Formalizing adhesive category theory in Rocq

Samuel Arsac, Russell Harmer, Damien Pous

Formalizing adhesive category theory in Rocq

Samuel Arsac[1], Russ Harmer[2], and Damien Pous[2]

[1]ENS Lyon, CNRS, UCBL1, LIP, UMR 5668, 69342, Lyon Cedex 07, France
[2]CNRS, ENS Lyon, UCBL1, LIP, UMR 5668, 69342, Lyon Cedex 07, France

We formalize adhesive categories in Rocq using Hierarchy Builder. This formalization is modular, with adhesive categories being at the top of a hierarchy which contains the weaker variants of adhesive categories and starts at the level of categories. Each level is equipped with several equivalent interfaces to define instances. This formalization comes with a library of lemmas about more basic categorical concepts, such as pullbacks and regular monomorphisms. We provide as instances the category of sets and the category of simple graphs.

1 Introduction

Adhesive categories, as well as the weaker notions of rm-adhesive and rm-quasiadhesive categories, are classes of categories used in categorical graph rewriting [LS05, GL12]. Categorical graph rewriting is a framework for defining rewriting rules on graphs, together with their so-called double pushout operational semantics, using categories to abstract the differences between different concrete graph models. This domain has found various applications in computer science, such as model-based software engineering [HT20]; in mathematical physics, such as the rewriting of string diagrams [BGK$^+$22]; and in modelling chemical and biological systems, where rewriting rules model the interactions between particles or molecules [DHW13, AFMS16].

The various definitions for those categories involve standard notions such as monomorphisms, pullbacks and pushouts; yet, proving the various implications existing between these definitions is sometimes non-trivial. Their formalization can therefore be seen as an interesting test case for the development of general purpose category theory libraries.

As usual for formalization, a challenging task consists in structuring the definitions so that lemmas can be stated and used in their most general form, and that instances can be defined in the most efficient way. To this end, we choose to use Hierarchy Builder[1] [CST20], which gives a way to define a hierarchy of structures, allowing each structure to inherit properties from multiple other structures. Hierarchy Builder internally translates these high level definitions into appropriate records, canonical structures and coercions declarations [GGMR09], in such a way that the hierarchy can be defined incrementally. In particular, an important feature is that Hierarchy Builder allows for the addition of new structures in the middle of the hierarchy without breaking existing code.

Our formalization currently includes (i) the definitions needed for adhesive categories and various lemmas about them; (ii) the definitions of adhesive, rm-adhesive and rm-quasiadhesive categories; (iii) two important theorems providing non-trivial characterisations of those structures; and (iv) two prototypical instances: sets and simple graphs.

[1]`https://github.com/math-comp/hierarchy-builder`

The library is available on Gitlab[2], with a readme file linking the theorems in this paper to the relevant parts of the code. The formalized proofs closely follow those from the literature, some of which are several pages long on paper. Trying to stay as high-level as possible, we present in this article the solutions we have implemented to solve the following problems.

- When there is an object satisfying a certain property, it is not always clear to what extent it should be bundled. That is, should we use a `Record` containing the object and the proof of the property; or should we keep the object and the proof of the property separately? For example, do we wish to state that we have an isomorphism between two objects, or that two given morphisms form an isomorphism. This question is well-known when dealing with algebraic hierarchies [GGMR09]; for us, it also turns out to be important for basic categorical concepts, such as isomorphisms or pullbacks.
- Several definitions we use have duals which we also use: pullbacks and pushouts, or monomorphisms and epimorphisms, for instance. We need appropriate mechanisms in order to avoid code duplication when dealing with such concepts.
- There are several equivalent definitions for adhesive and rm-adhesive categories. This leaves several choices for the definitions to use in the hierarchy, and we must choose carefully in order to reduce the burden on the user-side, and to adhere to the non-forgetful inheritance policy of Hierarchy Builder [CST20].

We discuss the first two points in Section 2, and the third in Section 3.

