qquad M \cdot \eta \;;\; \mu = M \;. \qquad \text{(monoid axioms)}$$

Monoids in the monoidal category of endofunctors are just *monads*, as known from the programming language community. A typical problem is to compose monads, and a standard solution consists in using distributive laws. In (strict) monoidal categories, this problem becomes: given two monoids M, N, define a monoid on $M \otimes N$. A distributive law in that case is a morphism $x \colon N \otimes M \rightsquigarrow M \otimes N$ satisfying the following coherence laws:

```
m : M⊗M ⇝ M
n : N⊗N ⇝ N
x : N⊗M ⇝ M⊗N
mA : m·M ; m ≡ M·m ; m
nA : n·N ; n ≡ N·n ; n
nx : n·M ; x ≡ N·x ; x·N ; M·n
mx : N·m ; x ≡ x·M ; M·x ; m·N
mn := M·x·N ; m·n
============================
mn·M·N ; mn ≡ M·N·mn ; mn
```

```
m : M⊗M ⇝ M
n : N⊗N ⇝ N
x : N⊗M ⇝ M⊗N
mA : m·M ; m ≡' M·m ; m
nA : n·N ; n ≡' N·n ; n
nx : n·M ; x ≡' N·x ;; x·N ;; M·n
mx : N·m ; x ≡' x·M ;; M·x ;; m·N
mn := M·x·N ;; m·n
==============================
mn·M·N ;; mn ≡' M·N·mn ;; mn
```

Figure 3 Running example: composing the multiplications $\mathtt{m} = \mu_M$ and $\mathtt{n} = \mu_N$ of two monoids M and N, via a distributive law x. The goal on the right will be discussed only in Section 5.1.

$$N \cdot \mu_M \ ; \ x = x \cdot M \ ; \ M \cdot x \ ; \ \mu_M \cdot N \qquad N \cdot \eta_M \ ; \ x = \eta_M \cdot N$$
$$\mu_N \cdot M \ ; \ x = N \cdot x \ ; \ x \cdot N \ ; \ M \cdot \mu_N \qquad \eta_N \cdot M \ ; \ x = M \cdot \eta_N \ .$$

The composite monoid can be defined as

$$\mu_{M\otimes N} \triangleq M \cdot x \cdot N \ ; \ \mu_M \cdot \mu_N \qquad \eta_{M\otimes N} \triangleq \eta_M \cdot \eta_M$$

and we have to show the monoid axioms for it, assuming the monoid axioms for M and N and the above coherence axioms about the distributive law x. We focus on multiplication and the associativity axiom in the sequel; the same methodology applies to prove left and right unitality, as can be seen in the accompanying code.

4.1 Parsing and rendering expressions

We use a text format which is flexible enough to encompass both Rocq goals under various notation systems, user annotations for standalone usage (e.g., about colours and shapes), and explicit diagram descriptions for debugging. This format includes types and hypothesis declarations, as well as the current goal. Parsing this format into abstract syntax trees is standard; we combine it with type inference to recover all implicit information and detect invalid inputs. Formalising our monoid composition example in Rocq results in the goal displayed in Figure 3, where `m` is the multiplication μ_M, `n` is μ_N, and `mn` is $\mu_{M\otimes N}$.

Most diagrams are initially described by terms, so that we can use the corresponding recursive structure to place them in the plane in a reasonable initial way. Then we use some force-directed dynamics in order to improve the placement: edges act as rubber bands, and vertical forces are adjusted so that nodes remain regularly distributed independently from the number of paths connecting them to the top and bottom interfaces. The initial placement is always a planar one; the user may freely move nodes but should be careful to do so in a continuous way – we plan to enforce this constraint in the future.

We assign colours to edges based on the name of the object labelling their endpoints. Similarly, we assign specific shapes to boxes of certain types: small triangles for morphisms of type $A \otimes A \rightsquigarrow A$ for some object A (here, multiplications), small circles for morphisms of type $A \otimes B \rightsquigarrow B \otimes A$ ("crossings", the distributive law here).

When starting from the Rocq goal in Figure 3, we obtain the rendering in Figure 4. Hypotheses are on the first row, and the conclusion is on the second one. We omit box and object names to alleviate the picture.

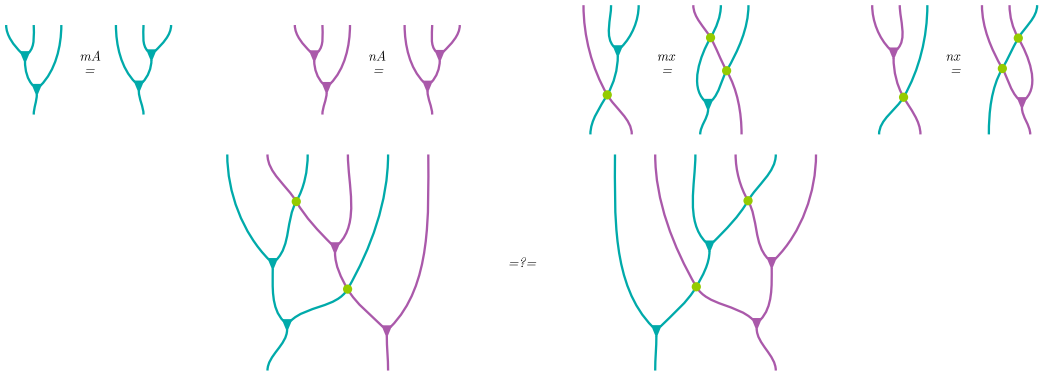


Figure 4 Graphical rendering of the goal from Figure 3.

