# Machine-Checked Categorical Diagrammatic Reasoning

**Benoît Guillemet**
École normale supérieure Paris-Saclay, France

**Assia Mahboubi**
Nantes Université, École Centrale Nantes, CNRS, INRIA, LS2N, UMR 6004, France
Vrije Universiteit Amsterdam, The Netherlands

**Matthieu Piquerez**
Nantes Université, École Centrale Nantes, CNRS, INRIA, LS2N, UMR 6004, France

**Abstract**

This paper describes a formal proof library, developed using the Coq proof assistant, designed to assist users in writing correct diagrammatic proofs, for 1-categories. This library proposes a deep-embedded, domain-specific formal language, which features dedicated proof commands to automate the synthesis, and the verification, of the technical parts often eluded in the literature.

**2012 ACM Subject Classification** Theory of computation → Logic and verification

**Keywords and phrases** Interactive theorem proving, categories, diagrams, formal proof automation

**Digital Object Identifier** 10.4230/LIPIcs.FSCD.2024.7

**Related Version** *Previous Version*: `https://arxiv.org/abs/2402.14485`
*Previous Version*: `https://inria.hal.science/hal-04471683v4/`

**Supplementary Material**
*Software (Source Code)*: `https://gitlab.inria.fr/mpiquere/coq-diagram-chasing` [10]
archived at `swh:1:dir:a730f6752583959dd173217435f10e0a491ff957`

**Funding** This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 101001995).

**Acknowledgements** The authors would like to thank the anonymous reviewers for their valuable comments and suggestions.

## 1 Introduction

*Abstract nonsense*, a non derogatory expression attributed to Steenrod, usually refers to the incursion of categorical methods for a proof step deemed both technical and little informative, and therefore often succinctly described. *Diagrams* are typically drawn in this case, so as to guide the intuition of the audience, and help visualize the existence of certain morphisms or objects, identities between composition of morphisms, etc.

Formally, a categorical diagram is a functor $F\colon J \to \mathcal{C}$, with $J$ a small category called the *shape* of the diagram [20]. Diagrams are depicted as directed multi-graphs, also called *quivers*, whose vertices are decorated with the objects of $\mathcal{C}$ and whose arrows each represent a certain morphism, between the objects respectively decorating its source and its target. A directed path in the diagram is hence associated with a chain of composable arrows and a diagram *commutes* when all directed paths with same source and target lead to equal compositions. Equalities between compositions of morphisms thus correspond to the commutativity of certain sub-diagrams of a larger diagram. Chasing commutative sub-diagrams in a larger diagram provides an elegant alternative to equational reasoning, when the latter becomes overly technical. Diagrams actually play a central role in category theory, for they provide such an efficient way of delivering convincing enough proofs. Some classical textbooks introduce diagrams as early as in their introduction chapter [17], while others devote an entire section to diagrammatic categorical reasoning [23, Section 1.6] [14, Session 17]. The following diagrammatic proof of Lemma 1 provides a toy illustrative example of this technique.

▶ **Lemma 1.** *Let $\mathcal{C}$ be a category. For any morphism $f$ and $g$ such that $g \circ f \in \mathrm{Hom}(\mathcal{C})$, if $g \circ f$ is a monomorphism, then so is $f$.*

**Proof.** Consider $f\colon A \to B$, $g\colon B \to C$ and $g \circ f\colon A \rightarrowtail C$ morphisms in a category $\mathcal{C}$. Here is a very detailed diagrammatic proof, taking place in the diagram of Figure 1.

$$\begin{tikzcd}
 & & B \arrow[dr, "g"] & \\
D \arrow[urr, bend left, "j"] \arrow[r, bend left, "h"] \arrow[r, bend right, "k"'] \arrow[rrr, bend right, "g\circ j"'] & A \arrow[ur, "f"] \arrow[rr, tail, "g\circ f"] & & C
\end{tikzcd}$$

**Figure 1** Diagrammatic proof that if $g \circ f$ is a monomorphism, then so is $f$.

We need to prove that for any two other morphisms $h, k\colon D \rightrightarrows A$ in $\mathrm{Hom}(\mathcal{C})$ such that $f \circ h = f \circ k$, we have $h = k$, i.e., that the two-arrow diagram $h, k$ commutes. By hypothesis on $h$ and $k$, there is an arrow $j$ such that the triangle diagrams respectively formed by arrows $h, f, j$ and $k, f, j$ both commute. By definition of composition, the triangle diagrams formed by arrows $f, g, g \circ f$ and $j, g, g \circ j$ also commute. As a consequence, both triangle diagrams formed by $h, g \circ f, g \circ j$ and $k, g \circ f, g \circ j$ commute. The conclusion follows because $g \circ f$ is a monomorphism. ◀

*Diagram chasing* actually refers to a central technique to homological algebra [16, 20], operating on diagrams over Abelian categories and used for proving the existence, injectivity, surjectivity of certain morphisms, the exactness of some sequences, etc. Classic examples of proofs by diagram chasing include that of the *five lemma* or of the *snake lemma* [16]. However, complex diagram chases (see, e.g., [22, p.338]) only remain readable at the price of hiding non-trivial technical arguments and are, in practice, challenging to rigorously verify by hand. Typically, the reader of a diagrammatic proof is asked to solve instances of variable difficulty of a decision problem hereafter referred to as the *commerge* problem: *Given a collection of sub-diagrams of a larger diagram which commute, must the entire diagram commute?* In addition, proofs may resort to non-trivial duality arguments, in which the reader has to believe a certain property about diagrams in any Abelian category remains true after reversing all the involved arrows.

The long-term objective of the present work is thus to build a computer-aided instrument for devising both fluent and reliable categorical diagrammatic reasoning, for 1-categories. The present article describes the implementation of the core of such a tool, as a library for the Coq proof assistant [24]. The design of this library follows two main design principles. First, it aims at being independent from any specific library of formalized category theory or abstract algebra, but rather usable as a helper for any existing one. This independence is achieved by formulating the propositions about diagrams as formulas of a deep-embedded language. Specific libraries then lead to specific interpretation functions, turning deep-embedded formulas into actual statements. Second, it strives to feature enough automation tools for synthesizing the bureaucratic parts of proofs "by abstract nonsense", and formal proofs thereof. The main contributions presented here are thus:

- a formalization of a deep-embedded first-order language for category theory, geared towards diagrammatic proofs, together with a generic formalized definition of categorical diagrams. This formalized material is based on a variant of the paper definitions introduced in a previous work by two of the authors [18];
- automation support for proofs by duality in the corresponding reified proof system, that we compare with another method to deal with duality that was introduced in [18];
- automation support for the *commerge* problem.

The corresponding code is available at the following url [10].

The rest of the article is organized as follows. Section 2 fixes some vocabulary and describes the corresponding formal definitions. Section 3 describes the deep-embedded formalization of the first-order language, and of the related reified proof system. Section 4 explains the algorithms involved in automating commutativity proofs. The formalized version of the proof of Lemma 1 is explained in detail in Section 5. We conclude in Section 6 by discussing related work and a few perspectives.

## 2 Preliminaries

This section fixes some definitions and notations, and introduces their formalized counterpart when relevant. Some of them coincide with the preliminaries of our previous text [18], for the purpose of being self-contained. Their formalized counterpart is novel. Throughout this article, we use the word *category* for 1-categories. By default, we do not display implicit arguments in Coq terms.

In all what follows, $\mathbb{N} := \{0, 1, \dots\}$ refers to the set of non-negative integers, represented in Coq by the type `nat`, from its standard library. We also use the standard polymorphic type `list` for finite sequences, equipped with the library on sequences distributed by the Mathematical Components [19] library. Some names thus slightly differ from those present in the standard library. For instance, the size $|l|$ of a finite sequence $l$, formalized as `l : list T`, is `size l`, instead of the standard `length l`. We document in comments the definitions we use from this library when their names are not self-explanatory. We recall that `b1 && b2` is the standard notation for the (boolean) conjunction of two boolean values `b1 b2 : bool`. If $n \in \mathbb{N}$, then $[n]$ denotes the finite collection $\{0, \dots, n-1\}$, which is implemented by the sequence `iota 0 n : list nat`.