2 Elementary structures

2.1 Categories

We use the definition of categories from the examples found in the repository of Hierarchy Builder. They are defined in three steps:

- quivers, with only objects and arrows;
- precategories, as quivers with identity arrows and composition;
- categories, as precategories with equalities for identity and associativity of composition.

While our development does not require such a level of detail in the definition of categories, this is good example of the way we will use a hierarchy.

For example, terminal objects can be defined in quivers, since they are just objects with a unique arrow from any other object; so there is no need for composition or identity. This means that any other structure built on top of quivers will be able to use this definition, even if it is not a category. As for the proof that a terminal object is unique up to a unique isomorphism, it is written in the setting of precategories since we need composition and identity for isomorphisms but there is no need for associativity or the fact that identity is neutral for composition.

2.2 Isomorphisms

When working with isomorphisms, we have to choose how much we bundle the two morphisms and the proofs that they cancel each other. There are three reasonable bundling levels:

- Only bundling the equalities, keeping both morphisms as arguments:

```
Definition IsIso2 {C: precat} [X Y: C] (f: X ⇝ Y) (g: Y ⇝ X) :=
    f ∘ g = idmap ∧ g ∘ f = idmap.
```

[2] `https://gitlab.com/SamuelArsac/graph-rewriting`

Where `idmap` is the identity morphism on an object that is left implicit.

This is useful when the two morphisms appear independently, before we even know they form an isomorphism. We also rely on this definition in the next two bundling levels.

- Bundling the equalities and the inverse, keeping the first morphism as argument. This corresponds to stating that a particular morphism is an isomorphism.

```
Definition IsIso {C: precat} [X Y: C] (f: X ⇝ Y) :=
  {g: Y ⇝ X | IsIso2 f g}.
```

This is useful when we do not care much about the inverse of the isomorphism. However, it requires an inference mechanism: even once we have proved that the composition of two isomorphisms is an isomorphism, we need to use the corresponding lemma over and over again. Another inconvenience is that the proof context contains two entries to describe a single object: the morphism and the proof that it is an isomorphism.

- Bundling everything together (except the source and target objects):

```
Definition Iso {C: precat} (X Y: C) :=
  {f: X ⇝ Y & IsIso f}.
```

With this representation, all the information is stored in a single object, and we just have to use projections to extract the various components: given an isomorphism `i: Iso A B`, $\texttt{i}^1$ gives the morphism from `A` to `B`, and $\texttt{i}^{-1}$ gives the other from `B` to `A`.

Another important advantage of this representation is that we can use it to define the subcategory of isomorphisms, which makes bundled isomorphisms themselves morphisms. In particular, a composition of isomorphisms remains an isomorphism, simply by type-checking. Having the category of isomorphisms is also useful when stating unicity results up to unique isomorphism: like in John Wiegley's library, we have defined a single notation and a few tools to state and prove that a morphism is the only one satisfying a given property; these tools can readily be used for isomorphisms once they form a category.

We try to use the third representation as much as possible. Note that there are nevertheless some cases where we need the second, for instance when stating that the pullback along an isomorphism yields an isomorphism. In that case, the isomorphism taken as input can be fully bundled while the output one cannot.

To bridge the gap between the two representations, we have defined a tactic `promote_iso f`, which checks if there is an hypothesis of type `IsIso f` and then turns `f` into a bundled isomorphism, making the relevant changes to the other hypotheses.

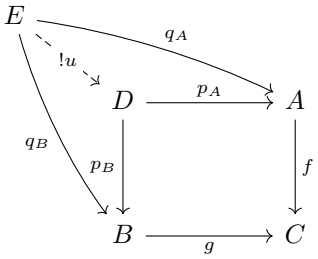
Our library contains many lemmas related to isomorphisms, since it is often possible to introduce, remove or move around isomorphisms. Unfortunately, their use still induces an increased complexity compared to paper proofs where isomorphisms are often handled implicitly.

2.3 Pullbacks

Similarly to isomorphisms, there are several possible bundling levels for pullbacks. We start by recalling the definition of a pullback.