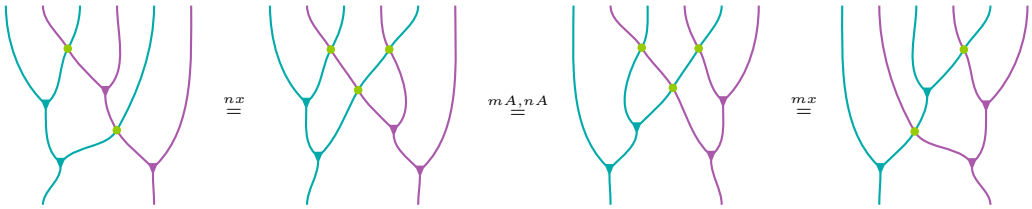
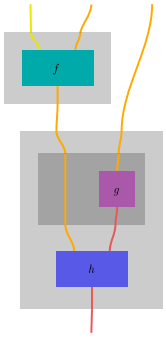


Figure 5 String diagrammatic proof for associativity of the composite monoid.

4.2 Isolating subdiagrams

Thanks to this graphical rendering of the goal, it becomes clearer how to proceed. The first two associativity axioms make it possible to swap two consecutive triangles with the same colour, while the two coherence axioms make it possible for a triangle to traverse a crossing downwards, generating two new crossings. To equate the two diagrams on the bottom line, we can let the two triangles in the middle traverse the two bottom crossings, and then swap the two consecutive triangles on the left and on the right. Such a proof is depicted in Figure 5.

Our graphical interface makes it possible to proceed as follow: by surrounding a subdiagram with the pointer, we can create a box around it. If the subdiagram within the box matches a member of one of the hypotheses then it is highlighted, and we can press a button to replace it with the other member of the hypothesis (Figure 6). For this to work we need an algorithm to detect when two diagrams are equivalent, which is essentially an isomorphism test in our case, and a notion of *box*. In the graphical editor, this is done by generalising the notion of string diagram: we allow nodes to be labelled either by a morphism name as before, or by a string diagram, recursively. Accordingly, our syntax for expressions includes a notation [_] for boxes. For instance, for morphisms $f\colon A \otimes B \rightsquigarrow B$, $g\colon B \rightsquigarrow C$, and $h\colon B \otimes C \rightsquigarrow C$, the diagram of the expression $[f] \cdot B \ ; \ [[B \cdot g] \ ; \ h]$ is depicted below.



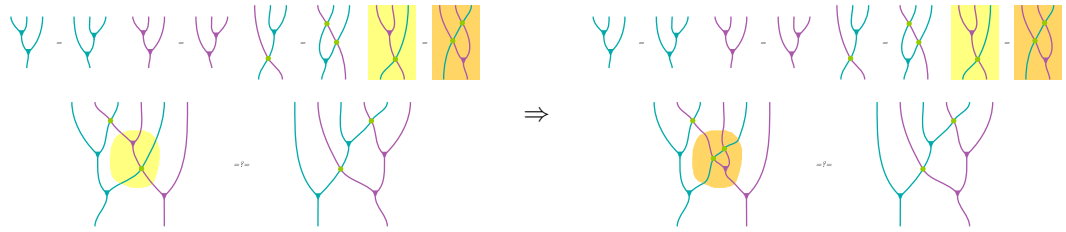


Figure 6 Performing a rewriting step graphically, by surrounding a matching subdiagram.

In Figure 6, the yellow and orange hand-written shapes are such boxes: new nodes in the top-level diagrams, with their own inner nodes and ports.

The two key operations on string diagrams with boxes are *boxing* and *unboxing*. Unboxing inlines the content of a box within the surrounding diagram; it is mainly a matter of rewiring graphs. Boxing is more involved: given a polygon (the shape drawn by the user), we first have to compute the interface of the subdiagram to be created. To this end, we compute the intersections of the polygon with the edges of the diagram and we check that each edge intersects at most twice; these intersections are computed in an oriented way, so that we can detect whether they correspond to inputs or to outputs. Then we check that there are exactly zero or two alternations between inputs and outputs along the polygon, so that we can extract a list of inputs and a list of outputs: the interface of our subdiagram. We finally check that all nodes within the polygon form a valid diagram with this interface, we turn it into a box node and we rewire the starting diagram accordingly.

4.3 Extracting expressions

The aforementioned program should not be part of the trusted code-base. In order to validate the graphical rewriting steps in the proof assistant, we need a way to translate them back to the standard, textual format.

To this end, we need a function that extracts an expression from a given string diagram. Potentially many expressions yield a given string diagram, but any of them will do thanks to the Rocq tactics we develop in Section 5. Still, we define this function so that it returns human-friendly expressions. In fact, it computes expressions which are in *tetris normal form*, in the following sense:

1. the expression is a composition of tensor products of identity morphisms and *atoms*: boxes of normal forms or leaf morphisms;
2. each atom appears as right as possible in this composition.

This corresponds to an algorithm where we read the string diagram line by line, bottom up, each time gathering as many nodes as possible, and recursively extracting expressions when we reach box nodes.

Consider for instance the string diagram on the right of Figure 1. The first line we read, at the bottom, is $g \cdot h$; we could have read $g \cdot C \cdot C$ (or $C \cdot h$, or $C \cdot C \cdot C$), but these choices are not optimal. Then the next and final line is $f \cdot B$, so that we return $f \cdot B \ ; \ g \cdot h$. Applying the same procedure to the bottom-left diagram of Figure 4, we get the expression

$$M \cdot x \cdot N \cdot M \cdot N \ ; \ M \cdot M \cdot n \cdot M \cdot N \ ; \ m \cdot x \cdot N \ ; \ m \cdot n \ .$$

Note that is expression is quite different from the starting one ($mn \cdot M \cdot N \ ; \ mn$ with $mn \triangleq M \cdot x \cdot N \ ; \ m \cdot n$) and that although labels m, n, x, M, N are omitted in the picture, they are recorded in the implementation, precisely to be able to extract expressions. Applying it to the bottom-left diagram on the left of Figure 6, we get the expression

$$M \cdot x \cdot N \cdot M \cdot N \ ; \ m \cdot [n \cdot M \ ; \ x] \cdot N \ ; \ m \cdot n \ .$$

Observe here that the fact we have highlighted a subdiagram completely changes the extracted expression. This is on purpose: now the left-hand side of hypothesis nx matches the subexpression within brackets, and we get to see why we can rewrite using this hypothesis.

