



HAL
open science

Machine-Checked Categorical Diagrammatic Reasoning

Benoît Guillemet, Assia Mahboubi, Matthieu Piquerez

► **To cite this version:**

Benoît Guillemet, Assia Mahboubi, Matthieu Piquerez. Machine-Checked Categorical Diagrammatic Reasoning. 9th International Conference on Formal Structures for Computation and Deduction (FSCD 2024), Jul 2024, Tallinn, Estonia. pp.692. hal-04471683v6

HAL Id: hal-04471683

<https://inria.hal.science/hal-04471683v6>

Submitted on 16 Dec 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

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.2024.4

Supplementary Material

Software (Source Code): <https://gitlab.inria.fr/mpiquere/coq-diagram-chasing>

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 $F: J \rightarrow \mathcal{C}$, 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



© Benoît Guillemet, Assia Mahboubi and Matthieu Piquerez;
licensed under Creative Commons License CC-BY 4.0

9th International Conference on Formal Structures for Computation and Deduction (FSCD 2024).

Editor: Jakob Rehof; Article No. 4; pp. 4:1–4:19



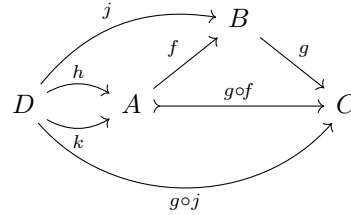
Leibniz International Proceedings in Informatics

LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

43 introduce diagrams as early as in their introduction chapter [17], while others devote an
 44 entire section to diagrammatic categorical reasoning [23, Section 1.6] [14, Session 17]. The
 45 following diagrammatic proof of Lemma 1 provides a toy illustrative example of this technique.

46 ► **Lemma 1.** *Let \mathcal{C} be a category. For any morphism f and g such that $g \circ f \in \text{Hom}(\mathcal{C})$, if
 47 $g \circ f$ is a monomorphism, then so is f .*

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



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

50 We need to prove that for any two other morphisms $h, k: D \rightarrow A$ in $\text{Hom}(\mathcal{C})$ such that
 51 $f \circ h = f \circ k$, we have $h = k$, i.e., that the two-arrow diagram h, k commutes. By hypothesis
 52 on h and k , there is an arrow j such that the triangle diagrams respectively formed by arrows
 53 h, f, j and k, f, j both commute. By definition of composition, the triangle diagrams formed
 54 by arrows $f, g, g \circ f$ and $j, g, g \circ j$ also commute. As a consequence, both triangle diagrams
 55 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
 56 monomorphism. ◀

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

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

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

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

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

94 2 Preliminaries

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

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

110 ► **Definition 2** (General quiver, dual). *A general quiver \mathcal{Q} is a quadruple $(V_{\mathcal{Q}}, A_{\mathcal{Q}}, s_{\mathcal{Q}} : A_{\mathcal{Q}} \rightarrow$
 111 $V_{\mathcal{Q}}, t_{\mathcal{Q}} : A_{\mathcal{Q}} \rightarrow V_{\mathcal{Q}})$ where $V_{\mathcal{Q}}$ and $A_{\mathcal{Q}}$ are two sets. The element of $V_{\mathcal{Q}}$ are called the vertices
 112 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
 113 and $t_{\mathcal{Q}}(a)$ is called its target. The dual of a quiver \mathcal{Q} is the quiver $\mathcal{Q}^{\dagger} := (V_{\mathcal{Q}}, A_{\mathcal{Q}}, t_{\mathcal{Q}}, s_{\mathcal{Q}})$,
 114 which swaps the source and the target maps of \mathcal{Q} .*

