



Aspects of category theory in proof assistants

Luc Chabassier

► To cite this version:

Luc Chabassier. Aspects of category theory in proof assistants. Logic in Computer Science [cs.LO]. Université Paris-Saclay, 2025. English. ⟨NNT : 2025UPASG055⟩. ⟨tel-05229403⟩

HAL Id: tel-05229403

<https://theses.hal.science/tel-05229403v1>

Submitted on 29 Aug 2025

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 CC BY 4.0 - Attribution - International License

Théorie des catégories et assistants de preuve

Aspects of category theory in proof assistants

Thèse de doctorat de l'université Paris-Saclay

École doctorale n° 84, École doctorale sciences et technologies de l'information et de
la communication, STIC
Spécialité de doctorat : Informatique
Graduate School : Informatique et sciences du numérique. Référent : ENS Paris-Saclay

Thèse préparée dans l'Université Paris-Saclay, CNRS, ENS Paris-Saclay, Laboratoire
Méthodes Formelles, 91190, Gif-sur-Yvette, France, sous la direction de **Gilles DOWEK**,
professeur, la co-direction de **Bruno BARRAS**, chargé de recherche

Thèse soutenue à Paris-Saclay, le 07 juillet 2025, par

Luc CHABASSIER

Composition du Jury

Membres du jury avec voix délibérative

Paul-André MEILLÈS Professeur émérite, CNRS	Président ou Présidente
Assia MAHBOUBI Directrice de recherche, INRIA	Rapporteuse & Examinatrice
Damien POUS Chargé de recherche, CNRS	Rapporteur & Examineur
Ambroise LAFONT Assistant professor, École Polytechnique	Examineur
Samuel MIMRAM Directeur de recherche, École Polytechnique	Examineur
Clément PIT-CLAUDEL Assistant professor, EPFL	Examineur

Titre : Théorie des catégories et assistants de preuve

Mots clés : Assistant de preuve, Interface, Théorie des Catégorie

Résumé : La théorie des catégories est un domaine actif, à la fois en mathématiques et en informatique. Elle tire ses origines de la géométrie algébrique. Sa capacité à manipuler des structures de manière abstraite en a fait un formalisme utile dans la plupart des domaines mathématiques. C'est depuis devenu un sujet de recherche à part entière. De plus, elle a trouvé des applications en logique et informatique en tant qu'outil pour définir des sémantiques de langages.

Une des raisons du succès de la théorie des catégories en tant qu'outil générique pour parler de structures vient non seulement de ses abstractions, mais aussi du développement de langages graphiques pour manipuler ces abstractions. Ces langages permettent à l'utilisateur de catégories d'utiliser son intuition graphique pour manipuler et comprendre des notions abstraites. Le plus primitif de ces langages est celui des diagramme commutatifs, qui représentent des équations dans des catégories sous la forme de graphes.

Simultanément avec l'émergence de la théorie des catégories, on a vu se développer au courant du dernier siècle les assistants de preuves, des logiciels qui manipulent des preuves mathématiques. La motivation derrière ces outils est double. D'abord, la complexité croissante à la fois des preuves mathématiques et des logiciels critiques, menant à une situation où des papiers fondamentaux s'avèrent faux ou à l'explosion d'Ariane 5, motivent l'automatisation de la vérification de la correction des preuves et des programmes. Ensuite, on peut espérer que les ordinateurs vont aider les mathématiciens et informaticiens à réaliser ces preuves.

Même si l'usage d'assistants de preuves dans la recherche en mathématiques reste marginal, un nombre croissant de résultats a été formalisé. La prévalence de la théorie des catégories dans la recherche récente en mathématiques sous-entend la nécessité de son implémentation dans des assistants de preuve. Cependant, la théorie des catégories s'avère compliquées à formaliser, pour des raisons sur lesquelles nous reviendrons.

La première difficulté réside dans la partialité de la composition, l'opération primitive d'une catégorie. Gérer cette partialité est souvent compliqué pour les assistants de preuve. Dans la première partie de nos travaux, nous détaillons l'état de l'art, et proposons une première tentative pour mieux gérer cette partialité en utilisant les capacité de réécriture de λ -calcul. Dans la deuxième partie, nous étendons cette approche en étudiant la relation entre la théorie des types intentionnelle, sur laquelle sont basés la plupart des assistants de preuves, et la théorie des types extensionnelle, plus apte à gérer la partialité de la composition.

La seconde difficulté est plus pragmatique. Comme indiqué ci dessus, un des avantages de la théorie des catégories est son support pour des raisonnements graphiques. Cependant, la plupart des assistants de preuve sont textuels; leur systèmes de notations ne peuvent exprimer un langage graphique. La contribution principale de ce doctorat est le développement d'un plugin pour faire avancer graphiquement des preuves de théorie des catégories dans l'assistant de preuve Coq. Ce plugin est décrit en détail dans les deux dernières parties du manuscrit.

Title : Aspect of Category Theory in Proof Assistants

Keywords : Category Theory, Proof Assistant, Interface

Abstract : Category theory is an active domain in both computer science and mathematics research. Taking its roots in algebraic geometry, its ability to talk about structures in an abstract manner has made it a useful formalism in most fields of mathematics. It has since become a domain of research on its own, especially with theories such as Topos theory. Furthermore, it has found applications in logic and computer science as a powerful tool for defining and discussing the semantics of languages. It has even escaped the realm of theoretical research by finding applications in applied science.

One of the reasons for the success of category theory as a generic tool to discuss structure comes not only from its abstractions but also from the development of graphical languages to manipulate those abstractions. These enable the user of category theory to reuse graphical intuition to manipulate and understand abstract notions. The most primitive of these languages is that of commutative diagrams, which represent equations in categories as graphs.

In parallel to the emergence of category theory, we have seen in the last century the creation of proof assistants, software that manipulate mathematical proofs. The motivation behind these tools is twofold. First, the increasing complexity of both mathematical proofs and critical software, which has led to a situation where fundamental papers were discovered to be false or the explosion of Ariane 5, created an incentive to automate the checking of the correctness of proofs and programs. Second, there is hope that computers will assist mathematicians and computer scientists in the elaboration of these proofs.

While the use of proof assistants in mathematical research has yet to become the norm, an increasing number of results have been formalized. The prevalence of category theory in recent mathematical research indicates the need for its formalization in proof assistants. However, category theory is challenging to formalize for reasons we will explore, and we believe that none of the existing formalizations are satisfactory.

The first difficulty is due to the fact that at its core, category rely on a partial operation, composition. Dealing with partiality is often troublesome for proof assistants. In the first part of our PhD, we describe how this is done in the current proof assistants, how an attempt to improve partiality handling using the rewriting features of the Lambdapi proof assistant. The second part goes further by exploring the relation between intentional type theory, on which most proof assistants are based, and extensional type theory, where many of the challenges linked to dealing with the partiality disappear.

The second difficulty is more pragmatic. As argued above, one big advantage of category theory is its ability to be reasoned with using graphical languages. However, most proof assistants are text based; thus their notation systems cannot express any graphical language. The main contribution of this PhD is a plugin to graphically progress category theory proofs in the Coq proof assistant, described in detail in the last two parts.

Acknowledgements

There are many people that have been instrumental to the realisation of this work, and that deserve to be thanked. I'll try to mention them all, but it is inevitable that I'll forget some people, and they have my deepest apologies.

I need to start by thanking my family, especially my parents, Denis and Geneviève. Without them, I wouldn't be here. This is both true as a trite biological fact, but also as they provided me with the education, context, support, material comfort and sociocultural codes that enabled my scholar success. Speaking of which, I need to thank all the teachers I had until this point, that all contributed getting me to this point. Shout out to Mr. Joaniquet, the first teacher that told me I would grow up to like mathematics in primary education. I also need to mention Mr. Onno and Mr. Boulouch, my two teachers of engineering science in high school, who first enabled and encouraged me to do computer science in a scholar context. Mr. Bayle and Mr. Brevet, my mathematics teachers in *classe préparatoire*, also played a crucial role in teaching me mathematical rigor, and to like it. Finally, I have nothing but gratitude for all teachers I encountered as part of my studies at *École Normale Supérieure*, who challenged me with the deep lessons and fascinating projects.

Even if it is maybe a bit unusual, I need to mention *RPG Maker VX*, as it is while trying to create small games using this software that I discovered programming. While looking to understand better its scripting abilities, I stumbled into what was to become the starting point of my computer career : *Le Site du Zéro*. This website was instrumental in teaching important skills in computer science not only to me, but to many people of my generation. I believe its rebranding as *OpenClassrooms* and subsequent loss of its soul is a deep loss for all the french programming scene, and I'm very grateful to have discovered it at its peak. I also need to thank the random blog post I read years ago, urging people to learn Haskell, since it is through learning this language that I first heard of category theory, a core part of the PhD I would do many years later.

Of course, my enjoyment of computer science wouldn't have flourished without the numerous friends I had to talk and geek with. Particular mention to Pablo Donato, who first incited me to install ArchLinux a long time ago, and may have started my appetite for more *unconventional* software. Since then, I encountered many people whose discussion over a factorio game have probably taught me as much as classes. Théophile, Thibaut, Paul, Lucas, Sarah, Mathieu, Clément, Béranger, Hector, Viviane, Nathanaëlle, Lionel, Martin, Marc, Anatole, Camille... and many more.

In addition to their contribution to my computer science pedigree, all those were and still are friends to enlighten my life through parties and games. To that effect, I also need to mention Clémence, Anthyme, Maëlle, Adélaïde, Maël, Lucie, Marie and Élodie. Sports plays also a big part in my life, especially Parkour. I need to thank the Parkour Paris club, and all the people I've met through it : Nico, Mathilde, the other Mathilde, Raphaël, Thib, Titi, Tof, PA, Simon, Anne, Guillaume, Alex, Oma, Adé, Éva, Marie, Chivi, and many more. Special mention to Thomas and Charles, with whom I have been friend for over 20 years at this point, and we still regularly climb and discuss together !

Finally, all the people I interacted with during my PhD to talk about more academic matters have all my gratitude. Of course, I need to start with Bruno Barras, my *de facto* adviser, for the freedom he left me. I also need to thank Yves Bertot and Jean-Christophe Filliâtre, the members of my CSI, for their help in navigating the complicated last year of my PhD. All the members of the lab have also been very helpful. I need to mention Frédéric, Gilles, Théo, Catherine and Chantal by name for their help at different points of this PhD. This list wouldn't be complete without mentioning all the other PhD student during my time there, Louïse, Amélie, Thiago, Rish, Thomas and Yoan. Finally, I must mention again Pablo Donato, who was my roommate during the 3 years of PhD, also working on his own PhD. Our discussions were very instrumental into shaping my work and helping me keep my sanity. Go check his work !



Aspects of category theory in proof assistants

PhD dissertation by :
Luc Chabassier
Under the supervision of :
Bruno Barras
University :
ENS Paris-Saclay
Université Paris-Saclay

Acknowledgements	1
Introduction	3
I Defining categories in proof assistants	7
1 State of the art	9
1.1 In set theory	9
1.2 Without dependent types	11
1.3 With dependent types	12
2 Lambdapi approach	15
2.1 Strict categories	15
2.2 Integration	16
2.3 Strict categories	17
II Extensional type theory in Lambdapi	25
3 Martin-Löf Type Theory	27
3.1 Definition	27
3.2 Equality	27
3.3 Formalisation in Lambdapi	32
4 Extensional Type Theory	35
4.1 Definition	35
4.2 Embedding in Lambdapi	36
5 Translation from ETT to MLTT	39
5.1 Intuition	39
5.2 Structure	40
5.3 Translating the congruence	42
5.4 Fundamental lemma	42
5.5 Performance evaluation	44
III Graphical interface for diagrams	47
6 Demo	49
6.1 Workflow	49
6.2 Replaying proofs	51
6.3 Comparison with other tools	52
7 The diagram structure	55
7.1 The graph	55
7.2 Path and morphisms	55
7.3 Second level structure	57
7.4 Graph operations	57
7.5 Decomposition	59
8 Lemmas	61
8.1 Language	61
8.2 Associated graph	61
8.3 Application	62
8.4 Proof flow	63
8.5 Generalisations	65

9	The solver	67
9.1	The problem	67
9.2	Decidability	68
9.3	Topological approach	69
9.4	Algorithm	70
9.5	Optimisation though rewriting	71
9.6	Comparison with other algorithms	72
IV	Implementation	77
10	Architecture	79
10.1	Split process organisation	79
10.2	The former data model	80
10.3	The data model	82
10.4	The protocol	86
11	Algorithms and data structures	91
11.1	Graph extraction	91
11.2	Proof relevant disjoint set	92
11.3	Graph layouting	93
11.4	Planar decomposition	96
11.5	Equality term simplification	99
11.6	Equality structure	101
11.7	Minimal associativity	105
11.8	Linear unification	107
11.9	Automatic lemma application	108
12	Coq plugin architecture	109
12.1	Interface communication	109
12.2	Term indexing	109
12.3	Term analysis	110
12.4	ELPI	111
12.5	Context handling	112
13	Interface architecture	115
13.1	Tactic language	115
13.2	Supporting rollbacks	115
13.3	Extensibility	117
13.4	Web version	121
A	Software list	125
B	Bibliography	131

Introduction

Category theory is an active domain in both computer science and mathematics research. Taking its roots in algebraic geometry, its ability to talk about structures in an abstract manner has made it a useful formalism in most fields of mathematics. It has since become a domain of research on its own, especially with theories such as *Topos theory* [1]. Furthermore, it has found applications in logic and computer science as a powerful tool for defining and discussing the semantics of languages. It has even escaped the realm of theoretical research by finding applications in applied science [2].

One of the reasons for the success of category theory as a generic tool to discuss structure comes not only from its abstractions but also from the development of graphical languages to manipulate those abstractions. These enable the user of category theory to reuse graphical intuition to manipulate and understand abstract notions. The most primitive of these languages is that of *commutative diagrams*, which represent equations in categories as graphs.

Arguably, one of the most successful graphical language in this family is *string diagrams* of monoidal categories, which have been applied in many domains, ranging from physics to logic [3]. This language is particularly interesting in that it is topological by nature : one can act on it by simply moving things around.

In parallel to the emergence of category theory, we have seen in the last century the creation of *proof assistants*, software that manipulate mathematical proofs. The motivation behind these tools is twofold. First, the increasing complexity of both mathematical proofs and critical software, which has led to a situation where fundamental papers were discovered to be false [4] or the explosion of Ariane 5 [5], created an incentive to automate the checking of the correctness of proofs and programs. Second, there is hope that computers will assist mathematicians and computer scientists in the elaboration of these proofs. The original example is that of the 4 color theorem, which states that any map can be colored with only 4 colors such that no two adjacent countries share the same color. The first known correct proof reduced the problem to 1834 cases, which were automatically verified using a computer [6].

While the use of proof assistants in mathematical research has yet to become the norm, an increasing number of results have been formalized [7]. The prevalence of category theory in recent mathematical research indicates the need for its formalization in proof assistants [8][9][10][11]. However, category theory is challenging to formalize for reasons we will explore, and we believe that none of the existing formalizations are satisfactory.

This PhD identified two of these challenges, which we will present along with our work to create tentative solutions.

The first difficulty is due to the fact that at its core, category rely on a *partial operation*, composition. Dealing with partiality is often troublesome for proof assistants. **Part I** is about how this is done in the current proof assistants. More precisely, **chapter 1** goes over the formal definition of a category, and provides an overview of the current formalizations. **Chapter 2** presents a failed attempt of our own to deal

[1] Johnstone, *Sketches of an elephant: a Topos theory compendium*, 2002

[2] Fong and Spivak, *Seven Sketches in Compositionality: An Invitation to Applied Category Theory*, 2018

[3] Baez and Stay, *Physics, Topology, Logic and Computation: A Rosetta Stone*, 2010

[4] Voevodsky, *The origins and motivations of Univalent Foundations*, 2014

[5] Dowson, *The Ariane 5 software failure*, 1997

[6] Appel and Haken, *Every Planar Map is Four Colorable*, 1989

[7] Scholze, *Liquid Tensor Experiment*, 2022

[8] Ahrens, Kapulkin and Shulman, *Univalent categories and the Rezk completion*, 2015

[9] Gross, Chlipala and Spivak, *Experience Implementing a Performant Category-Theory Library in Coq*, 2014

[10] Hu and Carette, *Formalizing of Category Theory in Agda*, 2021

[11] Stark, *Category Theory with Adjunctions and Limits*, 2016

[\(1\)](#) `Lambdapi`, a proof assistant based on the $\lambda\Pi$ -calculus modulo rewriting

[\(2\)](#) `Quiver`: a modern commutative diagram editor

[\(3\)](#) `Globular`

[\(4\)](#) `homotopy.io` - A proof assistant implementing the theory of associative n -categories

[\(5\)](#) `Actema` : une interface graphique et gestuelle pour preuves formelles

[\(6\)](#) `ProofWidgets`

[\(7\)](#) `Commutative Diagrams Interface`

[\(8\)](#) `The Rock Proof Assistant`

[\(9\)](#) A diagram editor in your browser

with the partiality in a new way using the specific expressive power of `Lambdapi`[\(1\)](#). [Part II](#) goes further by exploring the relation between *intentional type theory*, on which most proof assistants are based, and *extensional type theory*, where many of the challenges linked to dealing with the partiality disappear.