5 Rocq library support

Now we describe the library support we provide in Rocq to formalise proofs in monoidal categories, potentially with the help of the previous graphical tool.

This involves two main tasks: setting up powerful notations that give the illusion of working in strict monoidal categories (Section 5.1), and designing a tactic to solve automatically any equation whose members denote the same string diagram. This second task requires us to prove a generalised version of MacLane's coherence theorem (Section 5.2), and to implement a decision procedure for general expressions (Section 5.3).

5.1 Inferring MacLane isomorphisms

If we try to generalise our definition of monoids in strict monoidal categories to the non-strict case, we see that the monoid axioms do not type-check, *as is*. Indeed, we should insert appropriate associators and unitors:

$$\mu \cdot M \ ; \ \mu = \alpha_{M,M,M} \ ; \ M \cdot \mu \ ; \ \mu \qquad \eta \cdot M \ ; \ \mu = \lambda_M \qquad M \cdot \eta \ ; \ \mu = \rho_M \ .$$

The coherence axioms for distributive laws must be fixed in a similar way, as well as our definition of the multiplication and the unit of the composite monoid. We leave this as an exercise to the reader[1]: our goal in Rocq is to leave it to the assistant.

We achieve this task by designing a reflexive tactic [8] to infer automatically morphisms between two tensor-unit expressions denoting the same list of atomic objects: we say in this case that the tensor expressions are equivalent. Thanks to this tactic, which we bind to the constructor `mcl` of an ad-hoc typeclass, we can write

```
mcl: A ⊗ (B ⊗ (1 ⊗ C)) ⇝ (1 ⊗ (A ⊗ B)) ⊗ C} .
```

There, `mcl` stands for "MacLane isomorphism", as it always infers (iso)morphisms built from the associators (α) and the left and right unitors (λ, ρ). We use this notation to construct slightly more high-level ones[2]:

```
f ;; g := (f ; mcl ; g)     cast f := (mcl ; f ; mcl)    f ≡' g := (f ≡ cast g)} .
```

Intuitively: `f;;g` makes it possible to compose two morphisms when the target of f is only equivalent to the source of g, `cast f` makes it possible to view f as a morphism of a different yet equivalent type, and `f ≡' g` makes it possible to compare morphisms with distinct yet equivalent types (note the asymmetry in this last notation, whence the asymmetric symbol).

Using these notations, we can fix the goal on the left of Figure 3, which does not typecheck, into the one on the right which does in any monoidal category: we can write expressions in monoidal categories as if they were definitionally strict.

The tactic works by reifying the tensor expressions into trees of objects, and showing that every tree of objects can be put in list normal form via a MacLane isomorphism. We call list of objects *arities*; when the two trees denote the same arity, we get the desired morphism by

---

[1] Answer for the first coherence law: $N \cdot \mu_M \ ; \ x = \alpha^{-1}_{N,M,M} \ ; \ x \cdot M \ ; \ \alpha_{M,N,M} \ ; \ M \cdot x \ ; \ \alpha^{-1}_{M,M,N} \ ; \ \mu_M \cdot N$.

[2] This is pseudocode: the actual notations are defined through proper definitions triggering our dedicated tactic via typeclass resolution. We moreover use bidirectional typing hints [31] so that type information flows in such a way that our tactic gets to know the tensor-unit expressions when it is called.

```
Inductive tree :=
| t_ob(A: C)
| t_unit
| t_tensor(a b: tree).
Fixpoint eval_tree: tree → C := ...

Inductive maclane: tree → tree → Type :=
| mcl_id: ∀ a, maclane a a
| mcl_comp: ∀ a b c, maclane a b → maclane b c → maclane a c
| mcl_tensor: ∀ a b c d, maclane a b → maclane c d → maclane (a⊗c) (b⊗d)
| mcl_inv: ∀ a b, maclane a b → maclane b a
| mcl_assoc: ∀ a b c, maclane ((a⊗b)⊗c) (a⊗(b⊗c))
| mcl_unitl: ∀ a, maclane (1⊗a) a
| mcl_unitr: ∀ a, maclane (a⊗1) a.
Fixpoint eval_maclane a b (u: maclane a b): eval_tree a ⇝ eval_tree b := ...

Definition arity := list C.
Definition Nf: tree → arity := ...

Lemma find_MacLane a b: Nf a = Nf b → maclane a b.
Theorem MacLane a b (f g: maclane a b): eval_maclane f ≡ eval_maclane g.
```

Figure 7 Standard MacLane's coherence theorem.

going to this list normal form, and back. This is essentially how we get Lemma `find_MacLane` in Figure 7, where we provide the key datatypes and functions. Concretely on the above example, the notation `mcl` yields the following term, under the hood:

```
eval_maclane (find_Maclane
                (t_tensor (t_ob A) (t_tensor (t_ob B) (t_tensor t_unit (t_ob C))))
                (t_tensor (t_tensor t_unit (t_tensor (t_ob A) (t_ob B))) (t_ob C))
                eq_refl): A⊗(B⊗(1⊗C)) ⇝ (1⊗(A⊗B))⊗C .
```

5.2 MacLane's coherence theorem

The above approach is safe thanks to MacLane's coherence theorem for monoidal categories: all MacLane isomorphisms between two given tensor expressions are provably equal [32]. This property is crucial: it ensures that the statements we obtain do not depend on the specific choice of morphisms automatically inferred by our notations.

Accordingly, we prove this theorem, and we provide a high-level tactic `MacLane`, that proves the equality of two morphism expressions differing only on MacLane isomorphisms. This tactic is a cornerstone for the subsequent development: we use it around 40 times explicitly to solve various goals when we verify the decision procedure from Section 5.3, and around 170 times implicitly, via calls triggered by typeclass resolution.