Definition 1 (Pullback)**.** Given a cospan (a pair of arrows that share the same target object) $(f : A \rightsquigarrow C, g : B \rightsquigarrow C)$, a pullback of this cospan is a span $(p_A : D \rightsquigarrow A, p_B : D \rightsquigarrow B)$ commuting with the cospan and such that for any other span $(q_A : E \rightsquigarrow A, q_B : E \rightsquigarrow B)$

commuting with the cospan, there is a unique morphism $u : E \rightsquigarrow D$ making the two triangles commute.



In John Wiegley's library, pullbacks are defined as a `Record` with the pullback object and the two projections p_A and p_B, along with the universal property of pullbacks described in the previous definition. This definition is convenient when stating that a cospan *has* a pullback; for example, when stating that a category has all pullbacks. Is is however very inconvenient when stating that four morphisms form a *pullback square*, since the two projections are tied to the pullback proof, while they could be the projections of other pullbacks for example, which would necessitate tedious manipulations of equalities.

```
Record Pullback {C : cat} {A B C: C} (f: A ⇝ C) (g: B ⇝ C) := {
  D: C;
  pA: D ⇝ A;
  pB: D ⇝ B;
  pullback_commutes: f ∘ pA = g ∘ pB;
  ump_pullbacks: ∀ E (qA : E ⇝ A) (qB : E ⇝ B), f ∘ qA = g ∘ qB
    → ∃! u: E ⇝ D, pA ∘ u = qA ∧ pB ∘ u = qB
}.
```

Note: here ∃! is the custom construction for unique morphisms we alluded to in the previous section about isomorphisms (which is in `Type`).

We can unbundle the previous definition, leaving only the properties of a pullback square as fields. This is convenient when stating that a square of morphisms *is* a pullback, which is what happens most of the time in our development. Stating that a cospan has a pullback can still be done easily with a dependent sum type. The new definition would then be the following.

```
Record Pullback {C: cat} [A B C D: C] (f: A ⇝ C) (g: B ⇝ C)
  (pA: D ⇝ A) (pB: D ⇝ B) := {
  pullback_commutes : f ∘ pA = g ∘ pB;
  ump_pullbacks : ∀ E (qA : E ⇝ A) (qB : E ⇝ B), f ∘ qA = g ∘ qB
    → ∃! u : E ⇝ D, pA ∘ u = qA ∧ pB ∘ u = qB
}.
```

Since pullbacks can also be seen as terminal objects in the category of spans commuting with a given cospan, we have actually chosen to define pullbacks in this way in Rocq, as this allows for the transfer of results about terminal objects to pullbacks. Although the contents of this definition differ from the second version above, they still have the same level of bundling. We have proved equivalence lemmas which we use when we wish to use the textbook definition of pullbacks.

2.4 Pushouts

Pushouts are the dual of pullbacks: they are defined in exactly the same way, but with all arrows reversed. This duality is often exploited in category theory, and we have to do so when formalizing. As in other works [Pou13, AKS15, Wie], we use the fact that the opposite $\mathbf{C}^{op}$ of a category $\mathbf{C}$ is again a category, which works smoothly in dependent type theories:

```
Definition Pushout {C: cat} [A B C D: C] (f: C ⇝ A) (g: C ⇝ B)
    (iA: A ⇝ D) (iB: B ⇝ D) :=
    @Pullback C^op A B C D f g iA iB.
```

This can then be used to transfer results from pullbacks to pushouts. For example, it is straightforward to formalize the notion of monomorphism; and we then have a lemma stating that pulling back a monomorphism gives another monomorphism. Since we can define epimorphism as the dual of monomorphism, we immediately get a proof that pushing out an epimorphism gives an epimorphism.

```
Lemma Epi_stable_po [...] (po: Pushout f g iA iB): Epi f -> Epi iB.
Proof.
  apply (Mono_stable_pb (C := C^op) po).
Qed.
```

(The implicit arguments have been omitted here since they are the same as in the definition of pushouts.)