► **Definition 2** (General quiver, dual). *A* general quiver *$\mathcal{Q}$ is a quadruple $(V_\mathcal{Q}, A_\mathcal{Q}, s_\mathcal{Q} \colon A_\mathcal{Q} \to V_\mathcal{Q}, t_\mathcal{Q} \colon A_\mathcal{Q} \to V_\mathcal{Q})$ where $V_\mathcal{Q}$ and $A_\mathcal{Q}$ are two sets. The element of $V_\mathcal{Q}$ are called the* vertices *of $\mathcal{Q}$ and the element of $A_\mathcal{Q}$ are called* arrows*. If $a \in A_\mathcal{Q}$, $s_\mathcal{Q}(a)$ is called the* source *of $a$ and $t_\mathcal{Q}(a)$ is called its* target*. The* dual *of a quiver $\mathcal{Q}$ is the quiver $\mathcal{Q}^\dagger := (V_\mathcal{Q}, A_\mathcal{Q}, t_\mathcal{Q}, s_\mathcal{Q})$, which swaps the source and the target maps of $\mathcal{Q}$.*

From now on, we casually call *quivers* the special case of quivers with $V_\mathcal{Q}$ a finite subset of $\mathbb{N}$ and with a finite number of arrows. The formal definition moreover assumes that vertices are labelled in order:

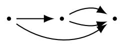
```
(* Data of a quiver *)
Record quiver : Type := quiver_Build {
  quiver_nb_vertex : nat;  (* number of vertices *)
  quiver_arc : list (nat * nat); (* sequence of arrows *) }.

(* Well-formedness condition for quivers :
  all arrows involved in A have a source and target in bound *)
Definition quiver_wf '(quiver_Build n A) : bool :=
  all (fun a => (a.1 < n) && (a.2 < n)) A.
```

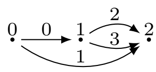
```
(* The dual quiver of a quiver, with same vertices and reversed arrows *)
Definition quiver_dual '(quiver_Build n A) : quiver :=
  quiver_Build n (map (fun a => (a.2,a.1)) A).
```

An arrow of a quiver `q : quiver` is thus given by an element in the sequence `quiver_arc q`, itself a pair of integers giving its source and target respectively. Note that the index in the sequence matters, as the sequence may have duplicate. A formalized quiver is well-formed when the sources and targets of its arrows are in bound.

For the sake of readability, we use drawings to describe some quivers, as for instance:

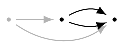


For a quiver $\mathcal{Q}$ denoted by such a drawing, the convention is that $V_{\mathcal{Q}} = [\mathrm{card}(V_{\mathcal{Q}})]$ and $A_{\mathcal{Q}} = [\mathrm{card}(A_{\mathcal{Q}})]$, where $\mathrm{card}(A)$ denotes the cardinal of a finite set $A$. From left to right, the drawn vertices correspond to $0, 1, \ldots, \mathrm{card}(V_{\mathcal{Q}})-1$. Arrows are then numbered by sorting pairs $(s_{\mathcal{Q}}, t_{\mathcal{Q}})$ in increasing lexicographical order, as in:




▶ **Definition 3** (Morphism, embedding). *A* morphism of quivers *$m \colon \mathcal{Q} \to \mathcal{Q}'$, is the data of two maps $m_V \colon V_{\mathcal{Q}} \to V_{\mathcal{Q}'}$ and $m_A \colon A_{\mathcal{Q}} \to A_{\mathcal{Q}'}$ such that $m_V \circ s_{\mathcal{Q}} = s_{\mathcal{Q}'} \circ m_A$ and $m_V \circ t_{\mathcal{Q}} = t_{\mathcal{Q}'} \circ m_A$. Such a morphism is called an* embedding of quivers *if moreover both $m_V$ and $m_A$ are injective. In this case we write $m \colon \mathcal{Q} \hookrightarrow \mathcal{Q}'$.*

We also use drawings to denote embeddings. The black part represents the domain of the morphism, the union of black and gray parts represents its codomain. Here is an example of an embedding of the quiver $\cdot \rightleftarrows \cdot$ into the quiver drawn above.



For the purpose of this work, we actually only need to define formally embedding morphisms, called sub-quivers, which select the relevant vertices and arrows from a quiver:

```
Record subquiver := subquiver_Build {
  subquiver_vertex : list nat; (* labels of the selected vertices *)
  subquiver_arc : list nat; (* indices of the selected arrows *) }.

(* Performs the expected selection of vertices and arrows *)
Definition quiver_restr '(subquiver_Build sV sA) : quiver -> quiver := (...)
```

Here as well, restrictions of quiver only make sense under well-formedness conditions:

```
(* Indices of the arrows to be selected are in bound *)
Definition quiver_restr_A_wf sA '(quiver_Build n A) : bool :=
    all (gtn (size A)) sA.

Definition quiver_restr_V_wf sV '(quiver_Build n A) : bool :=
  uniq sV && (* sV is duplicate-free *)
  all (gtn n) sV && (* all elements of sV are smaller than n *)
  all (fun a => (a.1 \in sV) && (a.2 \in sV)) A. (* any vertex involved in A is in sV*)

(* Well-formed condition on the restriction of a quiver *)
Definition quiver_restr_wf '(subquiver_Build sV sA) Q :=
  quiver_restr_A_wf sA Q && quiver_restr_V_wf sV (quiver_restr_A sA Q).
```

▶ **Definition 4** (Path-quiver). *The* path-quiver of length $k$*, denoted by* $\mathrm{PQ}_k$*, is the quiver with* $k+1$ *vertices and* $k$ *arrows* $([k+1], [k], \mathrm{id}, (i \mapsto i+1))$.

A path-quiver can be drawn as:

$$\bullet \longrightarrow \bullet \longrightarrow \bullet - \cdots \to \bullet$$

with at least one vertex. Such a path-quiver is called *nontrivial* if it has at least two vertices.

If $\mathcal{Q}$ is a general quiver, a morphism of the form $p\colon \mathrm{PQ}_k \to \mathcal{Q}$, for some $k$, is called a *path of* $\mathcal{Q}$ *from* $u$ *to* $v$ *of length* $k$, where $u := p(0)$ and $v := p(k)$. A general quiver is *acyclic* if any path of this quiver is an embedding.

If $P$ is a nontrivial path-quiver, we define $\mathsf{st}_P\colon \bullet\bullet \hookrightarrow P$ to be the embedding mapping the first vertex on the leftmost vertex of $P$ and the second vertex on the rightmost vertex of $P$. Two paths $p_1\colon P_1 \to \mathcal{Q}$, $p_2\colon P_2 \to \mathcal{Q}$ of $\mathcal{Q}$ *have the same extremities* if $p_1 \circ \mathsf{st}_{P_1} = p_2 \circ \mathsf{st}_{P_2}$. We denote by $\mathcal{BP}_{\mathcal{Q}}$ the set of pairs of paths of $\mathcal{Q}$ having the same extremities. Such a pair is called a *bipath*.

Paths in a formalized quivers are defined as a ternary relation between two vertices and a sequence of arrows:

```
(* Operations on lists:
  - (_ == _) is a generic boolean comparison test, in this case for lists of integers
  - rcons l x  is the list l followed by x
  - unzip[1 | 2] l is the list of fst (resp snd) elements of the list of pairs l
  - sub p A is the list of elements of A with index in p, in order *)
Definition path (A : list (nat * nat)) (u : nat) (p : list nat) (v: nat) : bool :=
  (* all elements in p are in bound *)
  all (gtn (size A)) p &&
  (* p selects in A a list of adjacent arrows from u to v *)
  (u :: unzip2 (sub p A) == rcons (unzip1 (sub p A)) v).
```

A path $p$ from a vertex $u$ to a vertex $v$ can thus be concatenated to a path $q$ from vertex $v$ to a vertex $w$: when endpoints are obvious, we just write $p{\cdot}q$ the resulting path from $u$ to $w$. We sometimes abuse notations and write $e{\cdot}p$ and $p{\cdot}e$ when one of the paths contains a single arrow $e$.