We prove a first version of the theorem following closely the strategy proposed by Ilya Beylin and Peter Dybjer [5]. Its formal statement is given in Figure 7; a technical point is that this coherence proof relies on a meta-coherence property: we need uniqueness of identity proofs on lists of objects of the starting category to conclude. For now we have added this requirement to our definition of monoidal category; we plan to lift it in a future version by using variable maps during reification, so that we can rely on uniqueness of identity proofs

```
Inductive gtree :=
| gt_ob(A: 𝒞)
| gt_unit
| gt_tensor(s t: gtree)
| gt_trees(h: gtrees)
with gtrees :=
| gt_ar (Γ: arity)
| gt_nf (t: tree)    (* normal form of a plain tree *)
| gt_gnf (t: gtree)  (* normal form of a generalized tree *)
| gt_nil
| gt_cons (s: gtree) (h: gtrees)
| gt_app (h k: gtrees).
Fixpoint eval_gtree: gtree → 𝒞 := ...
with eval_gtrees: gtrees → arity := ...

Fixpoint gtree_tree: gtree → tree := ...
Definition gmaclane (a b: gtree) := maclane (gtree_tree a) (gtree_tree b).
Definition eval_gmaclane a b (u: gmaclane a b): eval_gtree a ⇝ eval_gtree b
  := cast (eval_maclane u).  (* casting proofs given explicitly in the real code *)

Definition gNf: gtree → arity := ...

Lemma find_gMacLane a b: gNf a = gNf b → gmaclane a b.
Theorem gMacLane a b (f g: gmaclane a b): eval_gmaclane f ≡ eval_gmaclane g.
```

Figure 8 Extended MacLane's coherence theorem.

on lists of natural numbers rather than lists of objects, which is derivable in the calculus of inductive constructions. (Such a step is standard in tactics by reflection [20, 36], but it requires more work during reification to build the variable maps.)

While this first version of MacLane's theorem should be enough for end-users of the library, we needed to refine it to formalise the proofs required in Section 5.3 below. Indeed, in addition to abstract objects and tensor trees of such objects, we frequently need to manipulate arities (lists of objects), to infer MacLane isomorphisms between types involving such arities, and to prove equality of such isomorphisms.

For instance, if Γ, Δ are arities and `ev: arity` $\mathcal{C} \to \mathcal{C}$ is a shorthand for the semantic evaluation function of arities (`eval_arity`), we need the following automatic inferences:

```
mcl: ev (Γ++Δ) ⇝ ev Γ⊗ev Δ     mcl: ev (Γ++(A⊗B)::Δ) ⇝ ev Γ⊗A⊗B⊗ev Δ .
```

To get them, we use an extended syntax of "generalised trees", displayed in Figure 8. This syntax makes it possible to reify `ev` and the operations on arities (`::`, `++`); for instance, we reify the third expression above as

```
gt_trees (gt_app (gt_ar Γ) (gt_cons (gt_tens (gt_ob A) (gt_ob B)) (gt_ar Δ))) ,
```

whose normal form (via `gNf`) is `[ev Γ; A; B; ev Δ]`.

We manage to build on the standard MacLane's theorem (Figure 7) in order to setup this extended infrastructure (rest of Figure 8); still, the fact that we have to record trees in the type of MacLane isomorphisms makes it technically non-trivial.

```
Inductive tmor: gtree → gtree → Type :=
| t_var: ∀ a b, (eval_gtree a ⇝ eval_gtree b) → tmor a b
| t_mcl: ∀ a b, gNf a = gNf b → tmor a b
| t_comp: ∀ a b c, tmor a b → tmor b c → tmor a c
| t_tens: ∀ a b c d, tmor a b → tmor c d → tmor (gt_tens a c) (gt_tens b d).
Fixpoint eval_tmor a b (u: tmor a b): eval_gtree a ⇝ eval_gtree b := ...

Inductive amor: arity → arity → Type :=
| a_var: ∀ n m, (eval_arity n ⇝ eval_arity m) → amor n m
| a_id: ∀ n, amor n n
| a_comp: ∀ n m p, amor n m → amor m p → amor n p
| a_tens: ∀ n m p q, amor n m → amor p q → amor (n++p) (m++q).
Fixpoint eval_amor n m (u: amor n m): eval_arity n ⇝ eval_arity m := ...

Fixpoint gNf_mor a b (u: tmor a b): amor (gNf a) (gNf b) := ...
Theorem eval_gNf_mor a b (f: tmor a b): eval_tmor f ≡' eval_amor (gNf_mor f).
```

Figure 9 Strictifying reified morphism expressions.

5.3 Decision procedure

The aforementioned tactic `MacLane` makes it possible to deal with the bureaucracy induced by MacLane morphisms: it automates the use of the triangle and pentagon axioms from the definition of monoidal category, together with naturality of the associators and unitors. It remains to help the user with bifunctoriality, which is also extremely painful to use manually.

To this end, we develop a tactic `mcat` to solve equations in monoidal categories. Given two morphism expressions f, g, this tactic proves $f = g$ when f and g denote the same string diagram. However, formalising string diagrams and equivalence up to continuous deformations seems quite challenging. Instead, we choose to restrict to the case where all nodes have at least one output port (i.e., there is no morphism of type $A \rightsquigarrow 1$ for some A). This restriction makes it possible to completely bypass this difficulty. Indeed, when all nodes have at least one output port, all nodes in a string diagram are reachable bottom-up from its outputs, and the expressions in normal form computed by our standalone graphical editor are unique. Therefore, to solve this fragment, it suffices to show that every morphism expression can be put in normal form using the axioms of monoidal categories.

Even if this process is simpler than formalising string diagrams, it remains non-trivial. We proceed in two steps. First we define a syntax for end-user morphisms indexed by (generalised) trees, which we translate to a syntax of strict morphisms, indexed by arities. The datatypes and the key definitions and statements are given in Figure 9; in a sense this step corresponds to strictification, at a reified level: the source expressions contain non-trivial MacLane morphisms (constructor `t_mcl`), while the target expressions restrict this construction to plain identity morphisms (constructor `a_id`). Theorem `eval_gNf_mor` can be proved in a few lines including six calls to the aforementioned `(g)MacLane` tactic.