We use duality as much as possible in our development. There are however places where it does not suffice. For instance, the category of isomorphisms of $\mathbf{C}^{op}$ is only equivalent to the opposite of the category of isomorphisms of $\mathbf{C}$. This means we sometimes have to explicitly cast isomorphisms of one sort into the other.

Note that this is supposed to be as seamless as possible for the user, since the types are always explicitly written without using $\mathbf{C}^{op}$.

3 The hierarchy of adhesive categories

In this section, we present the hierarchy of adhesive categories that we have formalized. The mathematical definitions mention various concepts, such as *regular monomorphisms*, *union of subobjects*, *stable pushouts*, *Van Kampen pushouts* (stable pushouts with *exactness*) [LS05] and *adhesive monomorphisms* [GL12]. These definitions are all about limits and colimits, which makes them fairly easy to formalize. Nevertheless, we do not recall the corresponding definitions here, since they are not relevant to our considerations in this paper. (It should suffice to know that both regular and adhesive monomorphisms are monomorphisms, and similarly for the various kinds of pushouts.)

3.1 Rm-quasiadhesive categories

We start with rm-quasiadhesive categories, the least demanding structure in the hierarchy.

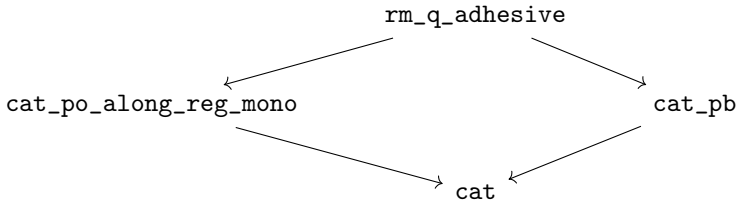
Definition 2 (Rm-quasiadhesive category [GL12])**.** A category is rm-quasiadhesive when it satisfies the following properties:

- it has all pullbacks,
- it has pushouts along regular monomorphisms,
- pushouts along regular monomorphisms are stable,
- pushouts along regular monomorphisms are pullbacks.

Since having all pullbacks (which means every cospan has a pullback) is a common requirement, we first define a structure for those categories. Similarly for categories with pushouts along regular monomorphisms (which means that every span where one of the morphisms is a regular monomorphism has a pushout).

The last two properties are provided explicitly by `rm_q_adhesive`. We see here—in a slightly more sophisticated case than for quivers, precategories and categories—how stronger

structures are built in Hierarchy Builder by combining independent pieces of weaker structure. At this point, our hierarchy is as in the following diagram, with arrows describing inheritance.



3.2 Rm-adhesive categories

Rm-adhesive categories (sometimes also named quasiadhesive categories, in [LS05] for instance) are the intermediate level in the hierarchy.

Definition 3 (Rm-adhesive category [GL12])**.** A category is rm-adhesive when it satisfies the following properties:

- it has all pullbacks,
- it has pushouts along regular monomoprhisms,
- pushouts along regular monomorphisms are Van Kampen.

Note that the first two properties are the same as for rm-quasiadhesive categories; but the third property appears, at first sight, to be unrelated. However, we have:

Theorem 1 ([GL12])**.** *In a category* **C** *with all pullbacks, the following are equivalent:*

1. **C** *is rm-adhesive,*
2. *regular monomorphisms are adhesive,*
   and regular subobjects are closed under binary union,
3. **C** *is rm-quasiadhesive,*
   and regular subobjects are closed under binary union.

Since there are three equivalent characterizations for this level of the hierarchy, we can choose the most convenient one as the native definition, and add it on top of the definition of rm-quasiadhesive categories.

The third point in the above theorem shows that rm-adhesive categories actually inherit from rm-quasiadhesive ones. We thus take it as the native definition, to optimise modularity, so that `rm_adhesive` adds only the property of "closure of regular subjects under binary union" to `rm_q_adhesive`. The other characterizations can be used either to build instances of the native definition or to get some properties from an existing instance.