The second difficulty is more pragmatic. As argued above, one big advantage of category theory is its ability to be reasoned with using graphical languages. However, most proof assistants are text based; thus their notation systems cannot express any graphical language. The need for computer based systems to deal with these diagrammatic proofs has led to three research directions. The first one is a tool for drawing and sharing these diagrams[\(2\)](#). This tool does not have any semantic understanding of the diagrams, and do not allow to do proofs with them. The second is the development of domain specific proof assistants, such as `Globular`[\(3\)](#) or `Homotopy.io`[\(4\)](#), which implement the logic of higher-order string diagrams. The limitation of these tools is that they are limited to proofs expressible in this specific logic; therefore they cannot be used with more generic proof assistants.

Finally, the last research direction is the development of interfaces to perform diagrammatic reasoning as *plugins* for more generic proof assistants. Adding domain specific interfaces to proof assistants has been an area of interest for quite some time, including projects such as `Actema`[\(5\)](#), a plugin for graphical first order proofs, or the Lean proof assistant widget system[\(6\)](#). Category theory is an obvious target for such systems, and one of the first Lean widgets to be developed is a widget to visualize categorical diagrams. During most of this PhD, we developed our own plugin[\(7\)](#) to graphically progress proofs in the `Coq` proof assistant[\(8\)](#). This tool is presented in [part III](#).

In [part IV](#), we discuss the implementation and architectural challenges in developing such a plugin. We also focus on the fact that the plugin is organized to be easy to port to new proof assistants. Therefore, it has the potential to be a domain specific proof assistant that directly operates as an overlay on more generic proof assistants. We believe such an approach is an interesting direction of research because it may enable some domain specific proofs to be stated in a more abstract logic that could work across all proof assistants supported by our plugin. Simultaneously, Ambroise Lafont is developing a plugin with goal similar to ours[\(9\)](#), but with different architecture choices.

Defining categories in proof assistants

I

1 State of the art

1.1 In set theory

This thesis explores the challenges of formalizing category theory in proof assistants. Before we can do so properly, we need to go over what category theory is. Because we focus on the fundamental challenges of such formalizations, we only need to define categories and functors, we do not need anything more advanced.

Category theory was initially introduced by Samuel Eilenberg and Saunders Mac Lane [12] in an attempt to understand natural transformations in algebraic topology. Since then, it has proven a useful formalism in many branches of mathematics, especially algebra, but also in the semantics of programming languages and formalizations of computing formalisms.

It has been proposed as the basis for a new formalization of mathematics, with ETCS [13], an attempt to formulate set theory in a categorical manner. More importantly for us computer scientist, it has been used as a new element in the famous *Curry-Howard correspondence*. Indeed, this correspondence states that there is an isomorphism between logic and computation, by mapping proofs in different logic to programs in the corresponding languages. Lambek [14] extended it to map proofs or programs to generalized elements in specific categories. This may explain the predominance of category theory in the study of computation and logic.

What is category theory ? In an abstract way, it is a formalism to talk about structures by focusing on the transformations that preserve them. A more technical point of view is to describe a category as :

- ◊ An abstraction of sets and functions¹.
- ◊ A proof relevant preorder.
- ◊ A monoid with partial composition.
- ◊ Transitively closed multigraph.

Definition 1. A category is a sextuplet $(M, O, s, t, i, \circ)$ with :

- ◊ M is a set of morphisms.
- ◊ O is a set of objects.
- ◊ s and t are functions from M to O , called respectively the source and target functions.
- ◊ i is a function from O to M called the identity.
- ◊ $\forall o \in O, s(i(o)) = t(i(o)) = o$ ².
- ◊ $\circ$ is a function from $M_s \times_t M^3$ to M called the composition.
- ◊ $\forall (m, m') \in M_s \times_t M, s(m \circ m') = s(m') \wedge t(m \circ m') = t(m)$ ⁴.

We also need two more axioms :

- ◊ The unitality⁵ of the identity : $\forall m \in M, m \circ i(s(m)) = m \wedge i(t(m)) \circ m = m$.
- ◊ The associativity of the composition : $\forall m_1, m_2, m_3 \in M, s(m_1) = t(m_2) \wedge s(m_2) = t(m_3) \implies m_1 \circ (m_2 \circ m_3) = (m_1 \circ m_2) \circ m_3$

Definition 2. Given a category C , we write $x \in C$ to say x is an object of C .

Given $a, b \in C$, we write $C(a, b) = \{m \in M \mid s(m) = a \wedge t(m) = b\}$.

[12] Eilenberg and MacLane, *General theory of natural equivalences*, 1945

[13] Lawvere, *An Elementary Theory of the Category of Sets*, 1964

[14] Lambek, Scott and Scott, *Introduction to higher order categorical logic*, 1994

1. Although this analogy is very common, care must be taken. Indeed, sets and functions have a lot of structure that is not present in generic categories, and so this intuition is better suited for more structured categories like *cartesian closed categories* or even *topos*.

2. In other words, i is a section for both s and t .

3. This notation is the pullback notation $M_s \times_t M = \{(m, m') \in M^2 \mid s(m) = t(m')\}$. It allows us to write the composition as a complete function, instead of interpreting it as a partial function on M^2 .

4. This axiom is more understandable visually, given $a, b, c \in O, m, m' \in M$ such that $s(m') = a, t(m') = s(m) = b$ and $t(m) = c$, we can represent them in the following manner : $a \xrightarrow{m'} b$ and $b \xrightarrow{m} c$. The composition is then simply the concatenation of those morphisms : $a \xrightarrow{m \circ m'} c$.

5. The identity is a neutral element for the composition.

Remark 1. If we forget the identities and composition, the quadruplet (M, O, s, t) is exactly a multigraph. The identities enforce it to be reflexively closed, and the composition enforces it to be transitively closed. Thus, the intuition that a category is a transitively (and reflexively) closed multigraph.

Remark 2. If we force the set of objects to be a singleton, the composition becomes a total function on M , s and t become trivial and i simply designates a single element in M that is neutral with regard to the composition : we have defined exactly a monoid. Indeed, monoids are exactly categories with a single object, and the addition of new objects force the composition to be partial, thus the intuition that a category is a monoid with partial composition.

Remark 3. If we require that, for all $a, b \in O$, $C(a, b)$ is at most a singleton, we obtain exactly a preorder. Let us define $a \leq b := C(a, b) \neq \emptyset$. The existence of the identities force $C(a, a)$ not to be empty, so we have $a \leq a$, and the composition gives us the transitivity of $\leq$. We can then interpret a category as a preorder where we not only remember if objects are comparable but also remember a witness of this relation, thus the intuition that a category is a proof relevant preorder.

Remark 4. We have $M = \bigsqcup_{a,b \in C} C(a, b)$. So instead of the triplet (M, s, t) , we could have asked for a family of sets indexed by pairs of objects $(C(a, b))_{a,b \in O}$. Furthermore, with this point of view, the identities, instead of being described as a function, can be described as an element of $\Pi_{a \in O} C(a, a)$. Finally, the composition could be replaced by a family of compositions living in :

$$\Pi_{a,b,c \in O} (C(b, c) \times C(a, b) \rightarrow C(a, c))$$

The previous remark gives us a view of the definition of a category that is almost directly translatable in dependent types. Indeed, all families of objects simply need to be replaced by dependent functions. This approach is described in more detail in [section 1.3](#).

One of the main insights of category theory is shifting the focus from the objects studied to their relations, the morphisms. However, in the previous definition, it is not obvious whether morphisms are actually more important than objects. However, we can define categories without mentioning objects, and obtain the same notion of categories.

Definition 3. A category is a quadruplet $(M, s, t, \circ)$ where :

- ◇ M is a set of morphisms.
- ◇ s and t are functions from M to M called the source and target functions.
- ◇ $\forall m \in M, s(s(m)) = t(s(m)) = s(m) \wedge s(t(m)) = t(t(m)) = t(m)$.
- ◇ $\circ$ is a function from $M_s \times_t M$ to M called the composition.
- ◇ $\forall (m, m') \in M_s \times_t M, s(m \circ m') = s(m') \wedge t(m \circ m') = t(m)$.

The associativity axiom is the same as before. We also need an equivalent to the unitality axiom :

$$\forall m \in M, t(m) \circ m = m \wedge m \circ s(m) = m$$

The intuition behind this definition is that identities are an injection of objects in the morphisms of a category, so we can directly use identity morphisms instead of objects. Then the source and target functions return the corresponding identity instead of the objects, which no longer exist explicitly. We can show that the identities are exactly the morphisms in the image of s , or equivalently t .

Lemma 1. $\text{Im } s = \text{Im } t$

Proof.

$$\begin{aligned} x \in \text{Im } s &\implies \exists m \in M, x = s(m) \\ &\implies \exists m \in M, x = t(s(m)) \\ &\implies x \in \text{Im } t \end{aligned}$$

And :

$$\begin{aligned} x \in \text{Im } t &\implies \exists m \in M, x = t(m) \\ &\implies \exists m \in M, x = s(t(m)) \\ &\implies x \in \text{Im } s \end{aligned}$$

□

We write $O = \text{Im } s$. s and t restricted on this set have their image on the set, and are the identity by definition.

Theorem 2. *Definition 1 and definition 3 are equivalent.*

Proof. Let C be a category according to definition 1. Then $(M, i \circ s, i \circ t, \circ)$ is a category according to definition 3.

Let C' be a category according to definition 3. Then $(M, \text{Im } s, s, t, \iota, \circ)$, where ι is the inclusion of $\text{Im } s$ in M , is a category according to definition 1.

The composition of these two associations does not give the identity, but an object that is isomorphic with its source. Indeed, since it forgets the set of objects, it can simply reconstruct a set in bijection, replacing each object with its identity morphism. $\square$

1.2 Without dependent types

We only studied one formalization without dependent types, done by Stark [11] in Isabelle/HOL. It is loosely based on definition 3 in that it does not refer to objects. Instead, it proceeds by first defining *partial magmas*, sets with a partial operation. The partial operation is encoded as a total operation, and an *null* element in the set that represents an invalid composition. This element must be an *absorbing element*⁶ for the composition.

With this definition, one can identify *identity* elements, which are objects that, whenever another element can be composed with them, either on the left or on the right, yield the element composed with them. For each element m in the magma, if there is an identity that can be composed with it on the right, it is called a *domain* of m , and if it can be composed on the left, it is called a *codomain* of m . Thus, each element of the magma has a set of domains and codomains, possibly empty.

With this, a category is defined as an associative partial magma where every element (except for *null*) has at non-empty set of domains and codomains, with the following conditions on compositions :

- ◇ $h \circ g \neq \text{null} \wedge (h \circ g) \circ f \neq \text{null} \implies g \circ f \neq \text{null}$
- ◇ $g \circ f \neq \text{null} \wedge h \circ (g \circ f) \neq \text{null} \implies h \circ g \neq \text{null}$
- ◇ $g \circ f \neq \text{null} \wedge h \circ g \neq \text{null} \implies (h \circ g) \circ f \neq \text{null}$

These conditions determine the partiality of the composition, giving it the structure of a graph. Indeed, under these conditions, we can prove that each non-null element has exactly one domain and one codomain, and that the domain and codomain of identities are themselves. Therefore, we indeed have a formalization equivalent to definition 3, but formulated in a more *intentional*⁷ manner.

This formalization has two primary advantages. The first is that it is a purely first order formalization, without the need for dependent types. The second is that every morphism is composable with each other, in the sense that you can always write $f \circ g$ for every f and g . While this is convenient, it also means that care must be taken to assert that it cannot be null in the right places. We think that the hassle of inserting checks and hypotheses for the non nullity of morphisms outweighs the convenience of being able to compose morphisms without checks.

[11] Stark, *Category Theory with Adjunctions and Limits*, 2016

6. Composition with *null* must always yield *null*.

7. By intentional, we mean without explicitly constructing the elements, but rather characterizing them through axioms and their relations to each other. Here the source and target functions and their relation to the partiality of composition are implicit.

1.3 With dependent types

With dependent types, formalization is straightforward. Indeed, [figure 1.1](#) shows a typical formalization in Coq, that follows pretty much directly the description in [remark 4](#). Instead of using a single type for morphisms, a dependent type is used, allowing the source and target of each morphism to be distinguished in the type system.

```
Record Category := Category {
  object: Type;
  morphism: object -> object -> Type;
  identity: forall x, morphism x x;
  compose: forall s d d',
    C[d, d'] -> C[s, d] -> C[s, d'];
  associativity: forall x1 x2 x3 x4
    (m1: C[x1,x2])(m2: C[x2,x3])(m3: C[x3,x4]),
    (m3 o m2) o m1 = m3 o (m2 o m1);
  left_identity: forall a b (f: C[a,b]), 1 o f = f;
  right_identity:
    forall a b (f: C[a,b]), f o 1 = f;
}.
```

Figure 1.1 Category formalisation in Coq.

In proof assistants such as Coq, type universes are implicit. Therefore, we have `Category : Type`, but this type would actually be a higher universe than the one for the object of a category, so we cannot create the category of categories. With *universe polymorphism*, an experimental feature, the type `Category` is implicitly parameterized by the universe level of the objects and morphisms. It would then be possible to have a category of categories up to a certain level, which would not include itself and would thus be consistent. In Agda, where universe polymorphism is explicit in the syntax, this is exactly what is done [\[10\]](#): the category type is parameterized by the universe level, so one can have a category of level 1 of categories of level 0⁸, and so forth for each level.

Dependent types theories have two main variants: intentional [\[15\]](#) and extensional [\[16\]](#). In extensional type theory, equality behaves as expected by classical mathematicians. However, type checking in extensional type theory is undecidable; therefore, proof assistants are usually based on the intentional variant. However, equality behaves counterintuitively in this type theory. More precisely, there are two equalities, *definitional equality* and *propositional equality*, which serve different roles and only partially correspond to the intuition of mathematical equality. A more detailed discussion of the relation between intentional and extensional type theory, and of these two equalities can be found in [part II](#).

What matters to our discussion is that definitional equality is decidable, automatically checked by the proof assistant and cannot be extended. On the other hand, propositional equality can be manipulated as a proposition, but hides a strange structure [\[17\]](#). The definition in [figure 1.1](#) allows the user to compose only morphisms for which the target of the second morphism is definitionally equal to the source of the first one. The advantage of this approach is that the

[\[10\]](#) Hu and Carette, *Formalizing of Category Theory in Agda*, 2021

8. It is actually a bit more complicated than that, since the computation for the level includes the level of the morphisms types, and the level of propositions for the identity and associativities.

[\[15\]](#) Martin-Löf, *An Intuitionistic Theory of Types: Predicative Part*, 1975

[\[16\]](#) Martin-Löf, *A Theory of Types*, 1971

[\[17\]](#) Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*, 2013

proof assistant automatically checks if morphisms are compatible, and if it allows a composition to be written, it means that the composition is valid, so there is no need to check afterward or to pollute the theorem statements with compatibility hypotheses. However, if we have two morphisms that are compatible only up to propositional equality, then there is no obvious way to compose them. In that sense, the composition is too restrictive, in that it does not allow to compose some morphism we want to compose.

This limitation in the composition has a deeper reason. Indeed, in the absence of an additional axiom, propositional equality behaves like a second category structure on the object, more precisely as a *groupoid* structure, and its interaction can be nontrivial with the category structure we define. On objects, one can show that every equality gives rise to an isomorphism⁹. Ahrens, Kapulkin, and Shulman showed [8] that with the univalence axiom, every category can be *completed* into a category where equalities on objects are exactly isomorphisms between objects¹⁰. This means that, in general, without any additional axioms, equalities between objects cannot be treated as stronger than isomorphism¹¹.

Two directions can now be considered. The first is to add axioms or hypotheses to ensure that the equality has no additional structure. This can generally be achieved by adding axiom K¹² or uniqueness of identity proofs¹³. To avoid adding axioms, one can ask for these properties to be true only of the object type, or replace the use of types by setoids¹⁴. The setoid approach was, for example, used by the Agda formalization[10].

However, one can embrace the structure hidden in equality by adding axioms that enable its use. This is the approach of Homotopy Type Theory. By adding the *univalence axiom*¹⁵, one can construct categories in which equalities between objects are exactly isomorphisms[8]. Thus, in this context, one can forget about propositional equality and think purely in terms of isomorphisms without losing any expressive power.

This problem of the interaction of the equality structure with the category definition also concerns the morphism type. Here, the problem is slightly different : the groupoid structure on the types means that what we wanted to be a category actually behaves like an $(\infty, 1)$ -category. To avoid this, most formalizations (except the one using setoids) add an additional axiom to their category definition, to force this higher structure to be trivial [9] :

morphism_hset:

```
foralll a b (m1 m2: morphism C a b) (p1 p2: m1 = m2),
  p1 = p2.
```

9. Given an equality $p: a = b$, we can rewrite the target in the identity $\text{id} : \text{morphism } C \ a \ a$ to obtain a morphism from a to b , which can be proven to be an isomorphism.