115 From now on, we casually call *quivers* the special case of quivers with $V_{\mathcal{Q}}$ a finite subset of
 116 \mathbb{N} and with a finite number of arrows. The formal definition moreover assumes that vertices
 117 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.
```

4:4 Machine-Checked Categorical Diagrammatic Reasoning

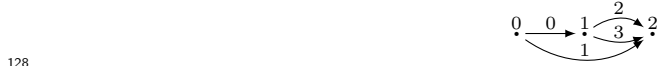
```
(* The dual quiver of a quiver, with same vertices and reversed arrows *)
Definition quiver_dual '(quiver_Build n A) : quiver :=
  quiver_Build n (map (fun a => (a.2,a.1)) A).
```

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

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



124 For a quiver Q denoted by such a drawing, the convention is that $V_Q = [\text{card}(V_Q)]$ and
 125 $A_Q = [\text{card}(A_Q)]$, where $\text{card}(A)$ denotes the cardinal of a finite set A . From left to right,
 126 the drawn vertices correspond to $0, 1, \dots, \text{card}(V_Q) - 1$. Arrows are then numbered by sorting
 127 pairs (s_Q, t_Q) in increasing lexicographical order, as in:



129 ► **Definition 3** (Morphism, embedding). A morphism of quivers $m: Q \rightarrow Q'$, is the data
 130 of two maps $m_V: V_Q \rightarrow V_{Q'}$ and $m_A: A_Q \rightarrow A_{Q'}$ such that $m_V \circ s_Q = s_{Q'} \circ m_A$ and
 131 $m_V \circ t_Q = t_{Q'} \circ m_A$. Such a morphism is called an embedding of quivers if moreover both
 132 m_V and m_A are injective. In this case we write $m: Q \hookrightarrow Q'$.

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



137 For the purpose of this work, we actually only need to define formally embedding
 138 morphisms, called subquivers, 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 := (...)
```

139 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).
```

140 ► **Definition 4** (Path-quiver). *The path-quiver of length k , denoted by PQ_k , is the quiver*
 141 *with $k + 1$ vertices and k arrows ($[k + 1], [k], \text{id}, (i \mapsto i + 1)$).*

142 A path-quiver can be drawn as:

143 $\bullet \longrightarrow \bullet \longrightarrow \bullet \cdots \bullet \longrightarrow \bullet$

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

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

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

153 Paths in a formalized quivers are defined as a ternary relation between two vertices and a
 154 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 (gtm (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).
```

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

159 We now introduce a special case of binary relation on paths with same extremities in
 160 a quiver, called *path relations*. A path relation is an equivalence relation induced from a
 161 congruence on the corresponding free category to the quiver. Conversely, in a small category,
 162 the composition axiom induces a path relation on the underlying quiver. The formalized
 163 definition of path relations is actually independent from that of quiver. A path relation is
 164 just a family of equivalence relations on sequences of integers (the paths), indexed by pairs
 165 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') }.

```

166 Given a quiver and a path relation $r : \text{path_relation}$, $(\text{pi_r } r \text{ } u \text{ } v)$ is expected to be a
 167 relation on the paths from vertex u to vertex v . Note that the path relation induced by
 168 a certain category on its underlying quiver only induces a partial collection of equivalence
 169 relations, indexed by the pairs of vertices in this quiver. The full relation on sequences,
 170 which relates any two sequences, can be used to complete this collection, so as to define a
 171 term of type $\text{forall } u \text{ } v : \text{nat}, \text{relation } (\text{list } \text{nat})$, and to avoid the need for otherwise
 172 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

173 3 A two-level approach

174 3.1 Formulas and diagrams

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

185 ► **Definition 5.** We define a many-sorted signature Σ with sorts the collection of finite
 186 acyclic quivers. Signature Σ has one function symbol $\text{restr}_m: \mathcal{Q}' \rightarrow \mathcal{Q}$, of arity $\mathcal{Q} \rightarrow \mathcal{Q}'$, per
 187 each injective quiver morphism $m: \mathcal{Q}' \hookrightarrow \mathcal{Q}$ between two quivers \mathcal{Q} and \mathcal{Q}' , and one predicate
 188 symbol $\text{commute}_{\mathcal{Q}}$, on sort \mathcal{Q} , for each finite acyclic quiver \mathcal{Q} .

189 ► **Example 6.** Writing the sorts of quantified variables as a subscript of the quantifier, here
 190 is for instance a predicate of arity $\rightarrow \cdot \times \rightarrow \cdot \times \rightarrow \cdot$ describing composite of arrows:

$$\begin{aligned}
 191 \quad \text{Comp}(x, y, z): \quad & \exists_{\rightarrow \cdot \triangleleft \rightarrow \cdot} w, \quad \text{restr}_{\rightarrow \cdot \triangleleft \rightarrow \cdot}(w) \approx x \wedge \text{restr}_{\rightarrow \cdot \triangleleft \rightarrow \cdot}(w) \approx y \\
 192 \quad & \wedge \text{restr}_{\rightarrow \cdot \triangleleft \rightarrow \cdot}(w) \approx z \wedge \text{commute}(w)
 \end{aligned}$$

193 ► **Example 7.** Here is a predicate of arity $\rightarrow \cdot \rightarrow \cdot$ describing monomorphisms:

$$\begin{aligned}
 194 \quad \text{Mono}(x): \quad & \forall_{\rightarrow \cdot \leftarrow \rightarrow \cdot} w, \quad \text{restr}_{\rightarrow \cdot \leftarrow \rightarrow \cdot}(w) \approx x \Rightarrow \text{commute}(\text{restr}_{\rightarrow \cdot \leftarrow \rightarrow \cdot}(w)) \\
 195 \quad & \Rightarrow \text{commute}(\text{restr}_{\rightarrow \cdot \leftarrow \rightarrow \cdot}(w)) \Rightarrow \text{commute}(\text{restr}_{\rightarrow \cdot \leftarrow \rightarrow \cdot}(w))
 \end{aligned}$$

196 Listing 1 is the formalized counterpart of Definition 5. A term $\mathbf{t} : \mathbf{term}$ is thus either a
 197 variable $\mathbf{Var} \ \mathbf{n}$, named with a natural number \mathbf{n} , or of the form $\mathbf{Restr} \ \mathbf{m} \ \mathbf{t}$ for \mathbf{t} a term
 198 and \mathbf{m} a sub-quiver, seen as a morphism of quivers. Observe that the source quiver of this
 199 morphism is left undefined – it only becomes explicit when the term is evaluated. Type
 200 $\mathbf{formula}$ defines a first-order logic whose atoms stand for equality, commutativity or true.
 201 Quantifiers bind de Bruijn indexes and are annotated with a quiver, the sort of the bound

202 variable. Note that theory Σ enforces the use of acyclic quivers. Computing the sort of
 203 a term thus requires first annotating each of its variables with a sort. The sort of a term

204 `Restr m t` is the quiver obtained by restricting the sort of term `t` using `m`.

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

211 Here is the corresponding formalized predicate to Example 7.