We now introduce a special case of binary relation on paths with same extremities in a quiver, called *path relations*. A path relation is an equivalence relation induced from a congruence on the corresponding free category to the quiver. Conversely, in a small category, the composition axiom induces a path relation on the underlying quiver. The formalized definition of path relations is actually independent from that of quiver. A path relation is just a family of equivalence relations on sequences of integers (the paths), indexed by pairs of integers (the endpoints), that are compatible with the concatenation of paths:

```
Record path_relation := {
  pi_r :> forall u v : nat, relation (list nat) ;
  pi_equiv : forall u v, equivalence _ (pi_r u v) ;
  pi_cat_stable : forall u v w p p' q q',
      pi_r u v p p' -> pi_r v w q q' -> pi_r u w (p ++ q) (p' ++ q') }.
```

Given a quiver and a path relation `r : path_relation`, `(pi_r r u v)` is expected to be a relation on the paths from vertex `u` to vertex `v`. Note that the path relation induced by a certain category on its underlying quiver only induces a partial collection of equivalence relations, indexed by the pairs of vertices in this quiver. The full relation on sequences, which relates any two sequences, can be used to complete this collection, so as to define a term of type `forall u v : nat, relation (list nat)`, and to avoid the need for otherwise cumbersome dependent types.

```
Inductive term :=
    | Var of nat (* variable, named with an integer, $k denotes term (Var k) *)
    | Restr of subquiver & term. (* the 'restr' symbol *)

Inductive formula :=
    | Forall of quiver & formula
    | Exists of quiver & formula
    | Imply of formula & formula (* Denoted with infix symbol -=> *)
    | And of formula & formula
    | FTrue (* Top atom *)
    | Commute of term (* the 'commute' predicate symbol *)
    | EqD of term & term.  (* the equality predicate symbol*)
```

**Listing 1** Terms and formulas.

# 3 A two-level approach

## 3.1 Formulas and diagrams

Paraphrasing Mac Lane [17], many properties of category theory can be "unified and simplified by a presentation with diagrams of arrows". Categorical diagrammatic reasoning consists in transforming a proof of category theory into a proof about some quivers, decorated with the data of a certain category. Actually, once the appropriate quivers are drawn, the data themselves can be forgotten, but for the induced path relation, which is the only relevant information for a diagrammatic proof. In [18], we proposed a multi-sorted first-order language for category theory, geared towards diagrammatic reasoning: following the structure of a formula in this language constructs the quivers associated with the corresponding statement of category theory, encoded in the sorts of the variables. We recall the definition of this language:

▶ **Definition 5.** *We define a many-sorted signature $\Sigma$ with sorts the collection of finite acyclic quivers. Signature $\Sigma$ has one function symbol* $\mathrm{restr}_{m\colon \mathcal{Q}'\hookrightarrow\mathcal{Q}}$*, of arity $\mathcal{Q} \to \mathcal{Q}'$, per each* injective *quiver morphism $m\colon \mathcal{Q}' \hookrightarrow \mathcal{Q}$ between two quivers $\mathcal{Q}$ and $\mathcal{Q}'$, and one predicate symbol* $\mathrm{commute}_{\mathcal{Q}}$*, on sort $\mathcal{Q}$, for each finite acyclic quiver $\mathcal{Q}$.*

▶ **Example 6.** Writing the sorts of quantified variables as a subscript of the quantifier, here is for instance a predicate of arity $\bullet\!\to\!\bullet \times \bullet\!\to\!\bullet \times \bullet\!\to\!\bullet$ describing composite of arrows:

$$\mathrm{Comp}(x,y,z)\colon \quad \exists\, w,\quad \mathrm{restr}(w) \approx x \;\wedge\; \mathrm{restr}(w) \approx y \;\wedge\; \mathrm{restr}(w) \approx z \;\wedge\; \mathrm{commute}(w)$$

▶ **Example 7.** Here is a predicate of arity $\bullet\!\to\!\bullet$ describing monomorphisms:

$$\mathrm{Mono}(x)\colon \quad \forall\, w,\quad \mathrm{restr}(w) \approx x \;\Rightarrow\; \mathrm{commute}(\mathrm{restr}(w)) \;\Rightarrow\; \mathrm{commute}(\,\mathrm{restr}(w)) \;\Rightarrow\; \mathrm{commute}(\mathrm{restr}(w))$$

Listing 1 is the formalized counterpart of Definition 5. A term `t : term` is thus either a variable `Var n`, named with a natural number `n`, or of the form `Restr m t` for `t` a term and `m` a sub-quiver, seen as a morphism of quivers. Observe that the source quiver of this morphism is left undefined – it only becomes explicit when the term is evaluated. Type `formula` defines a first-order logic whose atoms stand for equality, commutativity or true. Quantifiers bind de Bruijn indexes and are annotated with a quiver, the sort of the bound variable. Note that theory $\Sigma$ enforces the use of acyclic quivers. Computing the sort of a term thus requires first annotating each of its variables with a sort. The sort of a term `Restr m t` is the quiver obtained by restricting the sort of term `t` using `m`.

Given a list `l` of quivers, providing a sort to each of its variable, a term `t : term` is well-formed in this context, written `term_wf l t`, if `l` is long enough and if every subterm of `t` has a well-formed sort. In a closed formula, the sort of a variable is read on the corresponding quantifier. More generally, a formula `f : formula` is well-formed in a context `l`, written `formula_wf l f`, if `l` is long enough to provide a sort to each free variable in `f` and if every term appearing in `f` has a well formed sort.

Here is the corresponding formalized predicate to Example 7.

```
Definition monoQ := quiver_Build 3 [:: (0,1);(0,1);(0,2);(1,2)]. (* This is *)
Definition mapQ := quiver_Build 2 [:: (0,1)]. (* This is •→• *)

(* Lambda_arc constructs a predicate from a sequence of quivers and a formula, the first
argument is the arity, providing sorts for the free variables of the second, in order.*)
Definition monoF : predicate :=
  Lambda_arc [:: mapQ] (* the one-element arity sequence *)
  (Forall monoQ (
    EqD (Restr {sA [:: 3]} $0) $1
    -=> Commute (Restr {sA [:: 0 ; 2 ; 3]} $0)
    -=> Commute (Restr {sA [:: 1 ; 2 ; 3]} $0)
    -=> Commute (Restr {sA [:: 0 ; 1]} $0))) .
```

The interpretation of a term `f : formula` as a Coq statement, in sort `Prop`, is relative to a formal *diagram*, which is by definition an instance of the following structure `diagram_type`:

```
Record diagram_package (diagram : Type) := diagram_Pack {
  diagram_to_quiver : diagram -> quiver; (* underlying quiver of a diagram *)
  diagram_restr : subquiver -> diagram -> diagram; (* restriction *)
  eqD : equivalence diagram; (* setoid relation on type diagram *)
  eq_comp : diagram -> path_relation; (* path relation *) }.

Structure diagram_type := diagram_type_Build {
  diagram_sort :> Type; (* a diagram_type coerces to its carrier type *)
  diagram_to_package :> diagram_package diagram_sort; }.
```

A diagram is thus a term in the carrier type of an instance of structure `diagram_type`, which can be seen as a model of theory $\Sigma$. A diagram commutes when the associated path relation is *full*, i.e., any two paths in the underlying quiver with same source and target are related:

```
Definition commute (d : diagram_type) (D : d) :=
  path_total (diagram_to_quiver D) (eq_comp D).
```