[8] Ahrens, Kapulkin and Shulman, *Univalent categories and the Rezk completion*, 2015

10. Such a category, sometimes called a *univalent category*, has many good properties that would rely on the axiom of choice to prove in ZFC. Indeed, it is in a certain sense *skeletal*, since every isomorphic objects are equal, even if being equal is weaker than in ZFC. Therefore, many proofs that would require choosing an object in an isomorphic class work without needing choice.

11. This is an interesting restriction, because it is usually considered bad practice to consider the properties that distinguish isomorphic objects.

12. $K : \Pi(T : \text{Type}), \Pi(a : T), \Pi(p : a =_T a), p = \text{refl } a$

13. $\text{UIP} : \Pi(T : \text{Type}), \Pi(a \ b : T), \Pi(p_1 \ p_2 : a =_T b), p_1 = p_2$

14. A setoid is a pair of a type with an equivalence relation that is treated as the equality on this type. Setoids have better properties than types with propositional equality, therefore they are often used to circumvent the shortcomings of the latter without having to add axioms or change the type system.

15. A bit technical to formally enunciate, it tells that the equality on a universe is equivalent to the equivalence of type in this universe. Technicality resides in the definition of what it means to be *equivalent*.

[9] Gross, Chlipala and Spivak, *Experience Implementing a Performant Category-Theory Library in Coq*, 2014

2 Lambdapi approach

2.1 Strict categories

In the previous chapter, we have seen that a main difficulty in formalizing category theory in dependent type theory is dealing with the particular structure of propositional equality. Of course, this problem would not exist if the relevant equality were definitional, since it behaves more like what we expect of mathematical equality. However, intensional theories are often too limited to capture the entire equality structure of certain categories. Indeed, in proof assistants such as Coq, it is not extensible, so encoding a complex structure is often impossible. On the other hand, in extensional theories, definitional equality has the entire expressive power of propositional equality, but the theory has fewer metatheoretical properties, thus proof assistants do not usually implement it. We will discuss in more detail the specific properties of extensional type theory and its relation to intentional type theory in the next part.

In this development, we attempted to create a middle ground between extensional and intentional using *Lambdapi*. *Lambdapi* is a proof assistant which expands the functionalities of Dedukti⁽¹⁰⁾. Dedukti is a type checker implementing $\lambda\Pi$ -calculus modulo. The $\lambda\Pi$ -calculus is a lambda calculus with dependent types, but no polymorphism. The modulo part means the definitional equality is extensible.

The syntax of the $\lambda\Pi$ -calculus is shown in figure 2.1. Most typing rules are as expected for the λ -calculus. However there is a subtlety in the typing rule for abstraction :

$$\frac{\Gamma \vdash U : \text{TYPE} \quad \Gamma, x : U \vdash t : T}{\Gamma \vdash \lambda(x : U), t : \Pi(x : U), T}$$

Because the derivation $\Gamma \vdash \text{TYPE} : \text{TYPE}$ does not hold, as it would make the type system inconsistent, there is no way to quantify on types. As mentioned previously, this type system is not polymorphic. If the *modulo* part were not here, this would make it too restrictive to use for anything interesting, but the idea behind Dedukti and *Lambdapi* is that most of the features we would expect from a fully featured type system can be reimplemented by adding rewriting rules.

The modulo part means that the definitional equality can be extended in specific ways. In typical type systems, the definitional equality is defined as the reflexive and transitive closure of a confluent¹, terminating² and type preserving³ rewriting system on terms.

Such a system is usually centered on the β reduction, which *applies* a function given by a λ abstraction by replacing all instances of its argument variable in its body by the argument it is applied to. It is also often extended by specific reductions for specific types in the theory, of which the $\lambda\Pi$ -calculus has none, and the δ extension that replaces a name by its definition.

Lambdapi and Dedukti allow the user to write their own rewrite rules, by writing a pair of a pattern and a term that can use matched subterms. They then check whether the rule preserves typing⁴. The user must verify that the system is still confluent and terminating on its own. Then, terms can be written where the definitional equality

(10) Dedukti, a logical framework

$$\begin{array}{l} u, v, T, U := \lambda(x : T), u \\ \quad \mid x \\ \quad \mid \Pi(x : T), U \\ \quad \mid u \ v \\ \quad \mid \text{TYPE} \end{array}$$

Figure 2.1 $\lambda\Pi$ -calculus syntax.

1. Confluent means that in cases where the rewriting system is not deterministic, the choice made does not matter because as we keep rewriting we end up with the same term. The concrete definition is formulated to make sense in the presence of non termination, but we do not need to concern ourselves with it.

2. Termination means that whatever order the rewrite rules are applied, the term will end up being rewritten to a term on which no rewrite rule can be applied.

3. Type preserving means that if a term is well typed, and a rewriting rule applies to it, the resulting term is also well typed and has a type that is definitionally equal to the type of the first term.

Remark 1. An additional rewrite rule that is often considered, which is the η extension. It replaces a well typed function $f : \Pi(x : A), B$ with $\lambda(x : A), f \ x$. However, in this direction, it does not terminate, yet this is the direction that is sometimes necessary to get confluence. The correct handling of the η expansion is still source of many discussions.

4. This problem is undecidable, so they may refuse rules that would be type preserving.

5. This implies that it is not possible to have an hypothesis that says *suppose that a and b are definitionally equal*. Terms are either definitionally equal or not in each context, and this cannot be changed by quantification. Thus, although this equality can be extended, it cannot be reasoned with.

6. Sometimes also called *thin categories*, they are the categories where there is at most one morphism between any two object pairs. They are isomorphic to preorders.

7. Categories where the only morphisms are the identities. The discrete categories are exactly the sets, because they are characterized by their set of objects, and the functors are exactly the functions between objects.

8. The category of topological spaces.

9. The category of R modules for R a ring.

10. See [section 2.3.3](#) for details.

Definition 1. A functor F between two categories C and D is a pair of :

- ◇ A function $F_0 : O_C \rightarrow O_D$ between the objects.
- ◇ A function $F_1 : M_C \rightarrow M_D$ between the morphisms.

Such that, whenever it makes sense :

- ◇ $s(F_1(m)) = F_0(s(m))$.
- ◇ $t(F_1(m)) = F_0(t(m))$.
- ◇ $F_1(i_x) = i_{F_0(x)}$.
- ◇ $F_1(m_1 \circ m_2) = F_1(m_1) \circ F_1(m_2)$.

11. In typical formalizations in dependently typed proof assistants, the first two equalities are definitional and the last two are propositional.

12. We use the adjective *lax* to denote categories that are not strict.

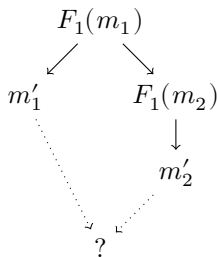


Figure 2.2 Confluence of a functor application.

has been extended by these rewrite rules, and the fully reduced version of the terms can also be queried, allowing computation modulo those rewrite rules.

Therefore, we have a system that is still intentional, in that the definitional equality must still be decidable and cannot be quantified upon⁵, but it can also be extended in some manner. We wonder if this could be used to define some categories in *Lambdapi* such that the equality of morphisms is exactly the definitional equality. We call these categories *strict categories*.

2.2 Integration

Of course, since *Lambdapi* must be able to *decide* equality, it means our approach will definitely be limited to categories with decidable equality, which excludes big categories such as *Sets*, topological spaces, or groups immediately. For this approach to be useful, we also need a method to define non-strict categories and ensure that the two can be used together in a coherent manner. We will first describe how this work could fit (or fail to do so) in a broader formalization of category theory. The details of our formalization are provided in the next section.

We defined a few simple categories, such as the *preorder categories*⁶, or the *discrete categories*⁷, and a few common constructions, such as product categories.

All of these strict categories work well together. However, some of the most interesting categories such as *Set*, *Top*⁸, and *R-Mod*⁹ do not have decidable equality. Furthermore, even simple constructions, such as the coproduct of categories, turned out to be technical to define¹⁰. Therefore, we cannot escape the need to ensure strict and non-strict categories coexist for our formalization to be useful.

Categories usually interact through *functors*. Functors are a pair of a function from the objects of the first category to the objects of the second and a function between the morphisms of the two categories, which is *compatible* with the categorical structure. Once again, there should be different notions of functors, depending on whether the equalities in [definition 1](#) are definitional or propositional¹¹.

In addition to this distinction, we must consider whether the functor is between strict categories, between lax categories¹², from strict to lax, and from lax to strict. Functors to strict categories will necessarily be strict, since the equalities are in the target category, where, by definition, propositional and definitional equality are the same on morphisms. While we have not explored this in detail, it probably makes no sense to try to define a strict functor between lax categories, but we know that functors using propositional equality can be defined between them, since this is what is done in usual formalizations.

Difficulties arise when attempting to define functors from strict categories to lax categories. Indeed, given m_1 and m_2 in a strict category such that $m_1 \equiv m_2$, to preserve confluence we need $F_1(m_1) \equiv F_1(m_2)$ (see [figure 2.2](#)). Therefore, the functor image must be a strict subcategory. This limits the types of functors that can be written. Thus, functors from strict categories to lax categories are intrinsically limited. This intrinsic limitation is particularly problematic because a very common case in category theory is studying the functor category from a simple category to a more complex one in an attempt to understand

the complex category. Thus, while we could hope to obtain a strict formalization of some shape categories such as simplices, cubes, or globules, it would result in complications when trying to do simplicial sets, globular spaces, or cubical groups. We could circumvent this by using functors that do not compute, which means that the application of the functor to a morphism does not reduce, and instead have axioms introducing propositional equalities between the functor and its image. Therefore, to remove propositional equalities in the source category, we need to introduce more in the functor definition, giving an object that is not much easier to work with.

This realization led us to abandon this approach. The next chapter describes what we achieved, but the work presented was stopped at an exploratory phase.

2.3 Strict categories

We defined the type for strict categories as displayed in [figure 2.3](#). It defines a type for categories, along with functions that provide the object and morphism types for categories, identities and compositions. We added reduction rules for the unitality and associativity, even though they are not necessary for any specific strict categories. However, this allows the construction of new categories depending on previous abstract ones, and on the unitality and associativity holding there.

```
symbol Cat : TYPE;
constant symbol Ob : Cat → TYPE;
constant symbol Mph {C : Cat} : Ob C → Ob C → TYPE;
constant symbol Id {C : Cat} (a : Ob C) : Mph a a;
symbol Comp {C : Cat} {a b c : Ob C}
  : Mph b c → Mph a b → Mph a c;

rule Comp (Id _) $g ↦ $g
with Comp $f (Id _) ↦ $f
with Comp $f (Comp $g $h) ↦ Comp (Comp $f $g) $h;
```

Figure 2.3 Strict categories definition in lambdapi.

2.3.1 Finite categories

Finite discrete¹³ and finite linear¹⁴ categories were defined as the basic building blocks. To define these categories, we first need to discuss *finite types*. The easiest way to do so is to first define the natural numbers, and then define finite types.

```
symbol plus : N → N → N;
rule plus n0 $n ↦ $n
with plus $n n0 ↦ $n
with plus (nsucc $n) $m ↦ nsucc (plus $n $m)
with plus $n (nsucc $m) ↦ nsucc (plus $n $m)
with plus $n (plus $m $k) ↦ plus (plus $n $m) $k;
```

Figure 2.4 Definition of addition in Lambdapi.

¹³ A *discrete* category is one that only has identities as morphisms.

¹⁴ A *linear* category is a *thin* category and thus a preorder, where the order is linear.

Natural numbers are simply defined as usual as Peano numbers. One advantage of `Lambdapi` is that it allows us to define addition by reducing on both sides, so addition is definitionally symmetric. From the naturals, we can define a family of types indexed by the naturals `Fin n` that represent a finite type of length n using the usual construction of such a type shown in [figure 2.5](#).

```
symbol Fin (n : N) : TYPE.
constant symbol fin0 (n : N) : Fin (nsucc n);
constant symbol finS {n : N} : Fin n → Fin (nsucc n);
```

Figure 2.5 Definition of finite types.

The finite discrete category is simply the category with objects the finite type :

```
constant symbol FinC (n : N) : Cat;
rule Ob (FinC $n) ↪ Fin $n;
```

This is all is needed. Indeed, the generic definition of a category ensures that every object has an identity morphism. It also ensures that compatible morphisms have a composition that is definitionally quotiented according to unitality and associativity. In this case, it is not useful since the identities are never compatible with each other. However, if we want to define a free category over a graph, we only need to add symbols for the morphisms corresponding to the edges of the graph, and the category definition ensures that the free compositions are well defined. This trick is exactly what we used to define finite linear categories. A finite linear category is the free category over the graph with a finite set of vertices, ordered according to a linear order. See [figure 2.6](#) for a representation of the case $n = 4$.

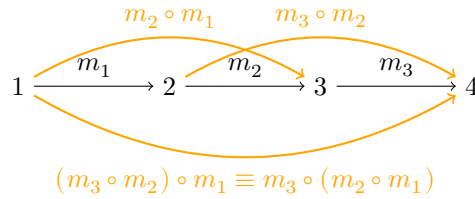


Figure 2.6 The free category over the linear graph of size 4. The graph is in black, and the freely generated morphisms in orange. The identities are not displayed.

Therefore, to define a finite linear category, we simply need to define a category in which the objects are `Fin (nsucc n)`, and for every $0 \leq k < n$ a morphism between k and $k + 1$. In practice, it is slightly more technical because the successor is not well defined in `Fin n`. However, it can be safely defined as an operator from `Fin n` to `Fin (nsucc n)`. Thus, while the objects are `Fin (nsucc n)`, we index the morphisms by `Fin n`. Then, we automatically obtain the free category from those morphisms.

2.3.2 Product

The product of two categories C and D is the category in which the objects are pairs in $C \times D$, and the morphisms are pairs of morphisms in C and D . Identities and composition are defined pairwise. The

formalization is straightforward and intuitive. [figure 2.7](#) shows the definition of objects and morphisms as encoding of products¹⁵.

```
constant symbol PCat : Cat → Cat → Cat;
symbol mkP {C D : Cat} : Ob C → Ob D → Ob (PCat C D);
symbol π1 {C D : Cat} : PCat C D → Ob C;
symbol π2 {C D : Cat} : PCat C D → Ob D;
rule π1 (mkP $p _) ↪ $p
with π2 (mkP _ $p) ↪ $p
with mkP (π1 $p) (π2 $p) ↪ $p;

symbol PHom {C D : Cat} (a b : Ob (PCat C D)) : TYPE
= @Mph (PCat C D) a b;
symbol mkPf {C D : Cat} {a b : Ob C} {c d : Ob D}
: @Mph C a b → @Mph D c d → PHom (mkP a c) (mkP b d);
symbol πf1 {C D : Cat} {a b : Ob (PCat C D)}
: PHom a b → @Mph C (π1 a) (π1 b);
symbol πf2 {C D : Cat} {a b : Ob (PCat C D)}
: PHom a b → @Mph D (π2 a) (π2 b);
rule πf1 (mkPf $f _) ↪ $f
with πf2 (mkPf _ $f) ↪ $f
with mkPf (πf1 $f) (πf2 $f) ↪ $f;
```

Figure 2.7 Definition of the objects and morphisms of the product category.

We can now define the identities by *pushing down* the identities, and doing the same with the compositions. This is displayed in [figure 2.8](#).

```
rule @Id (PCat $C $D) $a
↪ mkPf (@Id $C (π1 $a)) (@Id $D (π2 $a))
rule @Comp (PCat $C $D) $a $b $c $f $g
↪ mkPf
  (@Comp $C (π1 $a) (π1 $b) (π1 $c) (π1 $f) (π1 $g))
  (@Comp $D (π2 $a) (π2 $b) (π2 $c) (π2 $f) (π2 $g))
```

Figure 2.8 Definition of the identities and compositions in a product category.

An interesting property of the product of categories is that it defines an operation on categories that is associative up to isomorphism. Furthermore, if we add the discrete category with a single element, it acts as a neutral element for the product, again up to isomorphism. Having a neutral element and a binary operation that is (almost) associative can be considered a *biased*¹⁶ way to define an n -ary operation for every n . However, because associativity and neutrality are only up to isomorphism, there is still an arbitrary choice when trying to take the product of n categories. To circumvent this choice, we can directly define the product of n categories.

First, we require a method to specify a finite number of categories. One way is to define the known *vector*¹⁷ dependent type. However, because we already have a finite type, we can simply use the type $\text{Fin } n \rightarrow A$, which is isomorphic to $\text{Vect } n A$.

The n product of objects and morphisms is then defined in a similar way, except that instead of using a constructor that takes two arguments,

¹⁵. Instead of encoding the product twice, once for the object and once for the morphisms, we could have defined a generic product type former. However, we would have needed to encode polymorphism. While it is usual to do so, we do not need it, and it is a bit technical. Furthermore, the encoding of pairs is sufficiently simple that duplicating it is not a real problem.