```

Definition monoQ := quiver_Build 3 [:: (0,1);(0,1);(0,2);(1,2)]. (* This is  $\begin{matrix} \leftarrow & \rightarrow \\ \leftarrow & \rightarrow \end{matrix}$  *)
Definition mapQ := quiver_Build 2 [:: (0,1)]. (* This is  $\rightarrow$  *)

(* 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))) .
  
```

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

214

```

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; }.
  
```

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

```

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

219 Formalized diagrams can be defined from a formalization of categories, but not only. For
 220 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 *)
  zmmmap : forall u v : nat, nat -> {additive zmob u -> zmob v}; (* labels of arrows *)
  zmdiag_to_quiver : quiver (* underlying quiver *)}.
  
```

221 where `zmodType` is a structure for Abelian groups in the `Mathematical Components` library,
 222 and `{additive A -> B}` is the type of morphisms between two Abelian groups `A` and `B`.
 223 Labelling functions `zmob` and `zmmmap`, respectively for vertices and for arrows, are total and

4:8 Machine-Checked Categorical Diagrammatic Reasoning

224 thus require a default values for irrelevant arguments, albeit an arbitrary one. We can now
225 explain how to turn a term `f : formula` into a Coq statement, given a sequence of diagrams:
226

```
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
  | Implies 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.
```

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

233 3.2 Structural duality

234 As briefly alluded to in the conclusion of our previous article [18], it is possible to prove a
235 duality theorem at the meta-level of the deep-embedded first-order language. The variant
236 presented below is updated to the current, bundled, representation of diagram types, and to
237 a slightly different, albeit equivalent, definition of models.¹ For any formula `f : formula`, we
238 define its dual formula `formula_dual f` by structural induction on its argument `f`, dualizing
239 all the quivers involved in `f`. For the sake of readability, the code uses a few notations and
240 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).
```

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

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`.

247 As a corollary, if a formula holds for all diagrams in a certain diagram type, then so
 248 does its dual. The variant `duality_theorem_with_theory` is equally direct but slightly more
 249 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.

```

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

250

251 3.3 Proofs

252 The deep-embedded level also features a data-structure `valid_proof` for (deep-embedded)
 253 proofs of deep-embedded formulas, and implements a checker `check_proof` for these proofs.
 254 A correctness theorem ensures that for any well-formed deep-embedded formula `f : formula`,
 255 a positive answer of the proof checker entails the provability of the interpretation of `f` in
 256 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.
  