Formalized diagrams can be defined from a formalization of categories, but not only. For instance, one can define an instance of `diagram_type` with the following carrier type:

```
Record zmod_diagram : Type := ZModDiagram {
  zmob : nat -> zmodType; (* labels of vertices *)
  zmmap : forall u v : nat, nat -> {additive zmob u -> zmob v}; (* labels of arrows *)
  zmdiag_to_quiver : quiver (* underlying quiver *)}.
```

where `zmodType` is a structure for Abelian groups in the Mathematical Components library, and `{additive A -> B}` is the type of morphisms between two Abelian groups `A` and `B`. Labelling functions `zmob` and `zmap`, respectively for vertices and for arrows, are total and thus require a default values for irrelevant arguments, albeit an arbitrary one. We can now explain how to turn a term `f : formula` into a Coq statement, given a sequence of diagrams:

```
Fixpoint formula_eval (d : diagram_type) (stack : list d) (f : formula) : Prop :=
  match f with
  | Forall Q f => forallD D :: diagram_on Q, formula_eval d (D :: stack) f
  | Exists Q f => existsD D :: diagram_on Q, formula_eval d (D :: stack) f
  | Imply f1 f2 => formula_eval d stack f1 -> formula_eval d stack f2
  | And f1 f2 => formula_eval d stack f1 /\ formula_eval d stack f2
  | FTrue => True
  | Commute t => if term_oeval stack t is Some D then commute D else False
  | EqD t1 t2 =>
    match term_oeval stack t1, term_oeval stack t2 with
    | Some DG1, Some DG2 => eqD DG1 DG2
    | _, _ => False
    end
  end.
```

where the notations `forallD D :: diagram_on Q, P` and `existsD D :: diagram_on Q, P` bind variable `D` in `P`, so as to quantify `P` over diagrams `D : d` with underlying quiver `diagram_to_quiver D` equal to `Q`. The evaluation `term_oeval stack t : option d` of a term `t` in context `stack` defaults to `None` when the context `stack` is too small and otherwise computes, when possible, the prescribed restriction of the diagrams. Observe that the type annotation `stack : list d` actually hides a coercion, and is actually `stack : list (diagram_sort d)`.

## 3.2 Structural duality

As briefly alluded to in the conclusion of our previous article [18], it is possible to prove a duality theorem at the meta-level of the deep-embedded first-order language. The variant presented below is updated to the current, bundled, representation of diagram types, and to a slightly different, albeit equivalent, definition of models.[1] For any formula `f : formula`, we define its dual formula `formula_dual f` by structural induction on its argument `f`, dualizing all the quivers involved in `f`. For the sake of readability, the code uses a few notations and coercions:

```
(* We fix a type for diagrams until the end of the section. *)
Variable d : Type.

(* A few local notations to ease reading *)
Local Notation model := diagram_package.
Local Notation model_dual := dual_diagram_Pack.

(* Coercion "conflating" a model with carrier type d and its associated
diagram_package. *)
Local Coercion diagram_type_of := (@diagram_type_Build d).
```

Any model of diagrams `M : model d` has a dual `model_dual M : model d`, obtained from `M` by keeping the same data, but dualizing its quiver and reversing its path relation. We can now prove the property `formula_eval_duality`, stated in Listing 2, characterizing the evaluation of this dual formula in the dual of a diagram. Observe that type `d` is the common carrier type of `M` and `model_dual M`, which allows for the same context `ctx` to be used in both sides of the equivalence.

As a corollary, if a formula holds for all diagrams in a certain diagram type, then so does its dual. The variant `duality_theorem_with_theory` is equally direct but slightly more interesting, as dependent type `P` shall be used to describe a specific class of models, e.g., models of a given theory, provided that their description is “auto-dual”, cf. Listing 2.

[1] Both files, the one attached to [18], and the one updated to fit the new definition of models, can be found in the folder `duality_theorem`.

```
Theorem formula_eval_duality (M : model d) (ctx : list d) (f : formula) :
  @formula_eval (model_dual M) ctx (formula_dual f) <-> @formula_eval M ctx f.

(* If a formula is valid in every model, then so is the dual formula *)
Corollary duality_theorem (ctx : list d) (f : formula) :
  (forall M : model d, @formula_eval M ctx f) ->
  forall M : model d, @formula_eval M ctx (formula_dual f).

(* Relativized variant to a specific class P of models *)
Corollary duality_theorem_with_theory (ctx : list d) (f : formula) (P : model d -> Prop) :
  (forall M, P M -> P (model_dual M)) -> (forall M, P M -> @formula_eval M ctx f)
  -> forall M, P M -> @formula_eval M ctx (formula_dual f).
```

**Listing 2** Duality theorems.

## 3.3 Proofs

The deep-embedded level also features a data-structure `valid_proof` for (deep-embedded) proofs of deep-embedded formulas, and implements a checker `check_proof` for these proofs. A correctness theorem ensures that for any well-formed deep-embedded formula `f : formula`, a positive answer of the proof checker entails the provability of the interpretation of `f` in any model `d`:

```
Theorem check_proof_valid (d : diagram_type) (f : formula) (pf : valid_proof) :
  formula_wf [::] f -> check_proof f pf = true -> formula_eval d [::] f.
```

The deep-embedded level also features a data-structure `sequent`, used to reify a proof in progress, and a type `tactic` for actions making progress in a proof:

```
Inductive sequent := sequent_Build {
  context : list quiver;
  premises : list formula;
  goal : formula; }.

Definition tactic := sequent -> option sequent.
```

Note that there is only one goal attached to a sequent, as formulas are disjunction-free. To a sequent with context `[:: Q_1 ; ... ; Q_n ]`, premises `[:: H_1 ; ... ; H_m ]` and goal `G`, one can associate the following term in type `formula`:

```
Forall Q_1 ( ... ( Forall Q_n (
  Imply   (And H_1 ( ... (And H_m Ftrue) ... ))   G )) ... )
```

The sequent is well-formed if the corresponding formula is a well-formed formula. In the other direction, to any `f : formula`, one can associate the sequent with goal `f` and with empty context and no premise. A tactic $\tau$ is *valid* when for any well-formed sequent $s$, if $\tau\, s$ is some $s'$, then $s'$ is also well-formed and the evaluation of $s'$ implies the evaluation of $s$. In this context, a *valid proof* is just a list of valid tactics.

To check that a valid proof actually provides a proof of some given formula $f$, one can perform the following steps. First, compute the sequent associated to the formula. Then, for each tactic in the proof, apply the tactic. If at some point the tactic returns `None` the proof is not correct. Otherwise, check that the goal of the final sequent is `FTrue`. In this case, the evaluation of $f$ is valid. In the implementation, the verification is performed by the function `check_proof`, and the conclusion is proven in Theorem `check_proof_valid`.

The next step is to implement a set of *useful* tactics, and to prove that they are valid. Note that in this text, tactics refer to commands operating on deep-embedded proofs, and not on actual Coq goals. Basic tactics like introduction and elimination rules are available, but also for more involved tactics like the `Rewrite` tactic, or the `Comauto` tactic, for automating the proof of commerge problems. Starting from the rules of the proof system, more complex tactics provide relevant combinations of the existing building blocks, so as to considerably reduce the size of the proofs. Note that, in theory, the validity of certain tactics may depend on assumptions from the model used for interpreting formulas.

Here is for instance the statement of the formula corresponding to Lemma 1:

```
Definition compQ := quiver_Build 3 [:: (0,1);(0,2);(1,2)]. (* This is .*)

(* when the quiver of a formula have no isolated vertex, formula_fill_vertices
allows for a shorter description of sub-quivers, only by the arcs they select. *)
Definition mono_monomPF : formula :=
formula_fill_vertices [::] (
Forall compQ
  (Commute $0 -=> monoF App (Restr {sA [:: 1]} $0) -=> monoF App (Restr {sA [:: 0]} $0))).
```

The reified proof of this statement is detaild in Section 5.