¹⁶. The notion of bias comes from the realization that often, what one wants to define is an associative n -ary operation of a set, but instead of doing so directly for every n , only the $n = 0$ and $n = 2$ cases are given, along with proofs of unitality and associativity. Because all the other cases can be defined from these two, it is an efficient way to do so. However, this gives the impression that the case $n = 2$ is somehow more important, thus the *biased* denomination. In contrast, a definition is called *unbiased* when it directly defines the n -arity operation for each n .

¹⁷. $\text{Vect } n A$ is the type of n -uplets of objects of type A . It is often used as an example of both the power and limitation of dependent types, because it is simple to define.

the constructor takes a dependent function from $\text{Fin } n$ to the relevant type. For example, in the case of objects, the type is $\prod(k : \text{Fin } n), \text{Ob } (f \ k)$, where f is the function $\text{Fin } n \rightarrow \text{Cat}$ that specifies the categories. Then, there is a single projection function parameterized with an integer in $\text{Fin } n$, that computes with regard to the constructor as expected. This computation, along with the rules pushing identities and composition down are shown in [figure 2.9](#).

```
constant symbol UProd {n : N} : (Fin n → Cat) → Cat.
symbol un {n : N} {f : Fin n → Cat} (k : Fin n)
  : UProd f → Ob (f k);
rule un $k (mkUP $o) ↪ $o $k;
rule @Id (UProd $C) $x
  ↪ mkUPf $x $x (λ k, @Id ($C k) (un k $x));
rule @Comp (UProd _) $a _ $c $f $g
  ↪ mkUPf $a $c (λ k, Comp (unf k $f) (unf k $g));
```

Figure 2.9 The computation rules for the unbiased product category.

2.3.3 Coproduct

After the product, the next natural construction we want to tackle is the *coproduct*, also called the disjoint sum. The coproduct of two categories is the category formed of the two previous categories side-by-side. The objects are the disjoint set of objects, and the morphisms are the same as the source category if they come from the same source category, and the empty set otherwise.

The complications in implementing it appear when looking at the composition : we only know how to compose two morphisms if they belong to the same category. However, the fact that they have an endpoint in common forces them to belong to the same category. While we can *prove* this property, we would at most obtain a propositional equality between the categories the two morphisms belong to, which we cannot use as a reduction rule.

To obtain a system that computes, we require another solution, which is to introduce terms that are logically unsound but use our knowledge about the properties of a coproduct category to guarantee that whenever they are used, they reduce to something sound.

We begin by defining the disjoint sum of objects. There are several ways to encode a disjoint sum in `Lambdapi`¹⁸. We decided to use the fact that a sum type is isomorphic to the dependent product of a type with two constructors and a type that reduces to either of the two types depending on the first value. [figure 2.10](#) shows the definition of those two types. We chose this product as the definition of the coproduct because it made the definition of the unsound terms easier to manipulate in a uniform manner.

The category and its object type can then simply be defined as shown in [figure 2.11](#).

To define morphisms, we simply need to define two injections, from the morphisms of each category to the corresponding morphism type of the coproduct category. However, this would make formulating the composition too difficult; instead we find a way to create a unique injection parameterized by an index in $T2$ that chooses which category

18. The most common method would be to have two different functions for the two inductions and an elimination term that is essentially a match.

Remark 2. In [figure 2.10](#), we define a type $T2$ with two constructors. We could have reused $\text{Fin } 2$, because these two types are isomorphic. We used another type because it makes it more concise to match on the terms in it. `i1` is a lot easier to write and read than `finS (fin0 0)`. However, using $\text{Fin } 2$ would have made it clear that we can actually redo this entire development to support arbitrary finite sums by parameterizing with an integer n and replacing $T2$ with $\text{Fin } n$.

```

constant symbol T2 : TYPE;
constant symbol i0 : T2;
constant symbol i1 : T2;
symbol TFam : Cat → Cat → T2 → Cat;
rule TFam $C _ i0 ↪ $C
with TFam _ $D i1 ↪ $D;

```

Figure 2.10 Definition of the helper types for the coproduct.

```

constant symbol CPCat : Cat → Cat → Cat;
symbol ι {C D : Cat} (i : T2)
  : Ob (TFam C D i) → Ob (CPCat C D);
symbol π0 {C D : Cat} : Ob (CPCat C D) → T2;
symbol π1 {C D : Cat} (x : Ob (CPCat C D))
  : Ob (TFam C D (π0 x));
rule π0 _ _ (ι $i _) ↪ $i
with π1 _ _ (ι _ $x) ↪ $x;

```

Figure 2.11 Definition of the objects of the coproduct category.

it injects from using this index. To do so, we define a helper type HFam that does the choosing. The definitions are shown in [figure 2.12](#).

```

symbol HFam {C D : Cat} (i : T2)
  : Ob (TFam C D i) → Ob (TFam C D i) → TYPE;
rule @HFam $C _ i0 $a $b ↪ @Mph $C $a $b
with @HFam _ $D i1 $a $b ↪ @Mph $D $a $b;
symbol ι' {C D : Cat} (i : T2) (a b : Ob (TFam C D i))
  : @HFam C D i a b
  → @Mph (CPCat C D) (@ι C D i a) (@ι C D i b);

```

Figure 2.12 Definition of the morphisms of the coproduct category.

The identities are easy to define safely. Indeed, given an object in the coproduct category, one simply needs to associate the identity in the category corresponding to the first projection of the object. We already have all the tools to do so, as shown in [figure 2.13](#).

```

rule @Id (CPCat $C $D) $a
  ↪ @Id (TFam $C $D (π0 $a)) (π1 $a);

```

Figure 2.13 Definition of the identities in the coproduct category.

Defining the composition is more complicated because it is parameterized by three objects and two morphisms in the coproduct category, each of which could have a different first projection. Of course, the properties guarantee that they are all equal, but we could not find a way to make them definitionally equal. To address this, we created an unsafe projection, which allows us to choose the index on which we want to project (see [figure 2.14](#)). This allowed us to use objects and morphisms in the coproduct as if they were from a specific category. While the type system cannot guarantee this, we still know it to be true because of [proposition 1](#), so when executed, only safe terms will be produced.

Proposition 1. *Given two object a and b in $Ob (CPCat C D)$, if the type $@Hom (CPCat C D) a b$ is inhabited by any f , then we can construct a proof of equality between $\pi0 a$, $\pi0 b$ and $\pi0 f$. In other words, a , b and f come from the same category.*

```

symbol  $\pi U$  {C D : Cat} (i : T2)
  : Ob (CPCat C D)  $\rightarrow$  Ob (TFam C D i);
rule  $\pi U$  $i ( $\iota$  $i $a)  $\hookleftarrow$  $a;
symbol  $\pi U'$  {C D : Cat} {a b : Ob (CPCat C D)} (i : T2)
  : @Hom (CPCat C D) a b
   $\rightarrow$  @Hom (TFam C D i) ( $\pi U$  i a) ( $\pi U$  i b)
rule  $\pi U'$  $i ( $\iota'$  $i _ _ $m)  $\hookleftarrow$  $m;

```

Figure 2.14 The unsafe projections for objects and morphisms.

Remark 3. *The reason πU is so unsafe is that, given two categories C and D , it can create an object in $Ob\ C$ from an object in $Ob\ D$, by projecting onto the wrong index. For example, if C is $Fin\ 0$, the empty category, and D is $Fin\ 1$, it can be used to create a term of type $Ob\ (Fin\ 0)$, which is the empty type. Ideally the user of our library should never use this projection, and when we use it, care must be taken to ensure that it is only used in a safe context, where meta knowledge about the object used allows us to guarantee that the projection used is actually the right one, even if the type system is too weak to give us an internal guarantee.*

This unsafe projection allows us to write the composition as if we had internal knowledge that the morphisms were in the same source category, and thus reuse the composition of this category. The exact definition is provided in [figure 2.15](#).

```

rule @Comp (CPCat $C $D) $a $b $c $f $g
   $\hookleftarrow$  @Comp (TFam $C $D ( $\pi 0$  $a))
    ( $\pi 1$  $a)
    ( $\pi U$  ( $\pi 0$  $a) $b)
    ( $\pi U$  ( $\pi 0$  $a) $c)
    ( $\pi U'$  ( $\pi 0$  $a) $f)
    ( $\pi U'$  ( $\pi 0$  $a) $g);

```

Figure 2.15 Definition of the composition in the coproduct category.

Extensional type
theory in
Lambdapi

II

3 Martin-Löf Type Theory

In [section 1.3](#), we observed that a common difficulty in formalizing category theory in dependently typed proof assistants is dealing with the interaction of *definitional* and *propositional* equalities with the type system. We attempted to remediate this problem in [chapter 2](#) by using the extensibility of definitional equality in `Lambdapi`. This turned out to be more complicated than we had hoped. An extreme solution would be to use *extensional type theory*, which collapses the two kinds of equality together.

Of course, precisely understanding the difference in expressive power between intentional and extensional Martin-Löf type theories is not trivial. To do so, we proceeded in three steps. In this chapter, we discuss intentional type theory, and the structure of the two equalities. In the next chapter ([chapter 4](#)), we will examine the definition of extensional type theory, how it differs from intentional type theory, and how their expressivenesses are related. Finally, we will look at a procedure we implemented that transforms a proof in extensional type theory into a proof in intentional type theory in `Lambdapi`. Since this is the ultimate goal of this chapter, both the intentional and extensional type theory sections will be concluded by a discussion of the implementation of the related type theory in `Lambdapi`.

3.1 Definition

The syntax of Martin-Löf type theory is that of the usual λ -calculus with dependent sum, dependent product, equality types¹, and a hierarchy of universes². The explicit definition of the syntax is given in [figure 3.1](#). The typing rules in [figure 3.2](#) describe the expected rules for a dependently typed lambda calculus using Russell universes³. We did not include rules for congruence, but they are the usual rules of reflexivity, symmetry, transitivity, context closure, and computation. The specific computational rules are described in [figure 3.3](#). They include the β reduction, projections for the dependent pairs, and elimination of the J combinator, which allows the computation on the reflexivity.

Under context Γ , we thus have two ways for the two terms t and u of type T to be equal. Either they are *definitionally equal*; that is, $\Gamma \vdash t \equiv u$ is derivable, or they are *propositionally equal*; that is, there exists a term p such that $\Gamma \vdash p : t =_T u$. Because definitional equality is defined by the congruence rules of the system, it cannot be extended. Furthermore, these rules are chosen to be decidable so that type checking remains decidable. On the other hand, because propositional equality is a type, it can be abstracted upon, allowing us to reason under the hypothesis that some terms are internally equal. However, the synthesis of a proof of equality, or any proof, is generally undecidable. However, this is to be expected, since many types do not have a decidable equality, and propositional equality aims to capture the expressive power of mathematical equality.

3.2 Equality

The distinction between these two kinds of equality continues to be the source of confusion when dealing with dependent types. The first

1. A big realization in further type theories was that the equality type could be defined as a specific inductive type and did not need to be hardcoded into the theory. However, for simplicity, because we do not include inductive types in our discussion, we will include it in the syntax and rules.

2. Many possible hierarchies of universes can be chosen when defining a Martin-Löf type theory. Any preorder with *max* and *successor* operators can annotate universes. To keep things simple, we use a hierarchy with an infinite linear ordering of universes. In other words, the universes are annotated using integers.

3. Russell universes means we consider the elements in the universes are directly the types themselves. It is distinguished from Tarski universes, where elements in universes are to be understood as type codes, with a function *El* associating the codes with types. Russell universes are the most commonly implemented, but we will see in our `Lambdapi` formalisation in [section 3.3](#) of Martin-Löf type theory that Tarski universes can be used to simulate polymorphism in a type system without it.

$$\begin{array}{l}
 t, u, w, p, P, T, U, \dots := \lambda(x : T), t \\
 | \quad t \ u \\
 | \quad x \\
 | \quad \Pi(x : T), U \\
 | \quad \Sigma(x : T), U \\
 | \quad \pi_1(t) \\
 | \quad \pi_2(t) \\
 | \quad (t, u) \\
 | \quad t =_T u \\
 | \quad \text{refl}_T(t) \\
 | \quad J \ T \ t \ P \ w \ u \ p \\
 | \quad U_s \quad \text{for } s \in \mathbb{N}
 \end{array}$$

Figure 3.1 Syntax of Martin-Löf type theory.

$$\begin{array}{c}
 \frac{}{\Gamma, x : T, \Delta \vdash x : T} \text{Var} \quad \frac{\Gamma \vdash T \equiv U \quad \Gamma \vdash x : T}{\Gamma \vdash x : U} \text{Conv} \quad \frac{}{\Gamma \vdash U_s : U_{s+1}} \text{Univ} \\
 \frac{\Gamma \vdash T : U_s \quad \Gamma, x : T \vdash U : U_{s'}}{\Gamma \vdash \Pi(x : T), U : U_{s \wedge s'}} \Pi \quad \frac{\Gamma, x : T \vdash u : U}{\Gamma \vdash \lambda(x : T), u : \Pi(x : T), U} \text{Abs} \\
 \frac{\Gamma \vdash t : \Pi(x : T), U \quad \Gamma \vdash u : T}{\Gamma \vdash t \ u : U[x \rightarrow u]} \text{App} \\
 \frac{\Gamma \vdash T : U_s \quad \Gamma, x : T \vdash U : U_{s'}}{\Gamma \vdash \Sigma(x : T), U : U_{s \vee s'}} \Sigma \quad \frac{\Gamma \vdash t : T \quad \Gamma \vdash u : U[x \rightarrow t]}{\Gamma \vdash (t, u) : \Sigma(x : T), U} \text{Prod} \quad \frac{\Gamma \vdash t : \Sigma(x : T), U}{\Gamma \vdash \pi_1(t) : T} \text{Proj1} \\
 \frac{\Gamma \vdash t : \Sigma(x : T), U}{\Gamma \vdash \pi_2(t) : U[x \rightarrow \pi_1(t)]} \text{Proj2} \\
 \frac{\Gamma \vdash T : U_s \quad \Gamma \vdash t : T \quad \Gamma \vdash u : T}{\Gamma \vdash t =_T u : U_s} = \quad \frac{\Gamma \vdash t : T}{\Gamma \vdash \text{refl}_T(t) : t =_T t} \text{Refl} \\
 \frac{\Gamma \vdash t : T \quad \Gamma, x : T, p : t =_T x \vdash P : U \quad \Gamma \vdash w : P[x \rightarrow t, p \rightarrow \text{refl}_T(t)] \quad \Gamma \vdash u : T \quad \Gamma \vdash p : t =_T u}{\Gamma \vdash J \ T \ t \ P \ w \ u \ p : P[x \rightarrow u, p \rightarrow p]} J
 \end{array}$$

Figure 3.2 Martin-Löf type theory typing rules (without the computation rule).

question that comes to mind is the relationship between these two equalities. At first glance, it seems that propositional equality is more expressive than definitional equality. Indeed, one can show that if two terms are definitionally equal, they are also propositionally equal.

Proposition 1. *Let t and u be term of type T under context Γ such that $\Gamma \vdash t \equiv u$. Then there exists p such that $\Gamma \vdash p : t =_T u$.*

Proof. Since $\Gamma \vdash t \equiv u$, by the congruence of the definitional equality, we can show that $\Gamma \vdash t =_T t \equiv t =_T u$. Thus, by the conversion rule, we have $\Gamma \vdash \text{refl}_T(t) : t =_T u$. $\square$

$$\begin{array}{c}
\frac{\Gamma \vdash t : T \quad \Gamma, x : T \vdash u : U}{\Gamma \vdash (\lambda(x : T), u) \ t \equiv u[x \rightarrow t]} \beta \\
\frac{\Gamma \vdash t : T \quad \Gamma \vdash u : U[x \rightarrow t]}{\Gamma \vdash \pi_1((t, u)) \equiv t} \quad \frac{\Gamma \vdash t : T \quad \Gamma \vdash u : U[x \rightarrow t]}{\Gamma \vdash \pi_2((t, u)) \equiv u} \\
\frac{\Gamma \vdash t : T \quad \Gamma, x : T, p : t =_T x \vdash P : U \quad \Gamma \vdash w : P[x \rightarrow t, p \rightarrow \text{refl}_T(t)]}{\Gamma \vdash J \ T \ t \ P \ w \ t \ \text{refl}_T(t) \equiv w}
\end{array}$$

Figure 3.3 Computation rules for the equivalence.

On the other hand, if we have $\Gamma = T : U_s, x : T, y : T, p : x =_T y$, then we immediately have $\Gamma \vdash p : x =_T y$. However, since x and y are variables, no computation rule can apply to them, and we cannot derive $\Gamma \vdash x \equiv y$.

3.2.1 Transport

Since propositional equality seems more powerful, why do we bother with definitional equality. The answer is twofold. First, because definitional equality is decidable, it does not need to be stored, as it can be rechecked every time. Because it computes, one can construct terms where any proof of equality would require an exponential amount of storage⁴, while with the definitional equality, it does not require anything. Of course, type checking still takes the same amount of time, because the computation still needs to be run.