```

257 The deep-embedded level also features a data-structure `sequent`, used to reify a proof in
 258 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.
  
```

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

```

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

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

4:10 Machine-Checked Categorical Diagrammatic Reasoning

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

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

281 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  $\begin{array}{ccc} & \nearrow & \\ & & \searrow \\ & & \end{array}$  *)  
  
(* 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))).
```

282 The reified proof of this statement is detailed in Section 5.

283 3.4 Duality for reified proofs

284 In fact, duality arguments are currently implemented by instrumenting proofs so as to check
285 that they are amenable to duality arguments, which is more convenient in practice than the
286 structural argument described in Section 3.2. Indeed, this avoids the need to prove that
287 theories are self-dual. We thus gather a tactic τ and its dual tactic τ^* such that for any
288 sequent s , the dual of τs is equal to τ^* applied to the dual of s . Very often, a tactic and its
289 dual are just identical. A second theorem `duality_theorem`, not to be confused with that of
290 Section 3.2, ensures that, indeed, if every tactic of a proof of some formula comes with such
291 a dual tactic, then the proof obtained by taking the dual tactics will be a proof of the dual
292 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).
```

² See file `diagram_chasing/FanL.v`

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

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

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

```

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))).
  
```

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

305 4 Automating commutativity proofs

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

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

³ See file `tests_and_examples/mono_monom.v`.

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.

$$\begin{array}{ccccccccc}
A & \longrightarrow & B & \longrightarrow & C & \longrightarrow & D & \longrightarrow & E \\
\downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow \\
A' & \longrightarrow & B' & \longrightarrow & C' & \longrightarrow & D' & \longrightarrow & E'
\end{array}$$

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

322 4.1 Decision procedure for the commerge problem

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

329 The algorithm operates on an acyclic quiver \mathcal{Q} and a finite collection $\mathcal{Q}_1, \dots, \mathcal{Q}_k$ of
330 sub-quivers of \mathcal{Q} , representing the commutativity assumptions. Denote by l the union
331 $\bigcup_i \mathcal{BP}_{\mathcal{Q}_i}$, slightly abusing notations by denoting \mathcal{Q}_i the quiver induced by the corresponding
332 sub-quiver. The algorithm checks whether the smallest path relation $cl_{\mathcal{Q}}(l)$ induced by l is
333 full, that is $cl_{\mathcal{Q}}(l) = \mathcal{BP}_{\mathcal{Q}}$. Without loss of generality, we can assume that the acyclic quiver
334 \mathcal{Q} is topologically sorted in reverse order, that is, that any path in \mathcal{Q} follows a sequence of
335 vertices with decreasing labels. The implementation actually performs a topological sort of
336 the quiver⁵ and updates the representation of $cl_{\mathcal{Q}}(l)$ accordingly.

337 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
338 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
339 from u in \mathcal{Q} :

$$\begin{aligned}
340 \quad \mathcal{A}_{u,v} &\triangleq \{e \in A_{\mathcal{Q}} \mid \exists p, e \cdot p \text{ is a path from } u \text{ to } v\} \\
341 \quad \bar{\mathcal{A}}_{v,u} &\triangleq \{e \in A_{\mathcal{Q}} \mid \exists p, p \cdot e \text{ is a path to } v \text{ from } u\}
\end{aligned}$$

342 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
343 of $\mathcal{G}_{u,v}$ relates vertices $e_1, e_2 \in \mathcal{E}_{u,v}$ when:

- 344 ■ either there is a path in \mathcal{Q} from u to v containing both e_1 and e_2 ;
- 345 ■ 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
346 (resp. p_2).

347 We moreover introduce the binary relation $R_{u,v}$, on the elements of \mathcal{E} : for any $e_1, e_2 \in \mathcal{E}$,
348 $R_{u,v}(e_1, e_2)$ holds if and only if any path p_1 from u to v containing e_1 is related by $cl_{\mathcal{Q}}(l)$ to
349 any path p_2 from u to v containing e_2 . If $R_{u,v}$ is full, then $cl_{\mathcal{Q}}(l)$ contains all the pairs of
350 paths from u to v .

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

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

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

⁵ See `diagram_chasing/TopologicalSort.v`.