## 3.4 Duality for reified proofs

In fact, duality arguments are currently implemented by instrumenting proofs so as to check that they are amenable to duality arguments, which is more convenient in practice than the structural argument described in Section 3.2. Indeed, this avoids the need to prove that theories are self-dual. We thus gather a tactic $\tau$ and its dual tactic $\tau^*$ such that for any sequent $s$, the dual of $\tau s$ is equal to $\tau^*$ applied to the dual of $s$. Very often, a tactic and its dual are just identical. A second theorem `duality_theorem`, not to be confused with that of Section 3.2, ensures that, indeed, if every tactic of a proof of some formula comes with such a dual tactic, then the proof obtained by taking the dual tactics will be a proof of the dual formula.[2]

```
(* Biproofs are pairs of proofs of same size *)
Structure biproof := biproof_Build {
  biproof_primal : proof;
  biproof_dual : proof;
  biproof_eq_size : size biproof_primal == size biproof_dual; }.

Theorem duality_theorem f (bpf : biproof) :
(* assuming that the tactics in the pair of proofs bpf are pairwise dual *)
  biproofD bpf ->
(* then the primal proof proves a formula iff the dual proof proves its dual *)
  check_proof (formula_dual f) (biproof_dual bpf) =
  check_proof f (biproof_primal bpf).
```

Each of the tactics currently implemented has a dual tactic. A duality argument can for instance be used to prove the dual statement to Lemma 1, and we provide the corresponding deep-embedded formula.[3]

▶ **Lemma 8.** *Let $\mathcal{C}$ be a category. For any morphism $f$ and $g$ such that $g \circ f \in \mathrm{Hom}(\mathcal{C})$, if $g \circ f$ is an epimorphism, then so is $g$.*

**Proof.** From Lemma 1, by duality. ◀

[2] See file `diagram_chasing/FanL.v`
[3] See file `tests_and_examples/mono_monom.v`.

```
Definition epiF :=
  Lambda_arc [:: mapQD]
  (Forall monoQD (
    EqD (Restr {sA [:: 3]} $0) $1
    -=> Commute (Restr {sA [:: 0 ; 2 ; 3]} $0)
    -=> Commute (Restr {sA [:: 1 ; 2 ; 3]} $0)
    -=> Commute (Restr {sA [:: 0 ; 1]} $0))).

Definition epi_mepiPF : formula :=
formula_fill_vertices [::] (
Forall compQD (
  Commute $0 -=> epiF App (Restr {sA [:: 1]} $0) -=> epiF App (Restr {sA [:: 0]} $0))).
```

which is similar to term `monoF` and formula `mono_monomPF` except that quivers `mapQ`, `monoQ` and `compQ` have been dualized, respectively into `mapQD`, `monoQD` and `compQD`. Observe that instead of being explicitly written, definitions `epiF` and `epi_mepiPF` can also be *computed*, respectively from `monoF` and `mono_monomPF`, by applying the function `dual_formula`. The reified proof of `epi_mepiPF` takes a single tactic, which dualizes the proof of the statement on monomorphisms.

## 4 Automating commutativity proofs

In order to apply most lemmas obtained by diagram chasing, one first has to prove that a certain diagram is commutative. For instance, the so-called *five lemma*, which allows to prove that some map is an isomorphism, requires to have a commutative diagram over the quiver of Figure 2. Yet proving that such a diagram is commutative by checking one equality per bipaths in this diagram would be excessively tedious. Commutativity of a larger diagram is typically obtained from the commutativity of certain sub-diagrams, and the proof of this implication is often little detailed, or not at all. For instance, in the case of Figure 2, it actually suffices to check four equalities, one for each sub-square of the quiver. More precisely, each pair of paths going from the top-left corner to the bottom-right corner of a square must correspond to equal morphisms. Once these four equalities has been proven, we infer that each square commutes.[4]

$$\begin{tikzcd} A \arrow[r] \arrow[d] & B \arrow[r] \arrow[d] & C \arrow[r] \arrow[d] & D \arrow[r] \arrow[d] & E \arrow[d] \\ A' \arrow[r] & B' \arrow[r] & C' \arrow[r] & D' \arrow[r] & E' \end{tikzcd}$$

**Figure 2** The five-lemma diagram.

In section 4.1, we describe an algorithm for deciding the commerge problem for diagrams with acyclic underlying quivers, so as for instance to automate the proof that the diagram of Figure 2 commutes as soon as the aforementioned four bipaths commute. In section 4.2, we describe a heuristic for discovering a collection of sub-diagrams whose commutativity entails that of a larger one.

[4] See file `diagram_chasing/Bipath.v` for a formal proof that the commutativity of a square follows from the equality of the mentioned two morphisms.

## 4.1 Decision procedure for the commerge problem

In [18], we provided a pen-and-paper proof of the decidability of the commerge problem for diagrams with acyclic underlying quiver, and the undecidability of its generalization to possibly cyclic underlying quivers. In this section, we describe the more practical algorithm implemented by the tactic `Comauto`. In particular, this implementation comes with a formal proof of correctness, which is the main ingredient in the validity proof of the corresponding tactic.

The algorithm operates on an acyclic quiver $\mathcal{Q}$ and a finite collection $\mathcal{Q}_1, \ldots, \mathcal{Q}_k$ of sub-quivers of $\mathcal{Q}$, representing the commutativity assumptions. Denote by $l$ the union $\bigcup_i \mathcal{BP}_{\mathcal{Q}_i}$, slightly abusing notations by denoting $\mathcal{Q}_i$ the quiver induced by the corresponding sub-quiver. The algortithm checks whether the smallest path relation $c\ell_{\mathcal{Q}}(l)$ induced by $l$ is full, that is $c\ell_{\mathcal{Q}}(l) = \mathcal{BP}_{\mathcal{Q}}$. Without loss of generality, we can assume that the acyclic quiver $\mathcal{Q}$ is topologically sorted in reverse order, that is, that any path in $\mathcal{Q}$ follows a sequence of vertices with decreasing labels. The implementation actually performs a topological sort of the quiver[5] and updates the representation of $c\ell_{\mathcal{Q}}(l)$ accordingly.

For any given vertices $u$ and $v$ in $V_{\mathcal{Q}}$, we introduce $\mathcal{A}_{u,v} \subset A_{\mathcal{Q}}$ the set of incident arrows of $u$ starting a path to $v$ in $\mathcal{Q}$ and $\bar{\mathcal{A}}_{v,u} \subset A_{\mathcal{Q}}$ that of incident arrows of $v$ ending a path from $u$ in $\mathcal{Q}$:

$$\mathcal{A}_{u,v} \triangleq \{e \in A_{\mathcal{Q}} \mid \exists p,\ e{\cdot}p \text{ is a path from } u \text{ to } v\}$$
$$\bar{\mathcal{A}}_{v,u} \triangleq \{e \in A_{\mathcal{Q}} \mid \exists p,\ p{\cdot}e \text{ is a path to } v \text{ from } u\}$$

We define $\mathcal{G}_{u,v}$ the multigraph whose vertices are the elements of $\mathcal{E}_{u,v} \triangleq \mathcal{A}_{u,v} \cup \bar{\mathcal{A}}_{v,u}$. An arc of $\mathcal{G}_{u,v}$ relates vertices $e_1, e_2 \in \mathcal{E}_{u,v}$ when:

- either there is a path in $\mathcal{Q}$ from $u$ to $v$ containing both $e_1$ and $e_2$;
- or there exists $i$ and $p_1, p_2$ paths in $\mathcal{Q}_i$ from $u$ to $v$ such that $e_1$ (resp. $e_2$) appears in $p_1$ (resp. $p_2$).

We moreover introduce the binary relation $R_{u,v}$, on the elements of $\mathcal{E}$: for any $e_1, e_2 \in \mathcal{E}$, $R_{u,v}(e_1, e_2)$ holds if and only if any path $p_1$ from $u$ to $v$ containing $e_1$ is related by $c\ell_{\mathcal{Q}}(l)$ to any path $p_2$ from $u$ to $v$ containing $e_2$. If $R_{u,v}$ is full, then $c\ell_{\mathcal{Q}}(l)$ contains all the pairs of paths from $u$ to $v$.

▶ **Lemma 9.** *For any vertices $u, v \in V_{\mathcal{Q}}$, if $\mathcal{G}_{u,v}$ is connected, then $R_{u,v}$ is full.*

The decision procedure constructs the multigraphs $\mathcal{G}_{u,v}$ for any pair of vertices, and checks that they are all connected. If so, then $c\ell_{\mathcal{Q}}(l)$ contains $\mathcal{BP}_{\mathcal{Q}}$.