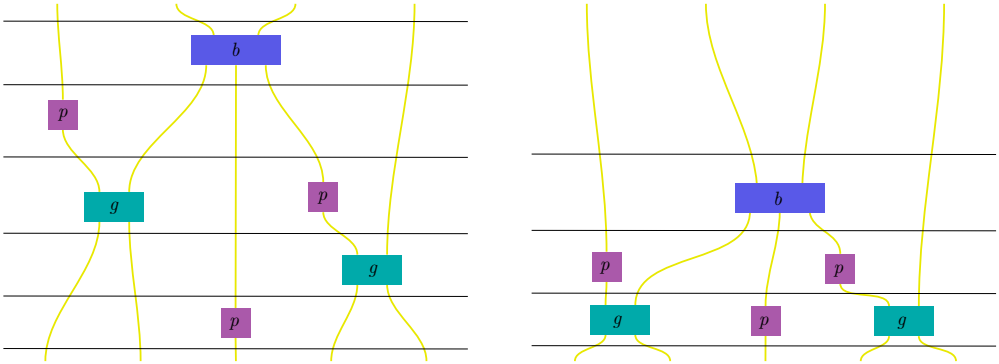
This first step is important: it makes it possible to work on a simplified syntax for the subsequent normalisation function. Recall that a tetris normal form is a composition of tensor products, where atoms in the tensor products are pushed as far as possible to the right (i.e., towards bottom in the string diagrams). We use the datatype given in Figure 10: a normal form is a list of rows, where a row is a list of atoms interleaved with identity morphisms. This datatype does not ensure that atoms are pushed as far as possible to the right (i.e., towards the tail of the list of rows): while our normalisation function enforces this requirement, proving formally that it does so is not needed for our tactic to work.

```
Inductive row: arity → arity → Type :=
| rnil: ∀ n, row n n
| rcons: ∀ n m m' (u: eval_arity m ⇝ eval_arity m') p p',
        row p p' → row (n ++ m ++ p) (n ++ m' ++ p').
Fixpoint eval_row n m (r: row n m): eval_arity n ⇝ eval_arity m := ...

Inductive nmor n: arity → Type :=
| nnil: nmor n n
| ncons: ∀ m p, nmor n m → row m p → nmor n p.
Fixpoint eval_nmor n m (u: nmor n m): eval_arity n ⇝ eval_arity m := ...

Fixpoint norm n m (u: amor n m): nmor n m := ...
Theorem eval_norm n m (u: amor n m): eval_nmor (norm u) ≡ eval_amor u.
```

Figure 10 Datatype for normal forms, and normalisation function.



$[[1; b; 1]; [0; p; 4]; [0; g; 1; p; 1]; [3; g; 0]; [2; p; 2]]$ $[[1; b; 1]; [0; p; 2; p; 1]; [0; g; 0; p; 0; g; 0]]$

Figure 11 Playing tetris on string diagrams: the five rows on the left have to be normalised into the three rows on the right. In the textual representation, arities are abstracted via their length.

The normalisation rests on 13 auxiliary functions. A first part of it consists in translating general morphisms into lists of rows, which is not too hard. The rest of it intuitively implements the tetris game: given a list of rows, let the atoms jump from one row to the next one down the list, as far as possible, as illustrated in Figure 11.

This problem reduces to defining a function that takes two rows, and returns either a single row if all atoms from the first one can be inserted in the second one, or the two rows where all atoms from the first one that can be inserted in the second one have been so. Writing such a function is an interesting exercise *per se*; here it is complicated by the fact that rows are dependently typed: we should carry all case analyses in such a way that we preserve typing using appropriate explicit casts, and do so in such a way that those casts do not block computations later when we execute the normalisation function.

We implement this function using 9 additional functions working on *relative arities*, that make it possible to compare two rows whose internal interfaces are slightly desynchronised (by a relative arity). The final function (`zrcomp` in the code) is written using tactics, it consists of 478 words spanning over 63 lines. The correctness of our normalisation function (Theorem `eval_norm`) spans over 400 lines, 280 of which being dedicated to the analysis of the function `zrcomp`. This proof remains tractable thanks to the `MacLane` tactic, which we use heavily to automate monoidal reasoning steps – we trigger this tactic automatically via typeclass resolution when we apply various simplification lemmas.

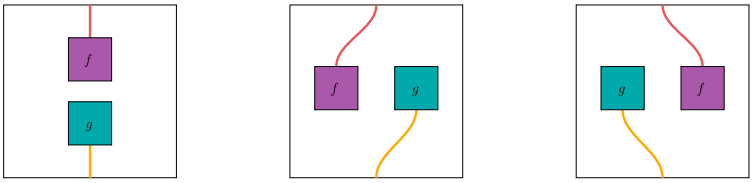


Figure 12 Non uniqueness of normal forms in the presence of empty-target morphisms.

We finally implement reification (in Ltac for now), and we pack everything into a reflexive tactic `mcat`. For instance, we may use it to prove that any two expressions for the string diagram on the right of Figure 1 are equal:

```
Goal ∀ f: A⊗A ⇝ B, ∀ g: B ⇝ C, ∀ h: B ⇝ C⊗C, (f;g)·h ≡ f·h ; g·C·C.
Proof. intros. mcat. Qed.
```

We do not consider completeness of the `mcat` tactic, which would require us to prove that we indeed compute normal forms, and that those are unique when all morphisms have at least one output. To see that normal forms are not unique in the general case, consider morphisms $f\colon C \rightsquigarrow 1$ and $g\colon 1 \rightsquigarrow B$. The diagram for $f;g$ is depicted on the left of Figure 12, and it is equivalent to the two diagrams on the right: when playing the tetris game, we may chose to move f either on the left or on the right of g, resulting in two distinct normal forms: $f \cdot g$ and $g \cdot f$. Accordingly, the `mcat` tactic may fail to prove valid equations involving empty-target morphisms: it fails on $f \cdot g = g \cdot f$ for instance.

5.4 Composing monoids, formally

Now we have all the necessary material to complete a formal proof of our running example about composition of monoids. Starting from the Rocq goal in Figure 3, we can copy-paste it in the diagram editor and render it graphically (Figure 4). There we may proceed with graphical rewriting steps, a potential first one being depicted in Figure 6. When we reach two equivalent diagrams, the editor detects it and exports a proof script which we can copy-paste and execute in Rocq.