356 We now assume that the result holds for any vertex and any $k < n$, and that $\mathcal{G}_{u+n,u}$ is
 357 connected. Observe first that as a consequence of the induction hypothesis, and because the
 358 graph is topologically sorted in reverse order, any two paths from $u+n$ to u sharing their
 359 initial or their final arrow are in $\text{cl}_{\mathcal{Q}}(l)$. We now fix $u \in V_{\mathcal{Q}}$ and $e_1, e_2 \in \mathcal{E}_{u+n,u}$ and we need
 360 to prove that $R_{u+n,u}(e_1, e_2)$. We proceed by induction on the length of a path in $\mathcal{G}_{u+n,u}$
 361 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
 362 $\bar{\mathcal{A}}_{v,u}$ and we can conclude using the initial observation. We now suppose that there is an arc
 363 $e \in \mathcal{E}_{u+n,u}$ 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
 364 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
 365 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,
 366 as the two paths share either their initial or their final arrow. But p is also related to p_2 by
 367 the (second) induction hypothesis as p contains e and p_2 contains e_2 . The conclusion follows
 368 by transitivity of the path relation. Now suppose that e and e_1 respectively belong to q and
 369 q_1 , paths from $u+n$ to u in some \mathcal{Q}_i . Paths q_1 and q are related, by definition of $\text{cl}_{\mathcal{Q}}(l)$. But
 370 q and p_2 are also related by the (second) induction hypothesis, as they respectively contain e
 371 and e_2 . The conclusion follows by transitivity. ◀

372 4.2 Finding sufficient commutativity conditions

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

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

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

$$389 \quad W := \{w \in \widetilde{W} \mid \forall w' \in \widetilde{W} \setminus \{w\}, w \notin \text{acc}(w')\}.$$

390 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
 391 $l := l' \cup \{(p_w, a_0 \cdot q_w) \mid w \in W\}$.

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

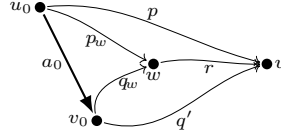
393 Before proving the proposition, let us quickly explain how to get an effective implementa-
 394 tion from the above description. To be able to compute accessibility and to reconstruct the
 395 different paths, one computes a square matrix indexed by vertices whose (u, v) entry is either
 396 empty if there is no nontrivial path from u to v , or contains an arc a such that there is a
 397 path from u to v which starts by a . To update such a matrix for the quiver \mathcal{Q}' into a matrix

⁶ See the folder *comcut*.

4:14 Machine-Checked Categorical Diagrammatic Reasoning

398 corresponding to the quiver \mathcal{Q} , it suffices to set a_0 to all the entries of the form (u_0, v) for
 399 $v \in \text{acc}(v_0)$. The rest of the algorithm is easy to write down.

400 **Proof.** Let p, q be paths from u to v in \mathcal{Q} . Note that the arc a can only appear as the first
 401 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
 402 already in $\text{cl}_{\mathcal{Q}}(l')$. Hence, up to symmetry, it remains the case where $u = u_0$, p belongs to
 403 \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 $(q_w \cdot r, q')$



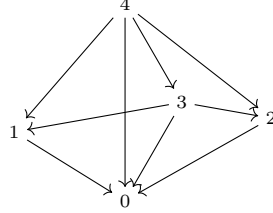
■ **Figure 3** Decomposition of the relation between two paths thanks to an element $w \in W$.

404 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
 405 relations in $\text{cl}_{\mathcal{Q}}(l)$.
 406

$$407 \quad p \sim p_w \cdot r \sim a_0 \cdot q_w \cdot r \sim a_0 \cdot q' = q,$$

408 which proves the proposition. ◀

409 ▶ **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.

410

411 5 Formal proof of Lemma 1









412 In this section, we explain in detail the formal proof of Lemma 1 obtained using our
 413 framework⁷. This proof actually requires diagrams to be instances of a structure called
 414 `category_diagram_type`. This structure describe models which verify some compatibility
 415 conditions that we do not precise, and three more axioms. The first axiom is the existence of
 416 the composition:

$$417 \quad \text{CompE:} \quad \forall \cdot \rightarrow \cdot x, y, \quad \text{restr}_{\cdot \rightarrow \cdot}(x) \approx \text{restr}_{\cdot \rightarrow \cdot}(y) \rightarrow \exists \cdot \rightarrow \cdot z, \text{Comp}(x, y, z). \quad (1)$$