**Proof.** We prove Lemma 9 for vertices respectively labelled $u+n$ and $u$, for a vertex $u \in V_{\mathcal{Q}}$, by induction on $n$. When $n = 0$, then the result holds because paths are empty.

We now assume that the result holds for any vertex and any $k < n$, and that $\mathcal{G}_{u+n,u}$ is connected. Observe first that as a consequence of the induction hypothesis, and because the graph is topologically sorted in reverse order, any two paths from $u+n$ to $u$ sharing their initial or their final arrow are in $c\ell_{\mathcal{Q}}(l)$. We now fix $u \in V_{\mathcal{Q}}$ and $e_1, e_2 \in \mathcal{E}_{u+n,u}$ and we need to prove that $R_{u+n,u}(e_1, e_2)$. We proceed by induction on the length of a path in $\mathcal{G}_{u+n,u}$ relating $e_1$ and $e_2$. If this path is empty, then $e_1 = e_2$ is either an element of $\mathcal{A}_{u,v}$ or of $\bar{\mathcal{A}}_{v,u}$ and we can conclude using the initial observation. We now suppose that there is an arc $e \in \mathcal{E}_{u+n,u}$ and a path $t$ in $\mathcal{G}_{u+n,u}$ such that $(e_1, e)$ is an edge of $\mathcal{G}_{u+n,u}$ and $t$ is a path from

[5] See `diagram_chasing/TopologicalSort.v`.

$e$ to $e_2$ in $\mathcal{G}_{u+n,u}$. Let $p_1$ (resp. $p_2$) be a path in $\mathcal{Q}$ containing $e_1$ (resp. $e_2$). If there is a path $p$ in $\mathcal{Q}$ from $u$ to $v$ containing both $e$ and $e_1$ then $p$ is related to $p_1$, by the observation, as the two paths share either their initial or their final arrow. But $p$ is also related to $p_2$ by the (second) induction hypothesis as $p$ contains $e$ and $p_2$ contains $e_2$. The conclusion follows by transitivity of the path relation. Now suppose that $e$ and $e_1$ respectively belong to $q$ and $q_1$, paths from $u+n$ to $u$ in some $\mathcal{Q}_i$. Paths $q_1$ and $q$ are related, by definition of $c\ell_{\mathcal{Q}}(l)$. But $q$ and $p_2$ are also related by the (second) induction hypothesis, as they respectively contain $e$ and $e_2$. The conclusion follows by transitivity. ◀

## 4.2 Finding sufficient commutativity conditions

In fact, one can even use the computer to guess a sufficient list of equalities entailing that a certain diagram commutes, instead of providing it explicitly by hand. In this section, we explain how to do so in practice. The corresponding algorithm has been implemented under the name *comcut*. Although not strictly needed for the purpose of producing formal proofs of commerge problems, its correctness has been proven formally.[6]

Let $\mathcal{Q}$ be a quiver with vertices $V = [n]$, for some $n \in \mathbb{N}$, and arcs $A_{\mathcal{Q}}$. Moreover, we assume that $\mathcal{Q}$ is topologically sorted in a reversed order, i.e., if an arc goes from $u$ to $v$, then $u > v$. From such an input, *comcut* returns a list of bipaths $l \subseteq \mathcal{BP}_{\mathcal{Q}}$ such that $c\ell_{\mathcal{Q}}(l)$, that is the smallest path relation in $\mathcal{Q}$ containing $l$, is equal to $\mathcal{BP}_{\mathcal{Q}}$.

The algorithm works by induction on the size of $\mathcal{Q}$. Let $u_0 := n-1$ be the top vertex. If there is no arc whose source is $u_0$, then we just have to apply *comcut* to $\mathcal{Q}$ deprived from the vertex $u_0$ (note that $u_0$ cannot be a target). Otherwise, let $a_0 \in A$ be an arc with source $u_0$. Denote by $v_0$ the target of $a_0$. Consider the quiver $\mathcal{Q}'$ obtain from $\mathcal{Q}$ by removing the arc $a_0$. Applying the algorithm to $\mathcal{Q}'$, we get a list $l' \subset \mathcal{BP}_{\mathcal{Q}'}$ such that $c\ell_{\mathcal{Q}'}(l') = \mathcal{BP}_{\mathcal{Q}'}$. We complete this list to a list $l$ as follows. Let $\widetilde{W} := \mathrm{acc}(u_0) \cap \mathrm{acc}(v_0)$ be the set of vertices accessible both from $u_0$ and from $v_0$ in $\mathcal{Q}'$. Set

$$W := \{w \in \widetilde{W} \mid \forall w' \in \widetilde{W} \setminus \{w\}, w \notin \mathrm{acc}(w')\}.$$

For each $w \in W$, select a path $p_w$ from $u_0$ to $w$ in $\mathcal{Q}'$ and a path $q_w$ from $v_0$ to $w$ in $\mathcal{Q}'$. Set $l := l' \cup \{(p_w, a_0 \cdot q_w) \mid w \in W\}$.

▶ **Proposition 10.** *The list $l$ constructed above verifies $c\ell_{\mathcal{Q}}(l) = \mathcal{BP}_{\mathcal{Q}}$.*

Before proving the proposition, let us quickly explain how to get an effective implementation from the above description. To be able to compute accessibility and to reconstruct the different paths, one computes a square matrix indexed by vertices whose $(u, v)$ entry is either empty if there is no nontrivial path from $u$ to $v$, or contains an arc $a$ such that there is a path from $u$ to $v$ which starts by $a$. To update such a matrix for the quiver $\mathcal{Q}'$ into a matrix corresponding to the quiver $\mathcal{Q}$, it suffices to set $a_0$ to all the entries of the form $(u_0, v)$ for $v \in \mathrm{acc}(v_0)$. The rest of the algorithm is easy to write down.

**Proof.** Let $p, q$ be paths from $u$ to $v$ in $\mathcal{Q}$. Note that the arc $a$ can only appear as the first element of $p$ and $q$. If $a$ does not appear in $p$ nor in $q$, or if it appears in both, then $(p, q)$ was already in $c\ell_{\mathcal{Q}}(l')$. Hence, up to symmetry, it remains the case where $u = u_0$, $p$ belongs to $\mathcal{Q}'$ and $q = a_0 \cdot q'$ with $q'$ a path of $\mathcal{Q}'$. By definition of $\widetilde{W}$, $v \in \widetilde{W}$. Let $w \in W$ such that $v$ is accessible from $w$. Let $r$ be a path from $w$ to $v$ (cf. Figure 3). Then, $(p, p_w \cdot r)$ and $(q_w \cdot r, q')$

[6] See the folder `comcut`.

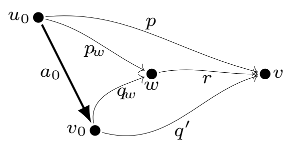



**Figure 3** Decomposition of the relation between two paths thanks to an element $w \in W$.

both belong to $\mathcal{BP}_{\mathcal{Q}'}$, and $(p_w, a_0 \cdot q_w)$ belongs to $l$. Hence we get the following sequence of relations in $c\ell_{\mathcal{Q}}(l)$.

$$p \ \sim \ p_w \cdot r \ \sim \ a_0 \cdot q_w \cdot r \ \sim \ a_0 \cdot q' = q,$$

which proves the proposition. ◀

▶ Remark 11. Note that it may happen that $l$ is not minimal, see Figure 4 for a counterexample.

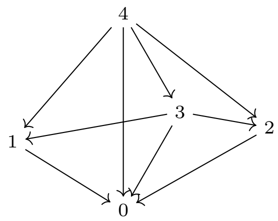



**Figure 4** Counterexample to the minimality of the comcut algorithm. Before adding the arc $(4, 3)$, we need four equalities to ensure the commutativity. Adding $(4, 3)$ forces to add two more relations, but one of the previous relations becomes useless.