Such a proof is given in Figure 13; it consists of four transitivity steps which make explicit the pattern which is next rewritten, within brackets. Each of these transitivity steps is validated by a call to `mcat`: they consist in moving from one expression for some diagram to another expression for the same diagram. A final call to `mcat` validates the fact that we have indeed reached the same diagram after the four rewriting steps.

Boxes have a specific behaviour in the diagram editor (Section 4.2). In Rocq, they are implemented via an identity function, bound to the bracket notation to highlight the pattern to be rewritten. The `rewrite` tactic we use here is a variant of the standard one, which only looks at occurrences of this bracket notation and deals with the asymmetry of ≡' when rewriting from right to left.

A key feature of the proof we obtain is that it is readable, robust to slight changes in the statement, and easily editable. The proof can also be replayed graphically step by step: each intermediate goal can be imported back in the editor to see what are the diagrams at that point – a prototype currently developed with the help of Johann Rosain makes it possible to automate this task from emacs/proofgeneral.

```
Lemma mnA: mn·M·N ;; mn ≡' M·N·mn ;; mn.
Proof.
  unfold mn.

  transitivity (M·x·N·M·N ;; m·[n·M ; x]·N ;; m·n). mcat.
  rewrite nx.

  transitivity (M·N·M·x·N ;; M·[N·m ; x]·n ;; m·n). 2: mcat.
  rewrite mx.

  transitivity (M·x·x·N ;; M·M·x·N·N ;; M·M·M·n·N ;; [m·M ; m]·n). mcat.
  rewrite mA.

  transitivity (M·x·x·N ;; M·M·x·N·N ;; M·m·N·N·N ;; m·[N·n ; n]). 2: mcat.
  rewrite -nA.

  mcat.
Qed.
```

Figure 13 Associativity of the multiplication for a composite monoid, formally.

In the above workflow, the user switches once from the proof assistant to the diagram, editor and once from the editor back to the assistant. Other workflows are possible: the user may interleave proof steps performed with the help of the diagram editor and proof steps performed directly in the assistant.

We may also optimise the proof manually: on this example, some rewriting steps can be performed in parallel, and we can use the diagram editor to extract expressions on which two rewriting steps can be applied, simply by surrounding the two matches and asking the editor to export the corresponding expressions. This is how we obtained the variant in Figure 14, for instance.

For the sake of comparison, we provide an alternative proof in Figure 15. This proof is displayed as traditional commutative diagram (not a string diagram). The two squares marked with uni-coloured disks correspond to the two steps where with use associativity of the starting multiplications; the two pentagons marked with mixed-colour disks correspond to the two steps where we use the coherence axioms of the distributive law. The remaining 8 squares are bifunctoriality, and the 4 triangles are the exchange law. Formalising those 12 steps explicitly would be quite painful as each of them would require us to specify precisely where to apply those laws in the considered expressions, modulo associativity of composition. In contrast, these 12 steps are entirely implicit in the string diagrammatic proofs we formalised (Figures 5, 13 and 14): they are covered automatically by the `mcat` tactic.

Also note that the commutative diagram in Figure 15 is only a proof in *strict* monoidal categories. A fully detailed commutative diagram in general monoidal categories would not fit in a page as it would require the insertion of many associators and unitors, as well as many uses of naturality and the triangle and pentagon coherence axioms (which are also dealt with by `mcat` in our case, which supersedes `MacLane`).

The complete proof for composing monoids, where we also deal with the unit laws of the composite monoid, can be found in the accompanying code, as well as the exercise consisting in composing three monoids, using three distributive laws.

```
Lemma mnA: mn·M·N ;; mn ≡' M·N·mn ;; mn.
Proof.
  unfold mn.

  transitivity (M·x·N·M·N ;; M·M·[n·M ; x]·N ;; [m·M ; m]·n). mcat.
  rewrite nx.
  rewrite mA.

  transitivity (M·N·M·x·N ;; M·[N·m ; x]·N·N ;; m·[N·n ; n]). 2: mcat.
  rewrite mx.
  rewrite -nA.

  mcat.
Qed.
```

Figure 14 Manually optimised formal proof.