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

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

```

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. validate 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.

422 The proof is given in Listing 3. In order to improve readability, we replaced quivers
 423 and sub-quivers by drawings. The heart of the proof is contained in `mono_monom_pf`. Before
 424 describing it in detail, we comment the rest of the code. Tactic `validate` is a custom Coq
 425 tactic which triggers Coq’s unification hints [2, 24] to infer a valid proof from the proof
 426 `mono_monom_pf`. This can be achieved because every tactic we use in our proof is canonically
 427 associated to a valid tactic. In the same way, the `dualify` Coq tactic infers dual valid proofs.
 428 Then Lemmas 1 and 8 can be proven by applying `check_proof_valid` with the corresponding
 429 valid proof. Observe that we make use of the `vm_compute` reduction machine: the default
 430 one used by Coq’s type-checker is not efficient enough for executing the commerge algorithm
 431 on this nature of problems.

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

434 1. First, the tactic `IntroAll` acts as the Coq tactic `intros`: it repeats the following
 435 modifications until this is not possible anymore. If the goal is of shape `forall Q f`, then
 436 the new goal will be `f`, while the quiver `Q` is added to the context. If the goal is of
 437 shape `f1 ==> f2`, then the new goal will be `f2`, and the formula `f1` is added to the
 438 premises. The tactic also destructs conjunctions and existential quantifiers in the new
 439 premises.

440 Let us detail the sequent obtained after applying this first tactic in our case. The first
 441 quiver added to the context is \triangleleft . We denote by D_0 the associated variable. Next, the
 442 two formulas `commute(D_0)` and `Mono(restr \triangleleft (D_0))` are added to the premises and are
 443 denoted by H_0 and H_1 respectively. At this point, the goal is `Mono(restr \triangleleft (D_0))`, whose
 444 full expression is written in Example 7. Then, the quiver \rightleftarrows is added to the context;
 445 let us denote by D_1 the associated variable. Finally, the three formulas `restr \rightleftarrows (D_1) \approx`
 446 `restr \triangleleft (D_0)`, `commute(restr \rightleftarrows (D_1))` and `commute(restr \rightleftarrows (D_1))` are added to
 447 the premises, and the final goal is `commute(restr \rightleftarrows (D_1))`.

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

$$450 \quad \text{restr}_{\rightleftarrows} (D_2) \approx D_1 \quad \text{and} \quad \text{restr}_{\triangleleft} (D_2) \approx D_0.$$

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

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

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

465 5-8. At this point of the proof, the goal is: `commute(restr \rightleftarrows (D_4))`. The premises contain

466 the commutativity of the restrictions of D_4 to the following sub-quivers: \rightleftarrows , \rightleftarrows
 467 , \rightleftarrows and \rightleftarrows . In order to conclude, we have to use the premise expressing that
 468 `restr \rightleftarrows (D_4)` is a monomorphism, together with the commutativity of `restr \rightleftarrows (D_4)`
 469 and of `restr \rightleftarrows (D_4)`. To obtain the latter, we shall use the tactic `Comauto`.

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

477 the commutativity of $\text{restr}_{Q_0}(D)$ by applying the decision procedure of the `commerge`
 478 problem.

479 From this description of the tactic, we see that we cannot directly get the commutativity
 480 of $\text{restr}_{\triangleleft}(\text{restr}_{\triangleleft}(\text{restr}_{\triangleleft}(D_4)))$ and of $\text{restr}_{\triangleleft}(\text{restr}_{\triangleleft}(\text{restr}_{\triangleleft}(D_4)))$. This is why we begin with checking the
 481 commutativity of $\text{restr}_{\triangleleft}(\text{restr}_{\triangleleft}(D_4))$ and of $\text{restr}_{\triangleleft}(\text{restr}_{\triangleleft}(D_4))$ thanks to two calls of the
 482 tactic `Comauto`. Then, two more calls of `Comauto` suffice to get the commutativity of
 483 $\text{restr}_{\triangleleft}(\text{restr}_{\triangleleft}(D_4))$ and of $\text{restr}_{\triangleleft}(\text{restr}_{\triangleleft}(D_4))$.

484-9-12. The final step of the proof consists in using the premise H_1 , that is, the fact that