## 5 Formal proof of Lemma 1

In this section, we explain in detail the formal proof of Lemma 1 obatined using our framework[7]. This proof actually requires diagrams to be instances of a structure called `category_diagram_type`. This structure describe models which verify some compatibility conditions that we do not precise, and three more axioms. The first axiom is the existence of the composition:

$$\text{CompE:} \qquad \forall_{\bullet \to \bullet} x, y, \quad \text{restr}_{\bullet \to \bullet}(x) \approx \text{restr}_{\bullet \to \bullet}(y) \ \to \ \exists_{\bullet \to \bullet} z, \ \text{Comp}(x, y, z). \tag{1}$$

The last two axioms correspond to the existence and uniqueness of pushout of diagrams. Intuitively, if two diagrams coincide on a sub-quiver, we can "glue" them along this sub-quiver. For an example, see Step 2 in the proof below. See also [18] for more details on the models of these axioms.

The proof is given in Listing 3. In order to improve readability, we replaced quivers and sub-quivers by drawings. The heart of the proof is contained in `mono_monom_pf`. Before describing it in detail, we comment the rest of the code. Tactic `validify` is a custom Coq tactic which triggers Coq's unification hints [1, 24] to infer a valid proof from the proof

[7] See `mono_monom_pf` in file `tests_and_examples/mono_monom.v`.

```
Definition mono_monom_pf : proof := [::
  IntroAll;
  Merge 2;
  ApplyEFT Comp [:: aT (Restr  (Var 2))];
  Merge 7;
  Comauto 4  ;
  Comauto 4  ;
  Comauto 4  ;
  Comauto 4  ;
  ApplyEFT (EqD_refl  ) [:: aT (Restr  (Var 4))];
  ApplyFT [:: aT (Restr  (Var 4)); aP 15; aP 14; aP 13] 1;
  ExactN 16
].

Definition mono_monom_vpf (diagram : category_diagram_type) : valid_proof diagram.
Proof. validify mono_monom_pf. Defined.

Lemma mono_monom (diagram : category_diagram_type) :
  @formula_eval diagram [::] mono_monomPF.
Proof.
  apply: (check_proof_valid (pf := mono_monom_vpf diagram)) ; first by [].
  vm_compute ; done.
Qed.

Definition epi_mepi_vpf (diagram : category_diagram_type) : valid_proof diagram.
Proof. dualify mono_monom_pf. Defined.

Lemma epic_mepic (diagram : category_diagram_type) :
  @formula_eval diagram [::] epi_mepiPF.
Proof.
  apply: (check_proof_valid (pf := epi_mepi_vpf diagram)) ; first by [].
  vm_compute ; done.
Qed.
```

**Listing 3** A formal proof of Lemmas 1 and 8.

`mono_monom_pf`. This can be achieved because every tactic we use in our proof is canonically associated to a valid tactic. In the same way, the `dualify` Coq tactic infers dual valid proofs. Then Lemmas 1 and 8 can be proven by applying `check_proof_valid` with the corresponding valid proof. Observe that we make use of the `vm_compute` reduction machine: the default one used by Coq's type-checker is not efficient enough for executing the commerge algorithm on this nature of problems.

We now detail the list of the tactics that, when applied to the initial sequent with goal `mono_monomPF`, empty context and no premise, return a sequent of goal `FTrue`.

1. First, the tactic `IntroAll` acts as the Coq tactic `intros`: it repeats the following modifications until this is not possible anymore. If the goal is of shape `Forall Q f`, then the new goal will be `f`, while the quiver `Q` is added to the context. If the goal is of shape `f1 ==> f2`, then the new goal will be `f2`, and the formula `f1` is added to the premises. The tactic also destructs conjunctions and existential quantifiers in the new premises.
   Let us detail the sequent obtained after applying this first tactic in our case. The first quiver added to the context is . We denote by $D_0$ the associated variable. Next, the two formulas commute$(D_0)$ and Mono(restr $(D_0)$) are added to the premises and are

denoted by $H_0$ and $H_1$ respectively. At this point, the goal is Mono(restr $(D_0)$), whose full expression is written in Example 7. Then, the quiver is added to the context; let us denote by $D_1$ the associated variable. Finally, the three formulas restr $(D_1) \approx$ restr $(D_0)$, commute(restr $(D_1)$) and commute(restr $(D_1)$) are added to the premises, and the final goal is commute(restr $(D_1)$).

2. From the premise restr $(D_1) \approx$ restr $(D_0)$, we can derive the existence of another variable $D_2$, of sort which verifies

   restr $(D_2) \approx D_1$ and restr $(D_2) \approx D_0$.

   The tactic `Merge` adds the new variable $D_2$ and the two above equalities to the sequent. Its validity is proven using the axiom of existence of pushouts of diagrams that comes from the `category_diagram_type` structure. In addition, the tactic substitutes all the occurences of $D_0$ and $D_1$ in the current sequent (including the goal) to restrictions of $D_2$. Thus, $D_0$ and $D_1$ can be forgotten until the end of the proof.

3. Following the diagrammatic proof of Lemma 1, we need to introduce the composition of the two arrows corresponding to the sub-quiver . It suffices to specialize the axiom of composition with these morphisms. The tactic `ApplyEFT` aims to specialize a formula, here the formula `Comp` corresponding to CompE in (1), to certain arguments, before simplifying the premises (in particular by eliminating conjunctions and existential quantifiers). After applying this tactic, we get a new variable of sort , say $D_3$, and a new premise: restr $(D_2) \approx$ restr $(D_3)$.

4. Once again, thanks to this premise, we can use the `Merge` tactic to merge $D_2$ and $D_3$ and to gather all the data in one variable, say $D_4$, of sort .

**5-8.** At this point of the proof, the goal is: commute(restr $(D_4)$). The premises contain the commutativity of the restrictions of $D_4$ to the following sub-quivers: , , , and . In order to conclude, we have to use the premise expressing that restr $(D_4)$ is a monomorphism, together with the commutativity of restr $(D_4)$ and of restr $(D_4)$. To obtain the latter, we shall use the tactic `Comauto`.

This tactic has two arguments: a variable $D$ of sort $Q$, for some quiver $Q$, and a sub-quiver $Q_0$ of $Q$. Its purpose is to add the premise commute($\mathrm{restr}_{Q_0}(D)$) to the sequent after checking its correctness. It relies on the algorithm described in Section 4.1 which allows us to get the commutativity of a diagram from the commutativity of some of its subdigrams. The tactic looks for every premise of the form commute($\mathrm{restr}_{Q'}(D)$) for some sub-quiver $Q'$ of $Q$ (actually the premise might contain a composition of several restrictions) and deduces from it that $\mathrm{restr}_{Q_0}(D)$ restricted to $Q_0 \cap Q'$ commutes. It then tries to deduce the commutativity of $\mathrm{restr}_{Q_0}(D)$ by applying the decision procedure of the commerge problem.

From this description of the tactic, we see that we cannot directly get the commutativity of restr $(D_4)$ and of restr $(D_4)$. This is why we begin with checking the commutativity of restr $(D_4)$ and of restr $(D_4)$ thanks to two calls of the tactic `Comauto`. Then, two more calls of `Comauto` suffice to get the commutativity of restr $(D_4)$ and of restr $(D_4)$.

**9-12.** The final step of the proof consists in using the premise $H_1$, that is, the fact that restr (restr (restr $(D_4)$))) is a monomorphism. This long list of restrictions has been automatically produced by the `Merge` tactics. Following the notations of Example 7, we will apply this premise to $w \coloneqq \mathrm{restr}\,(D_4)$. The first thing we need to check is

$$\mathrm{restr}\,(\mathrm{restr}\,(D_4)) \approx \mathrm{restr}\,(\mathrm{restr}\,(\mathrm{restr}\,(D_4))).$$

Fortunately, the tactic `ApplyFT` automatically put the formula $H_1$ into a normal form, that is, each term is simplified as a single restriction of a variable to a certain sub-quiver. Hence, after normalization, the above formula just follows from the reflexivity of equality. This formula is added to premises using the tactic `ApplyEFT` on the reflexivity axiom. All the assumptions to complete the proof are in the premises, and the tactic `ApplyFT` specializes the premise $\mathrm{Mono}(\mathrm{restr}\,(D_4))$ to the right arguments. Finally, the tactic `ExactN` ends the proof, acting just as the standard `exact` Coq tactic.