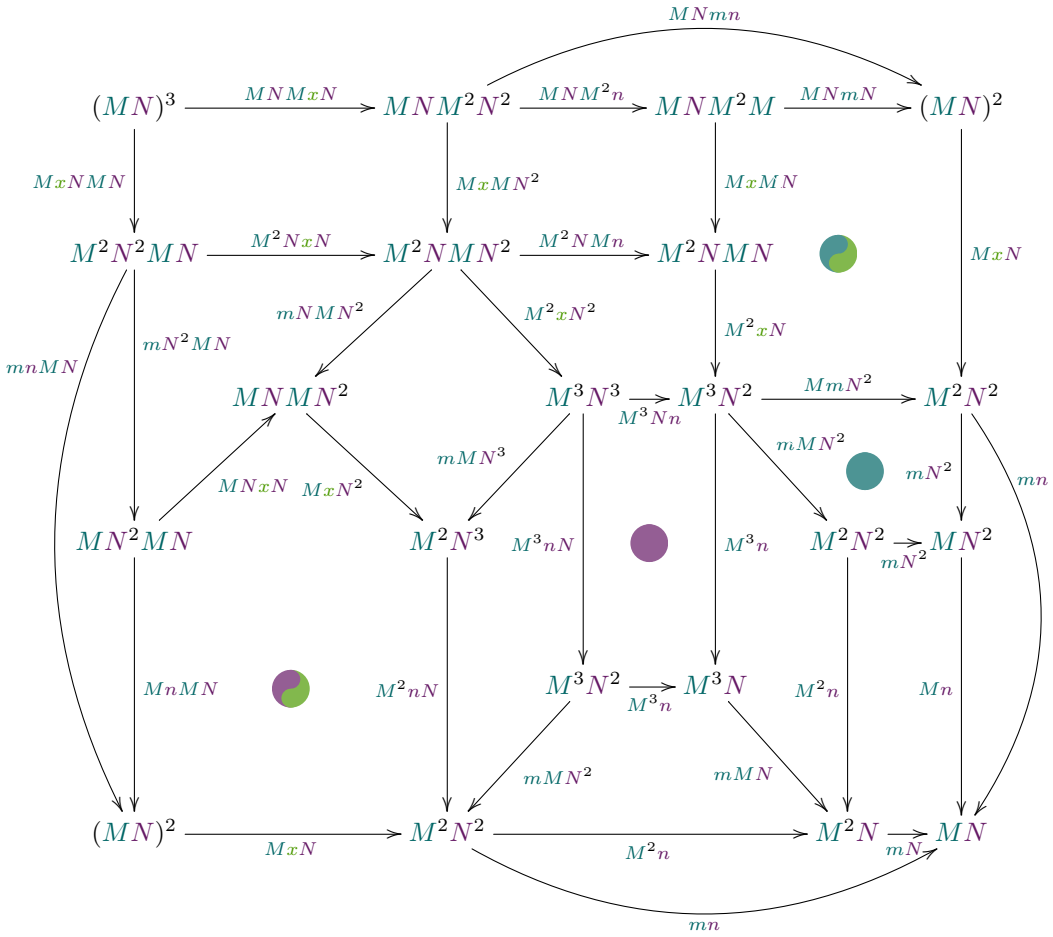


Figure 15 Proof using commutative diagrams rather than string diagrams.

6 Related work

There are several standalone domain-specific proof assistants or visualisers for various kinds of categories in the literature: Globular[3] [4], homotopy.io[4] [15], Quantomatic [27], or Chyp [26] which is a graphical prover for symmetric monoidal categories, implementing the theory from [6]. The latter is the closest in spirit to our work, but none of them connects to general purpose proof assistants.

Sam Ezeh proposed a visualiser for string diagrams in Lean 4 [18], and makes it possible to perform rewriting steps from the visualiser, but only in the strict monoidal category of endofunctors, only for specific forms of hypotheses (naturality, associativity of a monad), and only modulo associativity of composition.

In Rocq, Bhakti Shash et al. developed a visualiser for symmetric monoidal categories [37], together with applications to the ZX calculus [30]. They formalise MacLane's coherence theorem, but they do not provide tactics to infer MacLane isomorphisms or prove general morphism equalities, nor a way to perform rewriting steps graphically.

Leaving monoidal categories aside, Ambroise Lafont designed a diagram editor[5] [28] in which one can perform categorical proofs graphically and export them to Rocq. In the same vein, Luc Chabassier designed an integrated graphical interface for Rocq with similar objectives [13, 12]. In both cases, the tools do not deal with string diagrams but with standard commutative diagrams in general category theory (as in the proof given in Figure 15).

7 Directions for future work

Besides improving `mcat` to make it complete even in the presence of empty target morphisms, which seems quite challenging, let us mention three main directions for future work.

First we would like to extend the Rocq development to deal with *bicategories*; those generalise monoidal categories (these are one-object bicategories), and they share the same string diagrams. Extending the graphical editor would be easy, but extending the Rocq decision tactic would require us to add one more layer of dependent types to our normalisation procedure, which is likely to be unpleasant.

Second we would like to deal with *symmetric* monoidal categories. On the one hand, the diagrams become simpler: the planarity constraint disappears, and diagrams become plain graphs: "only connectivity matters". On the other hand, it is more difficult to define a satisfactory notion of normal form for expressions, as we have to record permutations at several places in expressions.

Lastly, while we advocate for a clear separation between the graphical editor and the proof assistant, we could also propose a more integrated version. This would make it possible to render the current goal without any user interaction, but also to design tactics that do not display any string diagram but compute them in the background to implement specific tasks, for instance to find matches modulo the monoidal category axioms automatically, or to implement specific equality saturation procedures.

[3] `http://globular.science`
[4] `https://beta.homotopy.io`
[5] `https://amblafont.github.io/graph-editor`

References

1 Benedikt Ahrens, Dan Frumin, Marco Maggesi, Niccolò Veltri, and Niels van der Weide. Bicategories in univalent foundations. *Mathematical Structures in Computer Science*, 31(10):1232–1269, 2021. doi:10.1017/S0960129522000032.

2 Clément Allain, Basile Clément, Alexandre Moine, and Gabriel Scherer. Snapshottable stores. *Proc. ACM Program. Lang.*, 8(ICFP), 2024. doi:10.1145/3674637.

3 Samuel Arsac, Russ Harmer, and Damien Pous. Adhesive category theory for graph rewriting in Rocq. In *Proc. CPP*, pages 59–74. ACM, 2026. doi:10.1145/3779031.3779105.

4 Krzysztof Bar, Aleks Kissinger, and Jamie Vicary. Globular: An Online Proof Assistant for Higher-Dimensional Rewriting. In *Proc. FSCD*, volume 52 of *LIPIcs*, pages 34:1–34:11. Schloss Dagstuhl, 2016. doi:10.4230/LIPIcs.FSCD.2016.34.

5 Ilya Beylin and Peter Dybjer. Extracting a proof of coherence for monoidal categories from a proof of normalization for monoids. In *Proc. TYPES, selected Papers*, volume 1158 of *LNCS*, pages 47–61. Springer, 1995. doi:10.1007/3-540-61780-9_61.

6 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. doi:10.1017/S0960129522000317.

7 Francis Borceux. *Handbook of Categorical Algebra: Volume 1: Basic Category Theory*. Encyclopedia of Mathematics and Its Applications. Cambridge University Press, 1994. doi:10.1017/CBO9780511525858.

8 Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *Proc. TACS*, pages 515–529. Springer, 1997. doi:10.5555/645869.668533.

9 Jean Bénabou. Catégories avec multiplication. *Compte rendus de l'académie des sciences*, 256, 1963.

10 Jean Bénabou. Les catégories multiplicatives. *Séminaire de mathématique pure*, 27, 1972. URL: https://ncatlab.org/nlab/files/Benabou-CategoriesMultiplicatives.pdf.