 485 $\text{restr}_{\triangleleft}(\text{restr}_{\triangleleft}(\text{restr}_{\triangleleft}(\text{restr}_{\triangleleft}(D_4))))$ is a monomorphism. This long list of restrictions
 486 has been automatically produced by the `Merge` tactics. Following the notations of
 487 Example 7, we will apply this premise to $w := \text{restr}_{\triangleleft}(\text{restr}_{\triangleleft}(\text{restr}_{\triangleleft}(D_4)))$. The first thing we need
 488 to check is

$$489 \quad \text{restr}_{\triangleleft}(\text{restr}_{\triangleleft}(\text{restr}_{\triangleleft}(\text{restr}_{\triangleleft}(D_4)))) \approx \text{restr}_{\triangleleft}(\text{restr}_{\triangleleft}(\text{restr}_{\triangleleft}(\text{restr}_{\triangleleft}(D_4))))$$

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

497 6 Conclusion

498 We have described a first step towards the implementation of a generic library for writing
 499 reliable categorical diagrammatic proofs, available online [10]. Such a library can serve two
 500 purposes. The first one is to assist mathematician authors in writing reliable proofs, the other
 501 is to provide the mandatory infrastructure for expanding the existing corpus of formalized
 502 category theory, but also of formalized algebraic topology, and homological algebra. As
 503 expressed by the `Mathlib` community [15], the lack for such a tool is major showstopper for
 504 the latter. However, the current state of the present library arguably only provides a low
 505 level language for categorical statements and the next steps should enrich the collection of
 506 formula combinators, e.g. for limits, pullbacks, etc. as well as the gallery of diagram models,
 507 in particular for existing Coq formal libraries of category theory [4, 8, 27, 26]. In parallel, we
 508 would develop a interactive graphical interface so as to make the tools more user-friendly.

509 Independence from any library of category theory is achieved by hosting a dedicated proof
 510 system inside that of a proof assistant, Coq in this case, following the classic formalization
 511 technique of *deep-embedding* [5]. Dependent tuples allow for a structural duality property
 512 for this language. We are not aware of any comparable formal-proof-producing automation
 513 tactics for proving the commutativity of diagrams, nor for performing duality arguments.
 514 However, some existing libraries of formalized category theory, notably `Mathlib` [1], for the
 515 `Lean` proof assistant, and `Unimath` [26], a Coq library for univalent mathematics, provide
 516 some tools to ease proofs by diagram chasing, either with brute-force rewrite-based tactics,
 517 or with a graphical editor for generating proof scripts [13]. Gross *et al.*'s experience report [9]
 518 advocates the use of definitional equality for duality arguments, although without employing

519 deep embeddings or quivers nor discussing diagrammatic reasoning. The other experience
 520 reports we are aware of on formalizing category theory, e.g., Carette and Hu’s one in *Agda* [12]
 521 or Jacobs and Timany’s one in *Coq* [25], do not include any specific support for diagrammatic
 522 proofs either. We refer the interested reader to the later article for a survey of existing
 523 libraries of formalized category theory, which remains quite relevant for the purpose of
 524 this discussion. A notable more recent endeavor is the *Mathlib* chapter on category theory.
 525 The later serves as a basis for Himmel’s formalization of abelian categories in *Lean* [11],
 526 including proofs of the five lemma and of the snake lemma, and proof (semi-)automation
 527 tied to this specific formalization. Duality arguments are not addressed. Also in the *Mathlib*
 528 ecosystem, Monbru [21] also discusses algorithmic issues related to the automation of diagram
 529 chases, and provides incomplete heuristics for generating automatically proofs expressed in a
 530 pseudo-language.

531 Other computer-aided tools exist for diagrammatic categorical reasoning, with a specific
 532 emphasis on the graphical interface. Notably, the accomplished *Globular/homotopy.io* proof
 533 assistant [3, 7] stems from similar concerns about the reliability of diagrammatic reasoning,
 534 but for higher category theory. It is geared towards graphical representation rather than
 535 formal verification and implements various efficient algorithms for constructing and comparing
 536 diagrams in higher categories. Barras and Chabassier have designed a graphical interface for
 537 diagrammatic proofs which also provides a graphical interface for generating *Coq* proof scripts