The second part of the answer is based on the use of equality. If we have $\Gamma \vdash t \equiv u$, we can substitute t with u anywhere in any terms, and the terms would still have the same semantics and types. However, if we have only $\Gamma \vdash p : t =_T u$ the same does not hold. For example, if T and U are definitionally equal types, then any term that can be shown to be of type T can also be shown to be of type U by using the conversion rule, and vice versa. However, this is not true if T and U are simply propositionally equal. Thus, although propositional equality makes more terms equal, its use is more restricted.

One legitimate question is, what can we do with a propositional equality between two types? Let's assume we have $\Gamma \vdash p : T =_{U_s} U$. We can then create a function from T to U by eliminating this equality on the identity function, as shown in [figure 3.4](#). Furthermore, we can show that it is an *equivalence*, that is, a function with a pointwise inverse up to propositional equality⁵. This function, or equivalence, is called the *transport* along p , and is written p^* .

4. For example, most type systems have a restricted recursion operator, which allows one to write well founded recursive function on inductive types. One such function is the Ackermann function, which is famous because it is a very simple function that grows very fast, much faster than any exponential. Any proof about the values of this function would involve computing it, resulting in a very large proof.

5. This means that composing the function with its inverse does not give the identity function definitionally, but that for every object it is applied to, there is a propositional equality between the result and the object itself. In other words, there exists a term p such that $\vdash p : \Pi(x : T), g(f x) =_T x$.

$$\frac{\frac{\overline{\Gamma \vdash T : U_s} \quad \overline{\Gamma, x : U_s, p : T =_{U_s} x \vdash x : U_s} \quad \overline{\Gamma \vdash x : T} \quad \overline{\Gamma \vdash U : U_s} \quad \overline{\Gamma \vdash p : T =_{U_s} U}}{\overline{T : U_s, U : U_s, p : T =_{U_s} U, x : T \vdash J \ U_s \ T \ x \ x \ U \ p : U}} J \\
\frac{}{T : U_s, U : U_s, p : T =_{U_s} U \vdash \lambda(x : T), J \ U_s \ T \ x \ x \ U \ p : \Pi(x : T), U} \text{Abs}$$

Figure 3.4 From a propositional equality between types to a function.

Remark 1. Given two terms f and g such that $\Gamma \vdash f : \Pi(x : A), B$ and $\Gamma \vdash g : \Pi(x : A), B$, we write $f \approx g$ to say that the type $\Pi(x : A), f = g$ is inhabited. That is, f and g are point-wise propositionally equal. This is not the same as stating that f and g are equal. We discuss this topic in more detail in [section 3.2.4](#).

6. Due to the computation rule of the J operator, this property even holds definitionally.

Remark 2. The link transport gives between equalities between types in a universe and equivalences can be seen as a way to characterize the structure of the equality type itself on the universe. However, it is undecidable whether transport is itself an equivalence between equalities and equivalences on a universe. The Univalent foundations program[17] is actually the study of the consequences of adding the univalence axiom, which states exactly that.

Remark 3. UIP can be seen as a way to characterize the structure of equality on equality types, the same way transport could characterize the structure of equality on universe (see [remark 2](#)). However, it is incompatible with the univalence axiom. Indeed, while the univalent program attempts to propagate the structure of the equality type, taken as it is, to the other type formers, UIP makes the structure of the equality type trivial. The reason almost all developments use either of these axioms is that without them, equality has an undecidable structure, that cannot be used (since it may not be there), but must still be preserved (since it may be there). The univalence axiom forces the structure to exist, enabling the user to use it, while UIP removes it, saving the user the effort of preserving it.

The transport is an interesting function because it describes the way the equality acts on types. Furthermore, it is compatible with the groupoid structure of the equality :

- ◊ $\text{refl}_{U_s}(T)^* \approx \lambda(x : T), x^6$
- ◊ $(p^{-1})^* \approx (p^*)^{-1}$
- ◊ $(p \cdot p')^* \approx p'^* \circ p^*$

One of the main pain points is that without any additional axiom, propositional equality between types can only be used through transport. Thus, one has to insert transports everywhere in types to get things to typecheck. This is the reason why in Coq, for example, apparently trivial changes in definition can yield huge improvements in the ease of stating some theorems, because they can make some equalities definitional, removing the need for transports. In addition, transport in the statement of a theorem will make it much harder to apply, since now there is an equality proof *within the statement itself*.

3.2.2 Uniqueness of identity proofs

An axiom that will become relevant later is the *uniqueness for identity proofs* (UIP). It states that two proofs of the same equality are always equal :

$$\vdash \text{UIP}_s : \Pi(A : U_s), \Pi(a : A), \Pi(p_1 p_2 : a =_A b), p_1 =_{a=_A b} p_2$$

Uniqueness of identity proofs is independent from the rules of Martin-Löf type theory. Indeed, while set-based models validate it, its incompatibility with univalence means that groupoid models refute it. As an axiom, it usually does not have computational content, that is, it breaks *canonicity*. Canonicity is the property that every term computes down to a canonical form under the empty context, where the canonical form depends on the type and is the application of the constructor. For example, the canonical form for a Π type is lambda abstraction, that for an equality is the reflexivity...However UIP allows the creation of equalities that are not reflexivities, even under the empty context.

However, UIP is equivalent to *axiom K*, which states that all equalities between twice the same term are always the reflexivity :

$$\vdash K_s : \Pi(A : U_s), \Pi(a : A), \Pi(p : a =_A a), p =_{a=_A a} \text{refl}_A(a)$$

Proposition 2. If UIP holds, then axiom K holds.

Proof. K is a special case of UIP where $b = a$ and $p_2 = \text{refl}_A(a)$. □

Proposition 3. If axiom K holds, then UIP holds.

Proof. This direction is a bit more technical. The intuition is that the J combinator make the equalities *freely generated* from the reflexivities, so we use it to reduce the problem to a problem on loops, where we can apply axiom K.

Let us assume we are under a context Γ with a type $A : U_s$, two elements $a b : U_s$ and two equalities $p_1 p_2 : a =_A b$. We want to build a propositional equality between p_1 and p_2 . Let us consider the following term :

$$J A a (\lambda x p, \Pi(p' : a =_A x), p = p') (K A a) b p_1$$

It has type $\Pi(p' : a =_A b), p_1 = p'$, so we can simply apply it to p_2 to get the equality we want. $\square$

If they are logically equivalent, axiom K has more interesting meta theoretical properties. Indeed, if we extend the definitional equality with the rule in figure 3.5, we recover canonicity [18].

[18] Streicher, *Investigations into intensional type theory*, 1993

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash K A a \text{refl}_A(a) \equiv \text{refl}_{a=_A a}(\text{refl}_A(a))}$$

Figure 3.5 Computation rule for axiom K.

3.2.3 Heterogeneous equality

We talked a lot about definitional and propositional equality, but there is a third kind of equality that is sometimes interesting. Both definitional and propositional equality are *homogeneous*, in the sense that they can only compare objects of the same type. But sometimes it may make sense to compare object with different types, especially if the types are themselves propositionally equal.

Definition 1. Let $A B : U_s$ be two types and $a : A$ and $b : B$. We say a and b are *heterogeneously equal* if there is an equality $p : A =_{U_s} B$ such that there is another equality $p' : p^*a =_B b$. We then write $a =_{A=B} b$.

Proposition 4. *Heterogeneous equality is reflexive, symmetric and transitive.*

Proof. Let $a : A$ be a well typed term under a context Γ . Then we have :

$$\Gamma \vdash \text{refl}_A(a) : \text{refl}_{U_s}(A)^* a = a$$

Given $(p, p') : a =_B b$, we have $(p^{-1})^* p^{-1'} : (p^{-1})^* b =_A (p^{-1})^* p^* a$. Furthermore, we can construct a proof of $(p^{-1})^* p^* a = (p^{-1} \cdot p)^* a \equiv \text{refl}_{U_s}(A)^* a \equiv a$. By concatenating the first equality with the inverse of the second once, we get the proof of $b =_{B=A} a$.

Finally, given $(p_1, p'_1) : a =_B b$ and $(p_2, p'_2) : b =_C c$, we have $p_2^* p'_1 : p_2^* p_1^* a =_C p_2^* b$, $p'_2 : p_2^* b =_C c$, and we can construct a proof of $(p_1 \cdot p_2)^* a =_C p_2^* p_1^* a$. $\square$

Proposition 5. *If two terms of the same type are heterogeneously equal, and axiom K holds, then they are propositionally equal.*

Proof. If the types are the same, the equality between the type and itself in the heterogeneous equality can be proved equal to the reflexivity using axiom K. So we have $p^*y = \text{refl}^*y \equiv y$. And thus from the equality $x = p^*y$ we can construct an equality of type $x = y$. $\square$

Remark 4. *Heterogeneous equality is actually something very simple : it is the propositional equality of the type $\Sigma(A : U_s), A$. In general, one can show that the equalities of dependent pairs are dependent pairs of equalities. Thus, the dependent pair is one of the few type formers in basic Martin-Löf type theory that plays well with equality.*

3.2.4 Functional extensionality

Remark 5. *Functional extensionality is a way to structure the equality of product type. We have seen in remark 4 that equality already has a nice structure on dependent pairs, and in remark 3, we mention that univalence is a way to give structure to the equality of universes. These properties are interesting because they provide a systematic approach for proving equality depending on the type. To get those properties and maintain canonicity (so without axioms), more recent developments define equality not through the J eliminator, nor as a specific inductive type, but inductively on the structure of the type it is applied to [19, 20].*

[18] Streicher, *Investigations into intensional type theory*, 1993

[17] Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*, 2013

[19] Cohen, Coquand, Huber et al., *Cubical Type Theory: a constructive interpretation of the univalence axiom*, 2016

[20] Altenkirch, McBride and Swierstra, *Observational equality, now!*, 2007

We saw in remark 4 that the dependent sum interacts well with equality. However, this is not true for the dependent product. Indeed, equality between functions is not pointwise equality. This is often problematic, since it means that there is no other way to prove the equality of functions than syntactic equality, despite them having the same behavior. To solve this problem, the *functional extensionality* (FunExt) axiom is often added. It states that if two functions are pointwise equal, then they are equal. Its complete statement is in figure 3.6.

$$\vdash \text{FunExt} : \Pi(A : U_s), \Pi(B : \Pi(_ : A), U_{s'}), \Pi(f\ g : \Pi(x : A), B\ x), \\ \Pi(_ : \Pi(x : A), f\ x =_{B\ x} g\ x), f =_{\Pi(x : A), B\ x} g$$

Figure 3.6 Functional extensionality statement.

This axiom is compatible with basic Martin-Löf type theory, but also with almost all of its variants. However, in 1993, Streicher presented a model of intensional type theory without functional extensionality in his investigations [18]. Because it is often required, the fact that it does not have computational content and breaks canonicity has often been seen as a limitation of Martin-Löf type theory. It is not only compatible with but also a consequence of the univalent axiom [17], so attempts to create canonical type theories where univalence holds, like Cubical type theory [19], also create theories in which functional extensionality has computational content. Observational type theory [20] is also a dependently typed type theory with both UIP and FunExt computing.

3.3 Formalisation in Lambdapi

There are two approaches when embedding systems in Lambdapi. *Deep embeddings* are embeddings of the kind usually seen in system like Coq, where the syntax is an inductive tree, and then typing and computation are defined as propositions on top of it. On the other hand, the extensible nature of the computation in Lambdapi enables *shallow embeddings*, embeddings where the computation and typing relation of the embedded system are directly that of Lambdapi. In other words, the syntax of the embedded system is encoded as new terms in Lambdapi, and a term will typecheck in Lambdapi if and only if it would have been correctly typed in the source system. Furthermore, the computation in the source system is fully captured by the computation in Lambdapi.

Of course, this is not possible for all systems (especially for extensional systems, as discussed in the next section). However, for many variations of intensional dependent type theory, it is known [21]. The main difficulty in embedding Martin-Löf type theory is simulating the hierarchy of universes and polymorphism in a system that has none. Once this is in place, embedding different type formers becomes straightforward.

[21] Cousineau and Dowek, *Embedding Pure Type Systems in the lambda-Pi-calculus modulo*, 2023

3.3.1 Universes and polymorphism

As a prerequisite for defining universes, we must first define *sorts*. Because we are using a simple linear hierarchy of universes, we can simply use a type isomorphic to the naturals, defined with an initial element and a successor. To define the dependent product and dependent sum, we must also implement the *max* operation. We do so by induction on the arguments. The Lambdapi code is shown in [figure 3.7](#).

```
constant symbol Sort;
constant symbol s0 : Sort;
constant symbol ssucc : Sort → Sort;
symbol smax : Sort → Sort → Sort;
rule smax s0 s0 ↦ s0
with smax (ssucc $s) s0 ↦ ssucc $s
with smax s0 (ssucc $s) ↦ ssucc $s
with smax (ssucc $s1) (ssucc $s2) ↦ ssucc (smax $s1 $s2);
```

Figure 3.7 Lambdapi code for sorts.

We can create universe types indexed by sort by declaring a symbol U that is a function from `Sort` to `TYPE`. However, an element in U s is not a type. The idea here is to make them Tarski universes. Indeed, in our description of Martin-Löf type theory, we used Russell universes, that is, universes whose elements are directly types. In Tarski's approach, elements in the universes are merely *codes* for types, which are interpreted into types by an *element* function.

We do the same in Lambdapi. We add a function `El` from any universe to `TYPE`. Then, for all type formers in our theory, we add both a symbol in the corresponding universe, and a computation rule for `El` that interprets it in the corresponding Lambdapi type, if it exists. For now, the only type former that we have are the universe types, so we need codes for them, and the rules that interpret the code for a universe into the corresponding universe. The code is in [figure 3.8](#).

```
constant symbol U : Sort → TYPE;
symbol El (s : Sort) : U s → TYPE;
constant symbol u (s : Sort) : U (ssucc s);
rule El (u $s) ↦ U $s;
```

Figure 3.8 The universes with the element function.

The rule *Univ* in [figure 3.2](#) stating that a universe lives in the successor universe is encoded by the fact that the code for a universe belongs to the successor universe.

Polymorphism can now be supported because although we still cannot quantify over `TYPE`, we can quantify over type codes living in a universe. It has the added benefit of isolating the formalization from other symbols defined in Lambdapi : a type is only visible to the Martin-Löf type theory formalization if a corresponding type code is added to a universe.

3.3.2 Type formers

We must support three type formers : equality, dependent sum and dependent product. Only the dependent product has an equivalent

in Lambdapi; therefore, its treatment will be slightly different. For equality and dependent sum, we need to add functions that create type codes from the relevant arguments. These type codes will not be interpreted by El because there is no dependent sum nor equality type in Lambdapi. To capture their behavior, we need to add symbols for the J eliminator and the two projections, along with reduction rules that capture the computation of these terms. There is no particular difficulty in doing so, and the code is displayed in [figure 3.10](#) for the equality and in [figure 3.9](#) for the dependent sum.

```
constant symbol S (s s' : Sort) (A : U s)
  (B : El s A → U s') : U (smax s s');
constant symbol mkS (s s' : Sort) (A : U s)
  (B : El s A → U s')
  : Π(a : El s A), El s' (B x)
  → El (smax s s') (S s s' A B);
symbol proj1 (s s' : Sort) (A : U s) (B : El s A → U s')
  : El (smax s s') (S s s' A B) → El s A;
rule proj1 _ _ _ (mkS _ _ _ $x _) ↪ $x;
symbol proj2 (s s' : Sort) (A : U s) (B : El s A → U s')
  : Π(p : El (smax s s') (S s s' A B)),
  El s' (B (proj1 s s' A B p));
rule proj2 _ _ _ (mkS _ _ _ $y) ↪ $y;
```

Figure 3.9 Definition of the dependent sum in Lambdapi.

```
constant symbol eq (s : Sort) (A : U s)
  : El s A → El s A → U s;
constant symbol refl (s : Sort) (A : U s) (x : El s A)
  : El s (eq s A x x);
symbol J (s s' : Sort) (A : U s) (u : El s A)
  (P : Π(x : El s A), El s' (eq s A u x) → U s')
  (_ : El s' (P u (refl s A u)))
  (v : El s A) (p : El s' (eq s A u v))
  : El s' (P v p);
rule J _ _ _ _ $w _ (refl _ _ _) ↪ $w;
```

Figure 3.10 Definition of the equality type in Lambdapi.

Something more interesting happens for the dependent product. Since there is already a dependent product in Lambdapi, we want to reuse that. Therefore, we add a computation rule for El that interprets the code for a product as a Lambdapi dependent product. Then, we do not need to add any new reduction rule because the application and β -reduction are directly those of Lambdapi, so they are already present. The code is shown in [figure 3.11](#).

```
constant symbol P (s s' : Sort) (A : U s)
  (B : El s A → U s') : U (smax s s');
rule El _ (P _ _ $A $B) ↪ Π(x : El s A), El s' (B x);
```

Figure 3.11 Definition of the product type in Lambdapi.