11 Daniel Bünzli. Vector graphic, 2013. URL: https://erratique.ch/software/vg.

12 Luc Chabassier. *Aspects of Category Theory in Proof Assistants*. Thèse de doctorat, Université Paris-Saclay, July 2025. URL: https://theses.fr/2025UPASG055.

13 Luc Chabassier. A Graphical Interface for Category Theory Proofs in Coq. *Electronic Proceedings in Theoretical Computer Science*, pages 28–41, 2025. doi:10.4204/EPTCS.419.2.

14 Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy Builder: Algebraic hierarchies made easy in Coq with Elpi. In *Proc. FSCD*, pages 34:1–34:21, 2020. doi:10.4230/LIPIcs.FSCD.2020.34.

15 Nathan Corbyn, Lukas Heidemann, Nick Hu, Chiara Sarti, Calin Tataru, and Jamie Vicary. homotopy.io: A Proof Assistant for Finitely-Presented Globular n-Categories. In *Proc. FSCD*, volume 299 of *LIPIcs*, pages 30:1–30:26. Schloss Dagstuhl, 2024. doi:10.4230/LIPIcs.FSCD.2024.30.

16 Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast, embeddable, λprolog interpreter. In *Proc. LPAR*, LNCS, pages 460–468. Springer, 2015. doi:10.1007/978-3-662-48899-7_32.

17 Samuel Eilenberg and Max Kelly. Closed categories. In *Proceedings of the Conference on Categorical Algebra*, pages 421–562. Springer, 1966. Section II.1. doi:10.1007/978-3-642-99902-4.

18 Sam Ezeh. Graphical Rewriting for Diagrammatic Reasoning in Monoidal Categories in Lean4. In *Proc. ITP*, volume 309 of *LIPIcs*, pages 41:1–41:8. Schloss Dagstuhl, 2024. doi:10.4230/LIPIcs.ITP.2024.41.

19 François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging Mathematical Structures. In *Proc. TPHOLs*, pages 327–342, 2009. doi:10.1007/978-3-642-03359-9_23.

20 Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In *Proc. TPHOL*, volume 3603 of *LNCS*, pages 98–113. Springer, 2005. doi:10.1007/11541868_7.

21 Günter Hotz. Eine algebraisierung des syntheseproblems von schaltkreisen. *EIK, Bd. 1, (185-205), Bd, 2, (209-231)*, 1965.

22 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.

23 Max Kelly. On MacLane's conditions for coherence of natural associativities, commutativities, etc. *Journal of Algebra*, 1:397–402, 1964.

24 Max Kelly. Many-variable functorial calculus. I. In *Coherence in Categories*, pages 66–105. Springer, 1972. doi:10.1007/BFb0059556.

25 Yoshiki Kinoshita. A bicategorical analysis of e-categories, February 1997. URL: https://www.researchgate.net/publication/2710125_A_bicategorical_analysis_of_E-categories.

26 Aleks Kissinger. Chyp, 2023. URL: https://github.com/akissinger/chyp.

27 Aleks Kissinger and Vladimir Zamdzhiev. Quantomatic: A proof assistant for diagrammatic reasoning. In *Proc. CADE*, pages 326–336, 2015.

28 Ambroise Lafont. A diagram editor to mechanise categorical proofs. In *Proc. JFLA*, 2024. URL: https://hal.science/hal-04407118.

29 Saunders Mac Lane. Natural associativity and commutativity. *Rice University Studies*, 49:28–46, 1963.

30 Adrian Lehmann, Ben Caldwell, Bhakti Shah, William Spencer, and Robert Rand. VyZX: Formal verification of a graphical quantum language. *ACM Transactions on Programming Languages and Systems*, April 2026. Just Accepted. doi:10.1145/3807780.

31 Meven Lennon-Bertrand. Complete bidirectional typing for the calculus of inductive constructions. In *Proc. ITP*, LIPIcs, pages 24:1–24:19. Schloss Dagstuhl, 2021. doi:10.4230/LIPIcs.ITP.2021.24.

32 Saunders Mac Lane. *Categories for the Working Mathematician*. Graduate texts in mathematics 5. Springer, 2nd ed., reprint edition, 1971 - 1998.

33 Roger Penrose. Applications of negative dimensional tensors. *Combinatorial Mathematics and its Applications*, 1971.

34 Roger Penrose and Wolfgang Rindler. *Spinors and Space-Time*. Cambridge Monographs on Mathematical Physics. Cambridge University Press, 1984. doi:10.1017/CBO9780511564048.

35 Robin Piedeleu and Fabio Zanasi. *An Introduction to String Diagrams for Computer Scientists*. Elements in Applied Category Theory. Cambridge University Press, 2025. doi:10.1017/9781009625715.

36 Damien Pous. Kleene Algebra with Tests and Coq tools for while programs. In *Proc. ITP*, volume 7998 of *LNCS*, pages 180–196. Springer, 2013. doi:10.1007/978-3-642-39634-2_15.

37 Bhakti Shah, Willam Spencer, Laura Zielinski, Ben Caldwell, Adrian Lehmann, and Robert Rand. ViCAR: Visualizing categories with automated rewriting in Coq. *Electronic Proceedings in Theoretical Computer Science*, 429:234–248, September 2025. doi:10.4204/eptcs.429.13.

38 Matthieu Sozeau. A new look at generalized rewriting in type theory. *Journal of Formalized Reasoning*, 2(1):41–62, 2009. doi:10.6092/ISSN.1972-5787/1574.

39 Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Proc. TPHOL*, LNCS, pages 278–293. Springer, 2008. doi:10.1007/978-3-540-71067-7_23.

40 Ross Street. Handbook of algebra. In *Categorical structures*. Elsevier, 1996.

41 Enrico Tassi. Elpi: rule-based meta-language for Rocq. In *Proc. CoqPL*, Denver, United States, January 2025. URL: https://inria.hal.science/hal-04990628.

42 Djordje Čubrić, Peter Dybjer, and Philip Scott. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, pages 153–192, 1998. doi:10.1017/S0960129597002508.