 538 of string diagrams, and visualizing *Coq* goals as diagrams [6]. But up to our knowledge, this
 539 tool does not include any specific automation.

540 ——— References ———

- 541 1 The lean mathematical library. In *CPP*, pages 367–381. ACM, 2020. doi:10.1145/3372885.
 542 3373824.
- 543 2 Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in
 544 unification. In *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 84–98.
 545 Springer, 2009. doi:10.1007/978-3-642-03359-9_8.
- 546 3 Krzysztof Bar, Aleks Kissinger, and Jamie Vicary. Globular: an online proof assistant for higher-
 547 dimensional rewriting. *Log. Methods Comput. Sci.*, 14(1), 2018. doi:10.23638/LMCS-14(1:
 548 8)2018.
- 549 4 Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau,
 550 and Bas Spitters. The HoTT library: a formalization of homotopy type theory in *coq*. In
 551 *CPP*, pages 164–172. ACM, 2017. doi:10.1145/3018610.3018615.
- 552 5 Richard J. Boulton, Andrew D. Gordon, Michael J. C. Gordon, John Harrison, John Herbert,
 553 and John Van Tassel. Experience with embedding hardware description languages in HOL. In
 554 *TPCD*, volume A-10 of *IFIP Transactions*, pages 129–156. North-Holland, 1992.
- 555 6 Luc Chabassier and Bruno Barras. A graphical interface for diagrammatic proofs in proof
 556 assistants. Contributed talks in the 29th International Conference on Types for Proofs and
 557 Programs (TYPES 2023), 2023. <https://types2023.webs.upv.es/TYPES2023.pdf>.
- 558 7 Nathan Corbyn, Lukas Heidemann, Nick Hu, Chiara Sarti, Calin Tataru, and Jamie Vicary. ho-
 559 motopy.io: a proof assistant for finitely-presented globular n-categories. *CoRR*, abs/2402.13179,
 560 2024. doi:10.48550/arXiv.2402.13179.
- 561 8 Burak Ekici and Cezary Kaliszzyk. Mac Lane’s Comparison Theorem for the Kleisli Con-
 562 struction Formalized in *Coq*. *Math. Comput. Sci.*, 14(3):533–549, 2020. doi:10.1007/
 563 s11786-020-00450-8.
- 564 9 Jason Gross, Adam Chlipala, and David I. Spivak. Experience implementing a performant
 565 category-theory library in *coq*. In *ITP*, volume 8558 of *Lecture Notes in Computer Science*,
 566 pages 275–291. Springer, 2014. doi:10.1007/978-3-319-08970-6_18.

- 567 10 Benoît Guillemet, Assia Mahboubi, and Matthieu Piquerez. coq-diagram-chasing. <https://gitlab.inria.fr/mpiquere/coq-diagram-chasing>, 2024.
- 568
- 569 11 Markus Himmel. Diagram chasing in interactive theorem proving. Bachelorarbeit. Karlsruhe Institut für Technologie, 2020. <https://pp.ipd.kit.edu/uploads/publikationen/himmel20bachelorarbeit.pdf>.
- 570
- 571
- 572 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.
- 573
- 574 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.
- 575
- 576
- 577 14 F. William Lawvere and Stephen H. Schanuel. *Conceptual mathematics. A first introduction to categories*. Cambridge: Cambridge University Press, 2nd ed. edition, 2009.
- 578
- 579 15 leanprover-community/mathlib. Condensed mathematics/snake lemma. <https://leanprover-community.github.io/archive/stream/267928-condensed-mathematics/topic/snake.20lemma.html>, 2021.
- 580
- 581
- 582 16 Saunders Mac Lane. *Homology*. Class. Math. Berlin: Springer-Verlag, reprint of the 3rd corr. print. 1975 edition, 1995.
- 583
- 584 17 Saunders Mac Lane. *Categories for the working mathematician.*, volume 5 of *Grad. Texts Math*. New York, NY: Springer, 2nd ed edition, 1998.
- 585
- 586 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.
- 587
- 588
- 589 19 Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, jan 2021. doi:10.5281/zenodo.4457887.
- 590
- 591 20 J. Peter May. *A concise course in algebraic topology*. Chicago, IL: University of Chicago Press, 1999.
- 592
- 593 21 Yannis Monbru. Towards automatic diagram chasing. M1 report. École Normale Supérieure Paris-Saclay, 2022. https://github.com/ymonbru/Diagram-chasing/blob/main/MONBRU_Yannis_Rapport.pdf.
- 594
- 595
- 596 22 Matthieu Piquerez. *Tropical Hodge theory and applications*. PhD thesis, Institut Polytechnique de Paris, nov 2021. URL: <https://theses.hal.science/tel-03499730#>.
- 597
- 598 23 Emily Riehl. *Category Theory in Context*. Dover Publications, 2017. <https://math.jhu.edu/~eriehl/context.pdf>.
- 599
- 600 24 The Coq Development Team. The coq proof assistant, jun 2023. doi:10.5281/zenodo.8161141.
- 601
- 602 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.
- 603
- 604
- 605 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.
- 606
- 607
- 608 27 John Wiegley. Category theory 1.0.0. <https://github.com/jwiegley/category-theory/>, 2022.
- 609