Because we use the UIP and FunExt axioms in [chapter 5](#), we also add them. To add them we simply declare new symbols with the correct types. In the case of UIP, we instead declare K along with a reduction rule to encode the computational aspect of K.

Remark 6. A very nice aspect of shallow embeddings is that because we reuse the reduction and computation of Lambdapi itself, there is no need to redefine α renaming, De Bruijn indices, substitution, context handling... We can only focus on the high level computational properties of the encoded theory without having to worry about the low level stuff.

4 Extensional Type Theory

4.1 Definition

Extensional Type Theory has the same syntax as Martin-Löf type theory, and the same rules. It differs only by the addition of the *reflection rule*, shown in figure 4.1. [proposition 1](#) tells us that if two terms are definitionally equal, then they are propositionally equal. The extension rules assert the reverse, effectively collapsing the two equalities.

$$\frac{\Gamma \vdash p : a =_A b}{\Gamma \vdash a \equiv b}$$

Figure 4.1 Reflection rule

Remark 1. *One particularity of the reflection rule is that it has a major difference from all the rules in [figure 3.2](#) : the term p in the hypothesis does not appear in the conclusion. This means that whenever we need to prove the definitional equality of two terms during typechecking, we may need to infer a term p . However, determining whether two terms are propositionally equal is generally undecidable. Therefore, extensional type theory has an undecidable typechecking, explaining why so few proof assistants use it¹.*

Collapsing the two equalities has further reaching consequences that meet the eye. Indeed, it does not just take the union of terms that are definitionally and propositionally equal. Since proofs of equality can alternate between the propositional and definitional points of view, it can mix their properties, allowing to prove equality of things that were neither definitionally nor propositionally equal before.

Proposition 1. *UIP is a theorem in extensional type theory.*

Proof. We will instead prove that K is a theorem in extensional type theory. The idea behind this proof is that $A : U_s, a : A \vdash \lambda(x : A), \lambda(p : a =_A x), p =_{a=_A x} \text{refl}_A(a)$ is derivable. Indeed, under the lambdas, we have $p : a =_A x$ in the context, so by the reflection rule $a \equiv x$, and by the congruence property $a =_A x \equiv a =_A a$.

This property is trivial in the case that $x \mapsto a$ and $p \mapsto \text{refl}_A(a)$. So given an equality p between a and itself, we can use the J eliminator with the above property to obtain that p is indeed equal to the reflexivity.

□

Proposition 2. *FunExt is a theorem in extensional type theory.*

Proof. The idea behind this proof is that functional extensionality holds if all the equalities are definitional, due to the congruence of the definitional equality around the λ -abstraction. So we take our propositional hypothesis, use the reflection to make it definitional, use the congruence of the definitional equality and then construct back a propositional equality.

1. However, one can imagine a proof assistant where the typechecking itself is interactive. When stating a theorem, the proof assistant first adds proof obligations to make the statement of the theorem typecheck, and then adds a proof obligation for the theorem itself. This is the approach explored by Andromeda[22]. Another approach is to use a heuristic algorithm that attempts to automatically prove the equalities that appear during typechecking, and refuse those that cannot be solved.

[23] Hofmann, *Conservativity of equality reflection over intensional type theory*, 1996

[24] Oury, *Extensionality in the Calculus of Constructions*, 2005

[25] Winterhalter, Sozeau and Tabareau, *Eliminating reflection from type theory*, 2019

2. `XAst.l`

3. De Bruijn indices represent variables by counting the number of binders between the occurrence of the variable and the binder introducing it. It provides a canonical representation of terms with regard to α equivalence.

4. `DeBruijn.l`

[26] De Bruijn, *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*, 1972

5. `XTyping.l`

6. Inversion lemmas allows the extraction of information from the derivation. For example, such a lemma constructs a derivation `der  $\Gamma$  A (tsort s) (snext s)` from a derivation `der  $\Gamma$  t A`. See figure 4.2 for all the inversions.

Remark 2. *The type `der` does not exactly correspond to a well-typed derivation. Indeed, it only encodes such a derivation if the context is well formed, but does not guarantee it. Thus, most functions expecting a correct derivation actually take two arguments : a proof of the well formedness of the context and a derivation. We decided to weaken `der` this way for performance reasons. Indeed, if `der` included a proof of well formedness of the context, since this proof contains as many derivations as its length, each would also contain proofs of the prefix context, and so forth recursively. This leads to an explosion of the derivation size. We expect many subtrees to be similar, but experiments showed that `Lambdapi` failed to deduplicate them; therefore, by separating the two we cut the recursion and reduce the derivation memory footprint.*

7. `Inversions.l`

We have :

- ◇ $\Gamma \vdash p : \Pi(x : A), f\ x =_B g\ x.$
- ◇ By abstracting $\Gamma, x : A \vdash p\ x : f\ x =_B g\ x.$
- ◇ By reflection $\Gamma, x : A \vdash f\ x \equiv g\ x.$
- ◇ By congruence $\Gamma \vdash \lambda(x : A), f\ x \equiv \lambda(x : A), g\ x.$
- ◇ By η reduction $\Gamma \vdash f \equiv g.$
- ◇ By [proposition 1](#), we have p' such that $\Gamma \vdash p' : f =_{\Pi(x:A), B} g.$

□

Since all rules of Martin-Löf type theory are present and the syntax is the same, any term that can be written and typed in the latter can also be typed in extensional type theory. On the other hand, [proposition 1](#) and [proposition 2](#) show that there are propositions that are theorems in extensional type theory that cannot be proven in Martin-Löf type theory. Therefore, the next logical question is that of the expressiveness of extensional with regard to intensional type theory.

In 1996, Hofmann showed that extensional type theory is conservative over intensional type theory with UIP and FunExt [\[23\]](#). However, his proof was not constructive, and so did not provide a procedure to construct intentional proofs from extensional proofs. In 2005, as part of his PhD thesis, Oury provided a constructive proof of Hofmann's results [\[24\]](#). However, he needed an additional axiom in intensional type theory for the proof to go through. Finally, in 2019, Winterhalter, Sozeau, and Tabareau managed to fix this and provided a constructive proof [\[25\]](#), formalized in Coq, which computed on extensional type theory proofs. The next chapter describes our attempt to reimplement Winterhalter's approach in `Lambdapi`.

4.2 Embedding in `Lambdapi`

Because the reflection rule leads to undecidable type checking, we cannot encode it as reduction rules in `Lambdapi`. Therefore, this rules out the possibility of using a *shallow embedding*. Instead, we will do a *deep embedding*, which is a pretentious way of saying that we will describe the syntax and typing derivations using inductive types.

The syntax is encoded as a single inductive type, following the syntax described in [figure 3.1](#)². The main difference is that we use De-Bruijn indices³ for the variables to avoid dealing with α renaming⁴. To define the derivations, we must implement the substitution and lifting [\[26\]](#).

The type of derivations is a bit more interesting. It is a mutually inductive type for three types of derivations : one for the contexts, one for the terms, and one for the definitional equality⁵. The context is implemented as a list of terms. The type `der_context : Context  $\rightarrow$  Prop` asserts that each term is a well formed type in the prefix context.

The most interesting type is `der : Context  $\rightarrow$  Term  $\rightarrow$  Term  $\rightarrow$  Sort  $\rightarrow$  TYPE`. Its values represent derivations showing that the first term is well typed of type the second term under the context. We also annotate it by the sort of the type because it makes some inversion lemmas⁶ easier to state.

For the same reason, the type `der_eq` for definitional equality takes not only the two terms it asserts convertible, but also their common type and the sort of the type. This enables us to write inversion lemmas⁷ that from a derivation of convertibility constructs a typing derivation for each term and type.

```
symbol inv_sort [ $\Gamma$  : Context] [s : Sort] [t T : Term]
  : der_context  $\Gamma$   $\rightarrow$  der  $\Gamma$  t T s  $\rightarrow$  der  $\Gamma$  T (tsort s) (snext s);
symbol inv_eq_sort [ $\Gamma$  : Context] [s : Sort] [T t1 t2 : Term]
  : der_context  $\Gamma$   $\rightarrow$  der_eq  $\Gamma$  s T t1 t2  $\rightarrow$  der  $\Gamma$  T (tsort s) (snext s);
symbol inv_eq_t2 [ $\Gamma$  : Context] [s : Sort] [T t1 t2 : Term]
  : der_context  $\Gamma$   $\rightarrow$  der_eq  $\Gamma$  s T t1 t2  $\rightarrow$  der  $\Gamma$  t2 T s;
symbol inv_eq_t1 [ $\Gamma$  : Context] [s : Sort] [T t1 t2 : Term]
  : der_context  $\Gamma$   $\rightarrow$  der_eq  $\Gamma$  s T t1 t2  $\rightarrow$  der  $\Gamma$  t1 T s;
```

Figure 4.2 The inversions lemmas.

5 Translation from ETT to MLTT

Extensional Type Theory is convenient for writing terms without having to worry about the distinction between definitional and propositional equality. However, its poor metatheoretical properties, especially the undecidability of its typechecking, mean that most proof assistants are based on Intentional Type Theory. As discussed in [section 4.1](#), if we add two axioms to intentional type theory, extensional type theory does not have additional expressive power. Furthermore, in a 2019 paper, Winterhalter, Sozeau and Tabareau [25] constructed a procedure that transforms extensional derivations into intentional derivations in Coq, enabling the local use of extensional type theory in Coq.

We reimplemented the ideas of this approach in `Lambdapi`. The motivation was twofold. First, since our previous approach to creating strict categories in `Lambdapi` turned out to be more limited than expected (see [section 2.2](#)), this approach could allow the development of a formalization of category theory directly in extensional type theory, and then translate it into a more commonly used intensional type theory¹. The second motivation was to use this translation as a middle step in encoding Cubical Type Theory [19] in `Lambdapi`. Indeed, Cubical comes with a decidable definitional equality that is too complex to encode as a rewriting system. However, first converting it into two level type theory [27] and then using our conversion to convert the extensional outer equality into an intentional one would be a feasible approach².

5.1 Intuition

Because typechecking an extensional term is undecidable, we must take the complete typing derivation as input. Then, we construct a well typed ITT term that has the *same semantics*. It is not clear how to formalize what it means to have the same semantics. The main thing to notice is that the resulting term is also typeable in ETT (because ETT is ITT with an additional rule). Thus the property is that the resulting term is equal to the initial term in ETT.

The intuition is very similar to the *double negation translation* [28], which takes a term that has a proof in classical logic, and constructs a term that is provable in intuitionistic logic in such a way that the resulting term is logically equivalent to the initial one *in classical logic*. The operation that is applied is the *double negation*, which is inserted on all operators or quantifiers that have different properties between classical and intuitionistic. Because double negation does not change provability in classical logic, the resulting term is equivalent in classical logic.

The operation we use in our translation is the transport, described in [section 3.2.1](#). The main idea is that if we have $\Gamma \vdash_{ETT} A \equiv B$ in extensional type theory, we can construct a proof of equality $\Gamma \vdash_{ITT} p : A =_{U_s} B$ valid in intentional type theory. Then, whenever the conversion rule is used in the typing derivation of a term t , we replace it with the term p^*t to remove the need for conversion. The subtlety comes from the fact that we may inject transports in term that also

[25] Winterhalter, Sozeau and Tabareau, *Eliminating reflection from type theory*, 2019

1. Because our focus shifted to the development of an interface to perform category theory proofs graphically, which we describe in [part III](#), we did not explore this possibility. We will return to this in the future.

[19] Cohen, Coquand, Huber et al., *Cubical Type Theory: a constructive interpretation of the univalence axiom*, 2016

[27] Annenkov, Capriotti, Kraus et al., *Two-Level Type Theory and Applications*, 2019

2. Nicolas Margulies, a new PhD student at Deducteam, is now pursuing this direction of research.

[28] Glivenko, *Sur quelques points de la logique de M. Brouwer*, 1929

appear in types, and dealing with the consequence of the cascade of transport injection.

Proposition 1. *Let $\Gamma \vdash p : A =_{U_s} B$ and $\Gamma \vdash x : A$ be two derivable sequents in *Extensional Type Theory*. Then $\Gamma \vdash x \equiv p^*x$ is also derivable.*

Proof. Since $\Gamma \vdash p : A =_{U_s} B$, then $\Gamma \vdash A \equiv B$, so $\Gamma \vdash A =_{U_s} B \equiv A =_{U_s} A$. By the conversion rule, we have $\Gamma \vdash p : A =_{U_s} A$, and by using the K rule (which is derivable in extensional type theory), we can derive $p = \text{refl}_A$. So $p^*x = \text{refl}_A^*x \equiv x$. $\square$

Proposition 1 tells us that, like the double negation preserves the meaning in classical logic, transport preserves the meaning of terms in extensional type theory. Therefore, injecting transports in a term does not change its meaning in extensional type theory, but can make it typeable in intensional type theory.

5.2 Structure

The translation comprises two mutually inductive predicates. First, τ_T^3 , takes an extensional derivation typing a type of sort s , and constructs an object in $\mathcal{U}^s$. As we are using a shallow embedding for intensional type theory, this translation could be interpreted as introducing a new *type former*, indexed by extensional derivations, in our theory. Of course, this symbol *computes*, meaning it will reduced to the types already described in [section 3.3](#).

Following this analogy, the translation itself, which takes a derivation d for a term and constructs a term in $\varepsilon(\tau_T(\text{inv_sort } d))$, acts as a *constructor* for the previously described type former τ_T . Again, type formers and constructors are constants, whereas here, the translation computes and fully reduces to the syntax of intensional type theory. Another interesting aspect of the translation is that we have two mutually inductive functions, with one constructing the type⁵ of the other. This is a special case of *induction-recursion* [29], which is more powerful than usual mutual induction. `Lambdapi` has no explicit support for induction recursion (or mutual inductive for that matter), but it allows the addition of many reduction rules to a term, in any order, as long as each step the system is confluent and terminating. Thus, the terms can be declared in a file, without any rule, and the rules are added in another file⁶ in any order that allows each rule to be typechecked.

Until now, we have discussed how to translate closed terms; however when recursing on the term structure, we will have to handle binders. Because we are translating into a shallow embedding in which the lambda abstraction is that of `Lambdapi`, we can simply reuse the variable binding of `Lambdapi` to introduce an intentional variable of the right type (see [figure 5.2](#) for an example). However, once we have this variable, we need some structure to store it when recursing, so that it can be used in the right places. To do so, we introduce the concept of a *translation context*, or `TContext`. A `TContext` is a list of extensional terms, along with derivations showing that they are well formed types, and intentional values with type the translation of the

3. `TranslationDefinition.lp`

4. In fact, we used different but isomorphic types for the extensional and intensional sorts. Therefore, we implemented a function `τ_st` that translates the extensional sorts into the intensional sorts. Because it is not really interesting, we will often skip it when displaying terms to avoid obscuring our meaning in syntactic details.

5. More precisely the type code, because we are using Tarski universes.

[29] Dybjer, *A general formulation of simultaneous inductive-recursive definitions in type theory*, 2000

6. `Translation.lp`

derivation. From such a context, we can extract an intentional value of the right type at an index, extract an extensional context, and extract a proof of well formedness of the extensional context⁷. Extraction of the extensional context is done by the τ_Γ function.

The translations are then parameterized by a translation context to handle the recursive case. The complete type of the translation is presented in figure 5.1.

```
constant symbol TContext : TYPE;
symbol  $\tau_\Gamma$  : TContext  $\rightarrow$  Context;
symbol  $\tau_d\Gamma$  ( $\Gamma$  : TContext) : der_context ( $\tau_\Gamma \Gamma$ );
symbol  $\tau_T$  ( $\Gamma$  : TContext) {T : ETT.Term} {s : ETT.Sort} :
  der ( $\tau_\Gamma \Gamma$ ) T (ETT.tsort s) (ETT.snext s)  $\rightarrow$  U ( $\tau_s s$ );
symbol  $\tau$  ( $\Gamma$  : TContext) {s : ETT.Sort} {t T : ETT.Term}
  (d : der ( $\tau_\Gamma \Gamma$ ) t T s) :
  let sort = I.inv_sort ( $\tau_d\Gamma \Gamma$ ) d in
   $\varepsilon$  ( $\tau_s s$ ) ( $\tau_T \Gamma$  sort);
```

Figure 5.1 The translation types.

```
rule  $\tau \$\Gamma$  (der_abs _ $s _ $A _ _ $dA _ $dt)
   $\hookrightarrow$   $\lambda a$ ,  $\tau$  (TPush  $\Gamma$  $s $A $dA a) $dt;
```

Figure 5.2 The abstraction case of the translation.

5.2.1 Comparison to other approaches

There are three approaches to compare with. The first, by Hofmann [23], used model arguments, and is purely on paper. Thus, while the proven result is the same, there is not much to compare. The second, by Oury [24], is again only on paper and necessitates one more axiom in ITT to go through. However, it improves over Hofmann's approach by providing a constructive proof. Finally, the approach by Winterhalter, Tabareau, and Sozeau [25] is an implementation in Metacoq of the translation and requires only UIP and functional extensionality in the target language.