## 6 Conclusion

We have described a first step towards the implementation of a generic library for writing reliable categorical diagrammatic proofs, available online [10]. Such a library can serve two purposes. The first one is to assist mathematician authors in writing reliable proofs, the other is to provide the mandatory infrastructure for expanding the existing corpus of formalized category theory, but also of formalized algebraic topology, and homological algebra. As expressed by the Mathlib community [15], the lack for such a tool is major showstopper for the latter. However, the current state of the present library arguably only provides a low level language for categorical statements and the next steps should enrich the collection of formula combinators, e.g. for limits, pullbacks, etc. as well as the gallery of diagram models, in particular for existing Coq formal libraries of category theory [3, 8, 27, 26]. In parallel, we would develop a interactive graphical interface so as to make the tools more user-friendly.

Independence from any library of category theory is achieved by hosting a dedicated proof system inside that of a proof assistant, Coq in this case, following the classic formalization technique of *deep-embedding* [4]. Dependent tuples allow for a structural duality property for this language. We are not aware of any comparable formal-proof-producing automation tactics for proving the commutativity of diagrams, nor for performing duality arguments. However, some existing libraries of formalized category theory, notably Mathlib [6], for the Lean proof assistant, and Unimath [26], a Coq library for univalent mathematics, provide some tools to ease proofs by diagram chasing, either with brute-force rewrite-based tactics, or with a graphical editor for generating proof scripts [13]. Gross *et al.*'s experience report [9] advocates the use of definitional equality for duality arguments, although without employing deep embeddings or quivers nor discussing diagrammatic reasoning. The other experience reports we are aware of on formalizing category theory, e.g., Carette and Hu's one in Agda [12] or Jacobs and Timany's one in Coq [25], do not include any specific support for diagrammatic proofs either. We refer the interested reader to the later article for a survey of existing libraries of formalized category theory, which remains quite relevant for the purpose of this discussion. A notable more recent endeavor is the Mathlib chapter on category theory. The later serves as a basis for Himmel's formalization of abelian categories in Lean [11], including proofs of the five lemma and of the snake lemma, and proof (semi-)automation tied to this specific formalization. Duality arguments are not addressed. Also in the Mathlib ecosystem, Monbru [21] also discusses algorithmic issues related to the automation of diagram chases, and provides incomplete heuristics for generating automatically proofs expressed in a pseudo-language.

Other computer-aided tools exist for diagrammatic categorical reasoning, with a specific emphasis on the graphical interface. Notably, the accomplished Globular/homotopy.io proof assistant [2, 7] stems from similar concerns about the reliability of diagrammatic reasoning, but for higher category theory. It is geared towards graphical representation rather than formal verification and implements various efficient algorithms for constructing and comparing diagrams in higher categories. Barras and Chabassier have designed a graphical interface for diagrammatic proofs which also provides a graphical interface for generating Coq proof scripts of string diagrams, and visualizing Coq goals as diagrams [5]. But up to our knowledge, this tool does not include any specific automation.

## References


1 Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2009. doi:10.1007/978-3-642-03359-9_8.

2 Krzysztof Bar, Aleks Kissinger, and Jamie Vicary. Globular: an online proof assistant for higher-dimensional rewriting. *Log. Methods Comput. Sci.*, 14(1), 2018. doi:10.23638/LMCS-14(1:8)2018.

3 Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. The HoTT library: a formalization of homotopy type theory in coq. In *CPP*, pages 164–172. ACM, 2017. doi:10.1145/3018610.3018615.

4 Richard J. Boulton, Andrew D. Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In *TPCD*, volume A-10 of *IFIP Transactions*, pages 129–156. North-Holland, 1992.

5 Luc Chabassier and Bruno Barras. A graphical interface for diagrammatic proofs in proof assistants. Contributed talks in the 29th International Conference on Types for Proofs and Programs (TYPES 2023), 2023. URL: https://types2023.webs.upv.es/TYPES2023.pdf.

6 The Mathlib Community. The lean mathematical library. In *CPP*, pages 367–381. ACM, 2020. doi:10.1145/3372885.3373824.

7 Nathan Corbyn, Lukas Heidemann, Nick Hu, Chiara Sarti, Calin Tataru, and Jamie Vicary. homotopy.io: a proof assistant for finitely-presented globular n-categories. *CoRR*, abs/2402.13179, 2024. doi:10.48550/arXiv.2402.13179.

8 Burak Ekici and Cezary Kaliszyk. Mac Lane's Comparison Theorem for the Kleisli Construction Formalized in Coq. *Math. Comput. Sci.*, 14(3):533–549, 2020. doi:10.1007/s11786-020-00450-8.

9 Jason Gross, Adam Chlipala, and David I. Spivak. Experience implementing a performant category-theory library in coq. In *ITP*, volume 8558 of *Lecture Notes in Computer Science*, pages 275–291. Springer, 2014. doi:10.1007/978-3-319-08970-6_18.

10 Benoît Guillemet, Assia Mahboubi, and Matthieu Piquerez. coq-diagram-chasing. https://gitlab.inria.fr/mpiquere/coq-diagram-chasing, 2024.

11 Markus Himmel. Diagram chasing in interactive theorem proving. Bachelorarbeit. Karlsruher Institut für Technologie, 2020. URL: https://pp.ipd.kit.edu/uploads/publikationen/himmel20bachelorarbeit.pdf.

12 Jason Z. S. Hu and Jacques Carette. Formalizing category theory in agda. In *CPP*, pages 327–342. ACM, 2021. doi:10.1145/3437992.3439922.

13 Ambroise Lafont. A categorical diagram editor to help formalising commutation proofs. https://amblafont.github.io/graph-editor/index.html, 2024. Short paper presented in the Journées Francophones des Langages Applicatifs.

14 F. William Lawvere and Stephen H. Schanuel. *Conceptual mathematics. A first introduction to categories.* Cambridge: Cambridge University Press, 2nd ed. edition, 2009.

**15** leanprover-community/mathlib. Condensed mathematics/snake lemma. URL: `https://leanprover-community.github.io/archive/stream/267928-condensed-mathematics/topic/snake.20lemma.html`, 2021.

**16** Saunders Mac Lane. *Homology*. Class. Math. Berlin: Springer-Verlag, reprint of the 3rd corr. print. 1975 edition, 1995.

**17** Saunders Mac Lane. *Categories for the working mathematician*, volume 5 of *Grad. Texts Math.* New York, NY: Springer, 2nd ed edition, 1998.

**18** Assia Mahboubi and Matthieu Piquerez. A first order theory of diagram chasing. In *CSL*, volume 288 of *LIPIcs*, pages 38:1–38:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. `doi:10.4230/LIPIcs.CSL.2024.38`.

**19** Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, January 2021. `doi:10.5281/zenodo.4457887`.

**20** J. Peter May. *A concise course in algebraic topology*. Chicago, IL: University of Chicago Press, 1999.

**21** Yannis Monbru. Towards automatic diagram chasing. M1 report. École Normale Supérieure Paris-Saclay, 2022. URL: `https://github.com/ymonbru/Diagram-chasing/blob/main/MONBRU_Yannis_Rapport.pdf`.

**22** Matthieu Piquerez. *Tropical Hodge theory and applications*. PhD thesis, Institut Polytechnique de Paris, November 2021. URL: `https://theses.hal.science/tel-03499730#`.

**23** Emily Riehl. *Category Theory in Context*. Dover Publications, 2017. URL: `https://math.jhu.edu/~eriehl/context.pdf`.

**24** The Coq Development Team. The coq proof assistant, June 2023. `doi:10.5281/zenodo.8161141`.

**25** Amin Timany and Bart Jacobs. Category theory in coq 8.5. In *FSCD*, volume 52 of *LIPIcs*, pages 30:1–30:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. `doi:10.4230/LIPIcs.FSCD.2016.30`.

**26** Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. Unimath – A computer-checked library of univalent mathematics. available at `http://unimath.org`. `doi:10.5281/zenodo.10849216`.

**27** John Wiegley. Category theory 1.0.0. `https://github.com/jwiegley/category-theory/`, 2022.