In Hierarchy Builder terms, this means that point 3 is a *mixin* with a dependency on rm-quasiadhesive categories, and points 1 and 2 are *factories* with a dependency on categories with pullbacks. The implications 1⇒3 and 2⇒3 are declared as *builders*. This makes it possible to create instances in three different ways, at the user's convenience. Conversely, implications 3⇒1 and 3⇒2 can be seen as part of the theory of rm-adhesive categories: the corresponding lemmas state properties which hold in every such category.

3.3 Adhesive categories

We finally move to adhesive categories, which form the richest structure in our hierarchy:

Definition 4 (Adhesive category [GL12])**.** A category is adhesive when it satisfies the following properties:

- it has all pullbacks,
- it has pushouts along monomorphisms,
- pushouts along monomorphisms are Van Kampen.

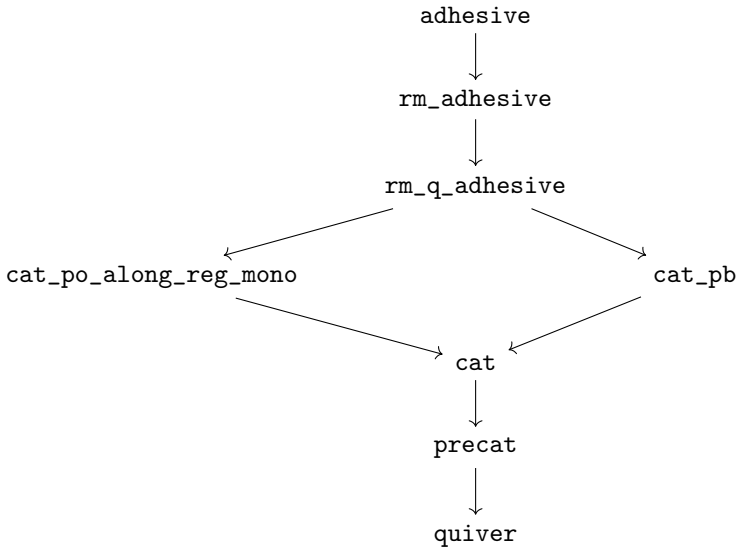
Theorem 2 ([GL12], [LS05])**.** *In a category* **C** *with all pullbacks, the following statements are equivalent:*

1. **C** *is adhesive,*
2. *all monomorphisms are adhesive,*
3. **C** *has pushouts along monomorphisms, those pushouts are stable and are pullbacks,*
4. **C** *is rm-adhesive and all monomorphisms are regular.*

For the same reasons as previously, we use the fourth point as the native definition, since it establishes that adhesive categories inherit from rm-adhesive ones. In Hierarchy Builder terms, point 4 is the new mixin which we use to define the structure, while the three other points are presented as factories.

Another argument in favour of this choice is the fact that when stacking the textbook definitions for rm-adhesive and adhesive categories, we have two functions giving pushouts: one along monomorphisms and one along regular monomorphisms. This means we would sometimes obtain pushouts of a given span in two different ways, without knowing if they result in the same cospan. Basic category theory guarantees that they would at least be isomorphic, but dealing with the resulting isomorphisms would be an additional burden.

In the end, the following diagram represents the full hierarchy.



3.4 Examples

We have formalized two key instances: the category of sets and the category of simple directed graphs (we allow self loops here for simplicity, so those graphs are exactly binary relations on the set of vertices). The former (Set) is adhesive while the latter (SGraph) is only rm-quasi-adhesive [BHK22]. Set is proven to be adhesive using the factory implementing the textbook definition. The textbook definition was also used for SGraph, since there are no factories for rm-quasiadhesive categories. We reuse the constructions in Set in SGraph, which leaves only the consistency regarding edges to be proven. We plan to write a second

proof for Set using a different factory, to compare with the standard one we have formalized so far.

Apart from this specific point, our library only relies on standard axioms: proof irrelevance and functional extensionality. It should be feasible to get rid of those axioms by working with a setoid-based definition of categories. For the two instances discussed above, we also rely on propositional extensionality and definite description: those axioms make it possible to implement the quotient types required for pushouts.