Most of the ideas for the structure and algorithm implemented were taken from the Metacoq approach. The main difference is the embedding of the intentional type theory. In their approach, they used a deep embedding for the intentional part, which shares the syntax of the extensional embedding. This purely syntactical approach enables them to prove, for example, that the result of the translation is exactly the same term as the source, with some transports injected. Such a lemma is beyond our reach because the term we produce is directly a `Lambdapi` term and thus cannot be inspected from within `Lambdapi`.

However, in addition to constructing the translated term, they also had to construct a typing derivation for the constructed term that proved that it was well typed. In many parts of the proof, this represents the main complexity, so it obfuscates the translation by having this construction mixed in. In our approach, because we are using a shallow embedding, if `Lambdapi` accepts our construction, then it is directly well typed; therefore, we outsource all the typechecking steps to `Lambdapi`. This means that the translation code is more concise in our approach, despite the fact that we wrote the lambda terms directly⁸.

7. `TContext.lp`

[23] Hofmann, *Conservativity of equality reflection over intensional type theory*, 1996

[24] Oury, *Extensionality in the Calculus of Constructions*, 2005

[25] Winterhalter, Sozeau and Tabareau, *Eliminating reflection from type theory*, 2019

8. Our approach has its own set of downsides when writing the code. Indeed, while the Metacoq approach allows the user to construct the derivation interactively using tactics, in `Lambdapi`, we have to write well typed terms on the first attempt, and in case of failure we are left with a large error that takes time to understand. Thus, although the constructed term is more readable, writing it may require more effort.

5.3 Translating the congruence

Looking back at [section 4.2](#), we notice that the typing derivation is mutually defined with the equivalence derivation. Therefore, the translation must also interpret these derivations. These derivations appear in the conversion rule, so we must interpret them as something that makes sense for the translation of the conversion rule to go through. Intuitively, we explained that from the equivalence derivation, we want to construct an equality between the two types. Therefore, the first approach would be to have a translation that takes an equivalence between two types and produces an equality.

However, since we are in a dependently typed system, terms may appear in types, and looking at the inductive structure of the equivalence derivation, there are indeed cases where we use an equivalence between terms to construct an equivalence between types. Therefore, the translation of equivalence should produce equalities between the translated terms.

However, when attempting to formulate the type of this translation, there is a problem. We want an equality between the translations of the two terms. However, we do not translate terms but derivations. The inversions `inv_eq_t1` and `inv_eq_t2` can be used to obtain derivations for the two terms in an equivalence. However, the translated terms *do not have the same type*. Indeed, their types are translations of the equivalence type, but the actual derivations are not the same. Indeed, the first is obtained by inverting the result of `inv_eq_t1`, and the second by inverting the result of `inv_eq_t2`.

While these two types are not the same⁹, we show in the next section that they are equal. We have seen in [section 3.2.3](#) a way to compare the terms of different but propositionally equal types : heterogeneous equality¹⁰. Thus, we can write a translation that takes an equivalence derivation and constructs an heterogeneous equality between the translations of the terms.

This is not sufficient, because while we can transport along an equality, we cannot transport along an heterogeneous equality. However in the case of types we can recover a simple equality thanks to two facts :

- ◇ As seen in [proposition 5](#), under K, if two term are heterogeneously equal and they have the same type, then they are propositionally equal.
- ◇ In the case of types, their own types are universe, and the translation of a universe is always a universe. So in this case the heterogeneous equality is indeed between two terms of the same type.

Therefore, in addition to the translation of equivalence of terms that constructs a heterogeneous equality¹¹, we also create a helper translation on types that use the previous one and constructs a propositional equality. The final types are shown in [figure 5.3](#).

5.4 Fundamental lemma

There is one particular lemma that allows tying all the parts we have mentioned until now together, and that is also a sanity check about the correction of our implementation. We call it the *conversion lemma*. It states that if we have two derivations of the same term, then the result of the translations of those two derivations are equal¹². Of course, this

9. We cannot add rewriting rules to make them equal because, in practice, they may reduce to different types, and so such a rule would break confluence.

10. `HEq.lp`

11. `EqualityTranslation.lp`

12. In a way, this lemma enables us to talk about the translation of a *term*, despite the fact that we really translate derivations, because it shows that, up to heterogeneous equality, the derivation itself does not really matter.

```

symbol  $\tau\_eq\_type$  ( $\Gamma : TContext$ ) { $s : ETT.Sort$ } { $T\ t1\ t2 : ETT.Term$ }
  ( $d : der\_eq\ (\tau\_f\ \Gamma)\ s\ T\ t1\ t2$ ) :  $U\ \_ =$ 
  let  $d1 = I.inv\_eq\_t1\ (\tau\_d\ \Gamma)\ d$  in
  let  $d2 = I.inv\_eq\_t2\ (\tau\_d\ \Gamma)\ d$  in
  let  $ds1 = I.inv\_sort\ (\tau\_d\ \Gamma)\ d1$  in
  let  $ds2 = I.inv\_sort\ (\tau\_d\ \Gamma)\ d2$  in
  HEq' ( $\tau\_s\ s$ ) ( $\tau\_T\ \Gamma\ ds1$ ) ( $\tau\_T\ \Gamma\ ds2$ ) ( $\tau\ \Gamma\ d1$ ) ( $\tau\ \Gamma\ d2$ );
symbol  $\tau\_eq$  ( $\Gamma : TContext$ ) { $s : ETT.Sort$ } { $T\ t1\ t2 : ETT.Term$ }
  ( $d : der\_eq\ (\tau\_f\ \Gamma)\ s\ T\ t1\ t2$ ) :  $\varepsilon\ \_ (\tau\_eq\_type\ \Gamma\ d)$ ;
symbol  $\tau\_eqT$  ( $\Gamma : TContext$ ) { $s : ETT.Sort$ } { $T1\ T2 : ETT.Term$ }
  ( $d : der\_eq\ (\tau\_f\ \Gamma)\ (ETT.snxt\ s)\ (ETT.tsort\ s)\ T1\ T2$ ) :
  let  $d1 = I.inv\_eq\_t1\ (\tau\_d\ \Gamma)\ d$  in
  let  $d2 = I.inv\_eq\_t2\ (\tau\_d\ \Gamma)\ d$  in
   $\varepsilon\ \_ (eq\ \_ (u\ (\tau\_s\ s))\ (\tau\_T\ \Gamma\ d1)\ (\tau\_T\ \Gamma\ d2))$ ;

```

Figure 5.3 The types of the equivalence translation.

equality is heterogeneous because the types of the translations are only propositionally equal.

Despite its usefulness, it was too weak to prove itself by recursion. We needed to strengthen it in two different ways before all the recursive cases worked.

The first case involved handling the situation in which one of the derivations was a type conversion. In this case, we cannot really call the conversion lemma recursively because we have two derivations typing the same term, but with two different types. However, because we have an equivalence derivation between the two types, we can construct a propositional equality between them, so we know that they are actually similar. To address this, we change our conversion lemma to accept derivations about the same term but not necessarily the same type.

The other difficulty arises from variable binding. For example, in the case of abstraction, because the two derivations can be different, the derivation typing the abstracted variable can also differ. Therefore, we must bind two variables that may have different types. Again, by recursively applying the lemma, we know that the two types are propositionally equal. It turns out that it is enough for the case to go through to bind two variables and a proof of their heterogeneous equality. However, we now have two different contexts, since we have bound two variables, one for each derivation.

To address this, we introduce the concept of a *packed context*, or **PContext**. A packed context should be understood as two translation contexts that have the same underlying context¹³ and, for each pair of intentional values, a heterogeneous equality between them. Then, we update the conversion lemma to take a packed context and construct an equality between the translation of the first derivation under the first translation context in the packed context, and the translation of the second derivation under the second translation context.

The final type is shown in figure 5.4. It is a bit technical to understand, but its full power was necessary for it to prove itself, and actually turned out to be useful in proving other lemmas.

13. This means they have the same length and extensional types, but the derivations and values are duplicated, and may not be the same.

```

constant symbol PContext : TContext → TYPE;
constant symbol PEmpty (Γ : TContext) : PContext Γ;
constant symbol PCPush
  {Γ : TContext} {s : ETT.Sort} {A1 A2 : ETT.Term} (P : PContext Γ)
  (dA1 : der (τ_Γ (projΓ1 P)) A1 (ETT.tsort s) (ETT.snext s))
  (dA2 : der (τ_Γ (projΓ2 P)) A2 (ETT.tsort s) (ETT.snext s))
  {a1 : ε (τ_s s) (τ_T (projΓ1 P) dA1)} {a2 : ε (τ_s s) (τ_T (projΓ2 P) dA2)}
  : H a1 a2 → PContext Γ;
symbol projΓ1 {Γ : TContext} : PContext Γ → TContext;
symbol projΓ2 {Γ : TContext} : PContext Γ → TContext;
symbol convertR (Γ : TContext) (h : PContext Γ) {s : ETT.Sort} {t T1 T2 : ETT.Term}
  (d1 : der (τ_Γ (projΓ1 h)) t T1 s)
  (d2 : der (τ_Γ (projΓ2 h)) t T2 s) :
  let ds1 = I.inv_sort (τ_dΓ (projΓ1 h)) d1 in
  let ds2 = I.inv_sort (τ_dΓ (projΓ2 h)) d2 in
  HEq (τ_s s) (τ_T (projΓ1 h) ds1) (τ_T (projΓ2 h) ds2)
  (τ (projΓ1 h) d1) (τ (projΓ2 h) d2);

```

Figure 5.4 The type of the conversion lemma.

5.5 Performance evaluation

14. `Examples.lp`

To evaluate the performance, we created some test cases. To do so, we wrote helper symbols to write extensional derivations manually¹⁴. The first problem we encountered was before the translation even had a chance to run. Indeed, as mentioned in [remark 2](#), our first approach used a slightly different type of extensional derivation, which included checking the well formedness of the context. This resulted in an explosion in the size of the derivation, causing Lambdapi to hang before it could compute the translation.

As one of the simplest closed terms, we first tried our translation on the polymorphic identity $\lambda(A : U_0), \lambda(x : A), x$, for which we created the straightforward derivation. While the derivation itself is valid in intentional type theory, there was a risk that the introduced transports did not reduce. However, the result was $\lambda A, \lambda a, a$, which is the best we could have hoped for, and it did so without requiring too much time.

Slightly more complex, we tried to translate the dependent function application :

$$\lambda(A : U_0), \lambda(B : A \rightarrow U_0), \lambda(f : \Pi(x : A), B x), \lambda(x : A), f x$$

The translation did not finish on this example. When investigating, it turned out that while the constructed term has the same shape as the derivation, and thus the same size, the inserted equalities can grow quite large. In this case, this is exactly what happens : the `funext` instances in the generated equalities prevent them from reducing; thus, the term grows increasingly larger until it causes Lambdapi to hang. Upon closer inspection, most instances of functional extensionality are applied to functions that are definitionally equal. To make the term compute more, we added the reduction rule in [figure 5.5](#), and indeed

with this rule the translation finished in reasonable time and gave us the expected $\lambda A B f x, f x$.

```
rule funext $s $s' $A $B $f $f _
  ↪ refl (smax $s $s') (P $s $s' $A $B) $f;
```

Figure 5.5 Computation rule for functional extensionality.

Now we should try on a term that only typechecks in extensional type theory.

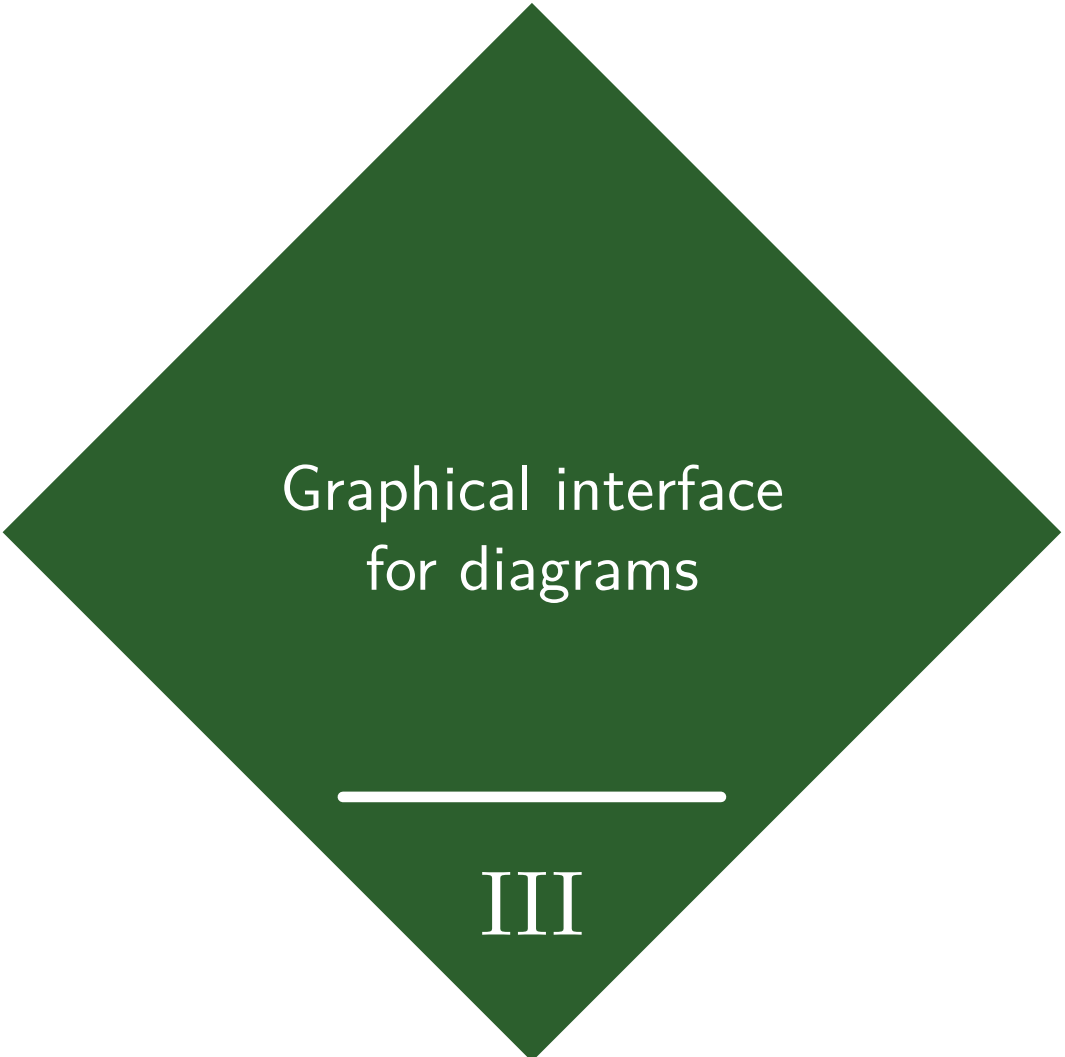
Until now, we have only translated terms that were already typeable in intensional type theory. We should now try one that necessitates the reflection rule. The simplest one we could think of is the transport. Its term is the identity function, but taking an equality between types in the context so that the result has a different (but propositionally equal) type :

$$\begin{aligned} &\vdash \lambda(A B : U_0), \lambda(p : A =_{U_0} B), \lambda(x : A), x \\ &\quad : \Pi(A B : U_0), A =_{U_0} B \rightarrow A \rightarrow B \end{aligned}$$

The result of the translation is the term $\lambda A B p x, (refl A \cdot refl A \cdot p) * x$. We would have hoped for the reflexivities to disappear, but because we implemented the transitivity of equality using the J combinator, it only reduces to the right. Although not perfect, the term we obtained is almost the expected one. Again, the execution time was not noticeable.

On larger examples, however, the translation never terminated. Indeed, the problem is the same as the second example, but without an easy workaround : the presence of atomic equalities, such as variables, or true uses of functional extensionality, results in equality terms that do not compute. Thus, the size of the equality term becomes too large for `Lambdapi` to handle. While we managed to avoid dealing with the non-computation of functional extensionality when it was trivially applied, we are still stuck in the other situations. And even if we translated to a theory where functional extensionality always computes, we would still be bothered by the equalities in the context, which cannot have computational content because they are only variables.

This problem could be mitigated by translating to an intermediate language, where the equality is able to compute more in the presence of free variables, simplifying the generated terms. However, in general, we expect this to be the main limiting factor when applying our translation method.



Graphical interface for diagrams

III

6 Demo

6.1 Workflow

We believe that the most interesting part of this work, rather than its theoretical foundations, which we describe in [chapter 7](#), is the fact that it is implemented and can actually be used in proofs. To reflect this, we will begin by providing an overview of what the tool looks like and how it can be used.