Additionally, there are some places where universe checking is disabled due to a limitation in Hierarchy Builder, but this should be fixed in the future.

4 Related work

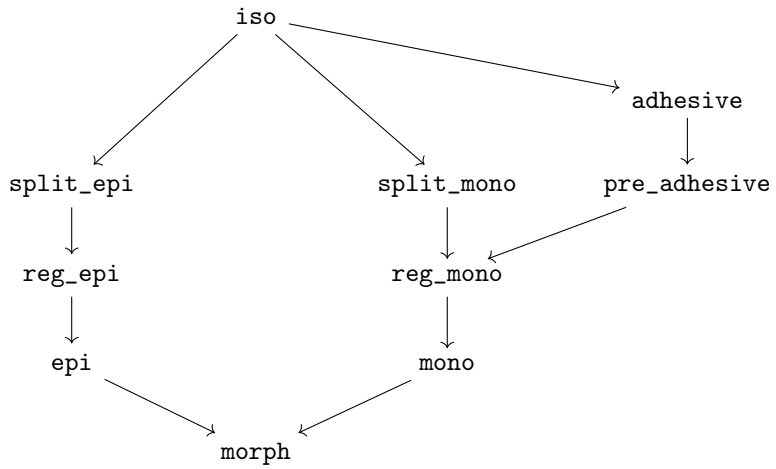
There are already several formalizations of category theory in Rocq and other proof assistants [Wie, mC20, AKS15, VAGo, HC21, GCS14, Sta16, TJ16]. We initially based our work on John Wiegley's library [Wie], but have since moved away from it to use HB. These works are more oriented towards a general-purpose library: few of them have regular monos/epis [HC21, VAGo], and we were only able to find a formalization of adhesive categories (but not rm-quasiadhesive or quasiadhesive) in the Lean Mathlib [mC20].

5 Conclusions and future work

Our library consists of about 6300 lines of code in total. About 2900 are for the preliminary work (pulbacks, pushouts, isomorphisms...), 1600 for the definitions regarding adhesivity (adhesive morphisms, adhesive categories...) and 1800 for the two instances (Set and SGraph). Our next objective is to formalize some results about graph rewriting in adhesive categories, such as those found in section 7 of [LS05], but also to make improvements to the current formalization.

In the short term, a first improvement would be to set up a hierarchy for morphisms, which would remove the need for the explicit use of lemmas to go, for example, from a regular monomorphism to a monomorphism. This hierarchy would have several branches, as shown in the diagram below, with isomorphisms being the richest structure in the hierarchy. (Note that several concepts in this diagram were not mentioned in this article, since they do not directly appear in the definitions of the categories in our hierarchy.)

An important feature here would be to be able to deal with the fact that some of the structures in this hierarchy of morphisms become equivalent in some categories. For example, all monomorphisms are regular in adhesive categories. Furthermore, since properties of morphisms in our library are currently unbundled (except for isomoprhisms), introducing this hierarchy would be a good opportunity to bundle them.



It would also be nice to set up tools for rewriting modulo associativity of composition, and dealing more systematically with isomorphisms. For instance, there are cases where isomorphisms can be removed by renaming the various morphisms appropriately. We would like to automate this task and develop a tactic for deleting a given isomorphism, keeping only its source or target object.

Finally, having a tool to display graphically the diagram currently being worked on would make the proof process easier—even just in read-only mode, unlike in the work of Ambroise Lafont [Laf24], whose more ambitious objective is to perform proofs graphically.

Acknowledgements We would like to thank Cyril Cohen for many helpful discussions on the subject of this paper and on Hierarchy Builder generally.

References

[AFMS16] Jakob L. Andersen, Christoph Flamm, Daniel Merkle, and Peter F. Stadler. A software package for chemically inspired graph transformation. In Rachid Echahed and Mark Minas, editors, *Graph Transformation. ICGT 2016*, LNCS, pages 73–88. Springer Cham, 2016.

[AKS15] Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Univalent categories and the Rezk completion. *Mathematical Structures in Computer Science*, 25(5):1010–1039, June 2015. doi:10.1017/S0960129514000486.

[BGK+22] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. String diagram rewrite theory i: Rewriting with frobenius structure. *Journal of the ACM (JACM)*, 69(2):1–58, 2022.

[BHK22] Nicolas Behr, Russ Harmer, and Jean Krivine. Fundamentals of Compositional Rewriting Theory, April 2022. arXiv:2204.07175 [cs]. URL: http://arxiv.org/abs/2204.07175, doi:10.48550/arXiv.2204.07175.

[CST20] Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy Builder: algebraic hierarchies made easy in Coq with Elpi. In *FSCD 2020 - 5th International Conference on Formal Structures for Computation and Deduction*, volume 167, pages 34:1 – 34:21, Paris, France, June 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://inria.hal.science/hal-02478907, doi:10.4230/LIPIcs.FSCD.2020.34.

[DHW13] Vincent Danos, Russ Harmer, and Glynn Winskel. Constraining rule-based dynamics with types. *Mathematical Structures in Computer Science*, 23(2):272–289, 2013. doi:10.1017/S0960129512000114.

[GCS14] Jason Gross, Adam Chlipala, and David I. Spivak. Experience Implementing a Performant Category-Theory Library in Coq. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, pages 275–291, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-08970-6_18.

[GGMR09] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging Mathematical Structures. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 327–342, Berlin, Heidelberg, 2009. Springer. doi:10.1007/978-3-642-03359-9_23.

[GL12] Richard Garner and Stephen Lack. On the axioms for adhesive and quasiadhesive categories. *Theory and Applications of Categories*, 27:27 – 46, May 2012. URL: http://www.tac.mta.ca/tac/volumes/27/3/27-03.pdf.

[HC21] Jason Z. S. Hu and Jacques Carette. Formalizing category theory in Agda. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, pages 327–342, New York, NY, USA, January 2021. Association for Computing Machinery. doi:10.1145/3437992.3439922.

[HT20] Reiko Heckel and Gabriele Taentzer. *Graph Transformation for Software Engineers: With Applications to Model-Based Development and Domain-Specific Language Engineering*. Springer International Publishing, Cham, 2020. URL: http://link.springer.com/10.1007/978-3-030-43916-3, doi:10.1007/978-3-030-43916-3.

[Laf24] Ambroise Lafont. A diagram editor to mechanise categorical proofs. In *35es Journées Francophones des Langages Applicatifs (JFLA 2024)*, Saint-Jacut-de-la-Mer, France, January 2024. URL: https://hal.science/hal-04407118.

[LS05] Stephen Lack and Paweł Sobociński. Adhesive and quasiadhesive categories. *RAIRO - Theoretical Informatics and Applications*, 39(3):511–545, July 2005. doi:10.1051/ita:2005028.

[mC20] The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 367–381, New York, NY, USA, January 2020. Association for Computing Machinery. doi:10.1145/3372885.3373824.

[Pou13] Damien Pous. Kleene Algebra with Tests and Coq Tools for while Programs. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 180–196, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-39634-2_15.

[Sta16] Eugene W. Stark. Category Theory with Adjunctions and Limits. In *Archive of Formal Proofs*, June 2016. Section: entries. URL: https://www.isa-afp.org/entries/Category3.html.

[TJ16] Amin Timany and Bart Jacobs. Category Theory in Coq 8.5. In Delia Kesner and Brigitte Pientka, editors, *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*, volume 52 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:18, Dagstuhl, Germany, 2016. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

ISSN: 1868-8969. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2016.30, doi:10.4230/LIPIcs.FSCD.2016.30.

[VAGo] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, and others. UniMath — a computer-checked library of univalent mathematics. URL: https://github.com/UniMath/UniMath.

[Wie] John Wiegley. Category Theory in Coq. URL: https://github.com/jwiegley/category-theory.