Currently, this plugin works only with the *UniMath* category library. Of course, porting to another library should be easy enough¹, but we have not yet found a satisfactory way to support multiples libraries simultaneously (see [section 12.4](#) for a more in depth discussion of this topic). Assuming our plugin is installed, a Coq development using it would start with the usual inclusions and the inclusion of the plugin itself :

```
From UniMath Require Import Preamble.
From UniMath Require Import Foundations.NaturalNumbers.
From UniMath Require Import CategoryTheory.Core.Categories.
From UniMath Require Import CategoryTheory.Core.Functors.
From CommutativeDiagrams Require Import Loader.
```

```
C : precategory
a, b, c, d : C
m1, m2, m3 : C [[ b, c ]]
m' : C [[ a, b ]]
m'' : C [[ c, d ]]
H1 : m1 = m2 · I
H2 : m3 = m2
f : C [[ a, b ]] → C [[ a, c ]]
Hf : ∀ m : C [[ a, b ]], f m = m · m1
=====
I · m' · (m3 · I · (I · m'')) · (I · I)
= f m' · (I · (I · I · m'' · I))
```

Figure 6.1 A Coq goal

[Figure 6.1](#) shows the goal we are considering. We see that we have in the context some morphisms, some hypotheses about which morphism is equal to which, and a function *f* that constructs new morphisms, along with a hypothesis *Hf* telling us the extension of this function. The goal itself is quite unclear, being polluted by an arbitrary association of morphisms, along with many unneeded identities.

This goal could be normalised by something as simple as :

```
repeat rewrite assoc.
repeat rewrite id_left.
repeat rewrite id_right.
```

However, we do not need to do that, because our plugin has its own normalization procedure that it will apply on start. More precisely, when started, the plugin looks into the proof goal to identify the objects and morphisms, and construct a graph from it. Therefore, instead of the previous tactics, we can start the plugin directly with :

```
diagram run "test.diag".
```

1. We actually changed the underlying category library from *HoTT* to *UniMath* before, and it was straightforward. Despite what the choice of these two libraries may imply, we do not depend on univalence and our plugin works exactly the same on non-univalent libraries.

The "test.diag" part of the tactic is the file used to save the interaction, enabling the replaying of proofs. We will discuss this in more detail in [section 6.2](#). When this tactic is executed, it opens a graphical interface, shown in [figure 6.2](#).

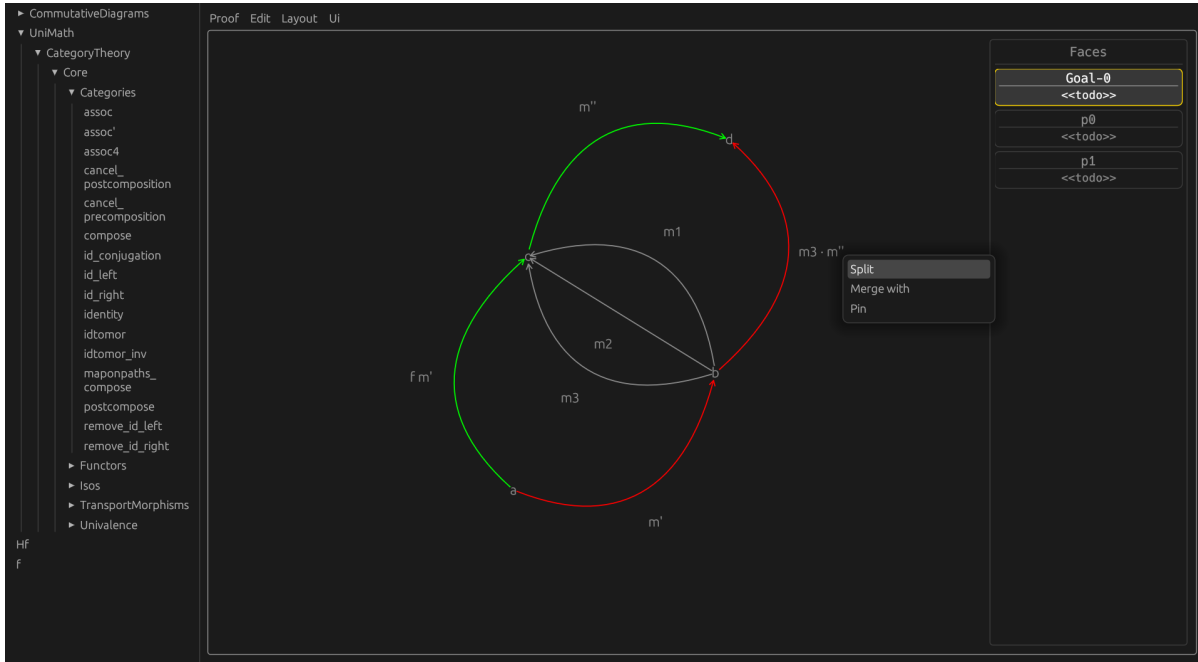


Figure 6.2 The plugin interface

This interface is split into three parts. The left column displays the tree of lemmas. We will go into more depth about lemmas in [chapter 8](#), but as a first approximation a lemma is a categorical object that is quantified, so you can see f and Hf in the lemma list. The graph is displayed in the center. The layout procedure is interactive, so the different objects can be dragged around, and the graph will adapt. Furthermore, one can right-click on the different objects to display a contextual menu with the different actions available. Finally, the equalities are displayed in the right column.

An equality is a Coq term whose type is an equality between two morphisms that have the same source and target. The exact structure of the graph with equalities is described in detail in [chapter 10](#). In the graph, the two sides are paths, since the left- and right-hand sides of the equality may be morphisms with compositions. When an equality is selected, its left-hand side is shown in red and its right-hand side in green. We can see that the first equality is selected because its background is a clearer gray, and looking at the red and green paths in the graph, we can conclude it is an equality between $m' \cdot m3 \cdot m''$ and $f m' \cdot m''$.

Furthermore, since the equality is surrounded in yellow, the equality is a goal, and not an hypothesis. Now that we can understand the interface, we see that what need to prove $m' \cdot m3 \cdot m'' = f m' \cdot m''$. Note that all identities are automatically hidden.

The interface exposes some operations to transform the graph and progress the proof, such as splitting an edge along compositions, splitting an equality along its planar decomposition, an automatic proce-

dures for faces... One can make significant progress using only these, but to conclude this proof using Hf is necessary to deal with f .

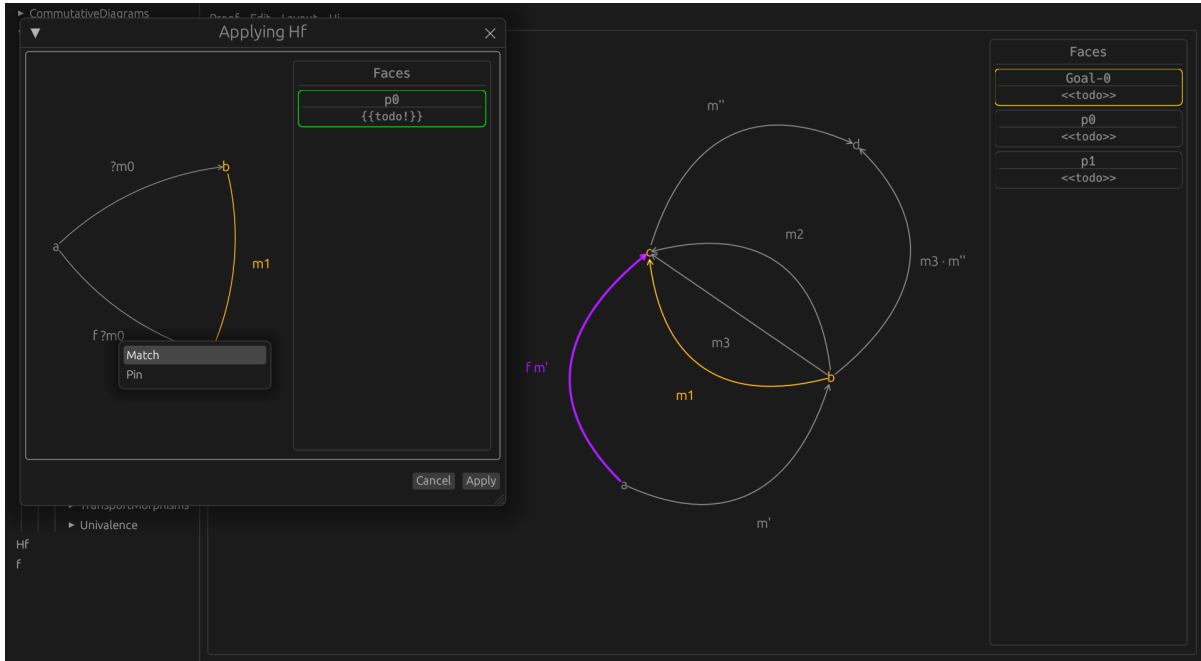


Figure 6.3 The lemma application window

Clicking on Hf opens the lemma application window, as shown in figure 6.3. The lemma is displayed as a graph, and the user can then match objects in the lemma graph with objects in the goal graph. Matched objects are displayed in orange, and the selected object is shown in purple. When matching, the underlying Coq terms are unified. Finally, when the lemma is applied, the pushout of the two graphs modulo the matching is computed, and the result is the new goal graph. A more precise description of this process is provided in chapter 8.

Finally, when the user is satisfied with what they have done in the interface, they can simply click on **Finish** in the **Proof** menu. The interface will then close, and the Coq state will be advanced such that any remaining proof obligation in the interface, if any, becomes a new Coq goal.

6.2 Replaying proofs

One thing that is not obvious is how this works when we want to replay the proof. This is a common difficulty of the graphical approach, as there is no obvious way to store a graphical interaction. The way we decided to do it is that we created our own very simple language to describe operations on the categoretical proof state. Subsequently, whenever the user interacts with the interface, an equivalent operation is added to the script. Finally, when the proof is terminated, the script is saved to a file.

We could have chosen a binary format to save the interactions, which would have probably been more efficient to parse and execute. However, we believe that there is value in a simple textual language that can

be read, shared, and written directly. Figure 6.4 shows an example of the script language we are using. Because it relies on names to refer to objects, and the names can only be found through the interface, its readability on its own is still limited.

```
merge mab mab_0
merge mcd_0 mcd
split mbd
decompose Goal-0 mab:<m3;m2>:mcd;mab:<m2;m1>:mcd
apply Hf b:b a:a mac:mac m1:m1 c:c mab:mab
solve Goal-0-0
succeed
```

Figure 6.4 An example of the plugin scripting language

There are two variants of the tactic. `diagram run` tries to run again a previously done proof, and only opens the interface on the first run or if the script failed. On the other hand, `diagram edit` always opens the interface, and if a previous proof is found, sets the interface such that `Edit > Redo` replays the different steps one by one, allowing changes to or visualization of the proof.

The question remains as to where to save the proof and where to look it up. We decided that every time the tactic is called, the user must give it as an argument a file name, and the script will be loaded from this file if it exists, and saved to it on success. This approach makes it easy to share the proof script with other users because it is simply another text file in the Coq development, and introduces no ambiguity as to where the proof comes from. However, this introduces many small files if the tactic is used in many places, and it makes reading the proof without executing it slightly more complicated because the script is in another file.

Another approach is to have the script directly inline in the Coq proof. We actually tried to implement it for a bit and got stuck due to limitations in Coq², but a bigger problem is that there is no way for a Coq plugin to change the Coq proof itself. Thus, we could either display the script at the end and remind the user to copy it inside the proof, and may create an integration with *coq-lsp*⁽¹¹⁾.

Yet another approach, which is the one taken by Actema⁽⁵⁾, is to store the proofs in a database indexed by a hash of the current proof state. The main advantage of this approach is that it is completely transparent to users. However, it is not possible to read the saved proof, which makes sense for Actema because the saved proof is not human readable, but is more problematic for our use case. Furthermore, sharing proofs through git becomes more complicated, and even a very simple change in the proof state where the tactic is called can result in Actema not finding the previous proof.

6.3 Comparison with other tools

There are many tools to work with different flavors of diagrams; however, there are mainly three that we are aware of that work directly with commutative diagrams, in addition to our own.

The first is Quiver⁽²⁾. Quiver is an online tool to draw diagram. As such, both the tools it supports and the range of diagrams it can

2. Or maybe a bug in Coq, we did not have the time to fully investigate it.

⁽¹¹⁾ Coq LSP

⁽⁵⁾ Actema : une interface graphique et gestuelle pour preuves formelles

⁽²⁾ Quiver: a modern commutative diagram editor

represent are much greater than what our work can do. However, it only deals with drawing diagrams and has no logic to perform proofs with them.

The second is a lean widget that displays diagrams^[6]. Similar to our tool, it is integrated into a proof assistant and can create a representation of a graph from the proof context. However, it is not interactive, and the proof cannot be progressed from it. It is only meant as a visualization helper. Furthermore, it was meant as a demo for the Lean widget system, but as far as we know, it is no longer developed.

The last is Lafont's diagram editor^[9]. It started as a tool similar to Quiver, more focused on drawing diagrams. However, it has evolved to become much more similar to our tool, integrating with Coq-LSP^[11] to enable interactive features. As it is now, it has more features to build a diagram, but the proof generation is less interactive. We expect our tools to converge on their abilities.

^[6] ProofWidgets

^[9] A diagram editor in your browser

^[11] Coq LSP

Remark 1. As discussed in *chapter 10*, our approach is organized so that it should be easy to port to a new proof assistant. However, this approach should also make it possible to easily change the integration with a proof assistant. For example, we could replace the Coq plugin with a Coq-LSP integration to use its additional features without changing the interface.

7 The diagram structure

7.1 The graph

To describe the operations available to edit the graph and progress the proof, we first need to describe precisely what we mean by *graph* and its relation to the proof. Using technical terms, by graph we mean an annotated 2-*quiver* (or equivalently 2-*globular set*), as shown in figure 7.1.

Let us go over these definitions step by step. A quiver (or equivalently a 1-globular set) is a directed multigraph, that is, a directed graph where there can be many edges with the same source and targets, and where there can be (potentially many) loops.

Definition 1. A quiver is a set N of nodes, along with a set E of edges and two functions $s, t : E \rightarrow N$ which indicate the source and target of each edge.

We need to relate this combinatorial structure to Coq terms. We assume we have the following types in Coq :

- ◇ `category` : Type a type for *categories*
- ◇ `object` : `category` $\rightarrow$ Type the type for objects of a category
- ◇ `morphism` : $\forall (C : \text{category}), \text{object } C \rightarrow \text{object } C \rightarrow \text{Type}$ the dependent type for morphisms

Of course for the interface to work we will need more structure on those types, but to define the graph this is all that is required. We write $\mathcal{E}_C$ the set of all Coq terms. Since the intent is to describe an implementation, they are not quotiented by $\beta\delta\iota$ reduction, meaning that this is the set of well typed syntactic terms. We assume that we have a function `typeof` : $\mathcal{E}_C \rightarrow \mathcal{E}_C$ mapping a term to its type¹. Because we have not quotiented by reduction, such a function is not unique, but we simply need $x : \text{typeof}(x)$ to be valid in Coq for the following to work.

Definition 2. An annotated quiver is a quiver $\mathcal{K}$ along with two functions $o : N_{\mathcal{K}} \rightarrow \mathcal{E}_C \times \mathcal{E}_C$ and $m : E_{\mathcal{K}} \rightarrow \mathcal{E}_C$ such that :

- ◇ $\forall n \in N_{\mathcal{K}}, \text{typeof}(\pi_2(o(n))) = \text{object } \pi_1(o(n))$
- ◇ $\forall e \in E_{\mathcal{K}}, \pi_1(o(s_{\mathcal{K}}(e))) = \pi_1(o(t_{\mathcal{K}}(e)))^2$
- ◇ $\forall e \in E_{\mathcal{K}}, \text{typeof}(m(e)) = \text{morphism } \pi_1(o(s(e))) \pi_2(o(s(e))) \pi_2(o(t(e)))$

Intuitively, this annotates nodes in the quiver with Coq terms corresponding to objects in a category and edges with morphisms in a manner that is consistent with both the structure of the quiver and that of the category.

7.2 Path and morphisms

A path in a quiver is a finite sequence of edges, such that the target of each edge is the same as the source of the following edge.

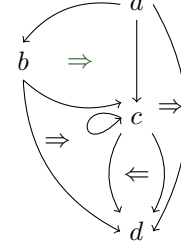


Figure 7.1 An example of a 2-quiver.

1. In the implementation, we use the Coq `pretyper` to implement this function. More precisely, we use the `Retyping.get_type_of` function of Coq.

2. This forces connected nodes to belong to the same category.

Definition 3. Given a quiver $\mathcal{K}$, we write $(P_n)_{n \geq 0}$ for the paths of length n . We also define two functions $s_n, t_n : P_n \rightarrow N_{\mathcal{K}}$ which respectively give the source and target of the path. The paths in $\mathcal{K}$ are then $P_{\mathcal{K}} = \bigcup_{n \geq 0} P_n$, with functions $s = \bigcup_{n \geq 0} s_n$ and $t = \bigcup_{n \geq 0} t_n$.

Remark 1. A corner case to consider is that of paths with length 0. There is exactly one for each node³, the same way there is one identity by object in a category.

Given a node $n \in N_{\mathcal{K}}$, we write $[n]$ for the empty path of source and target n . Given an edge in $\mathcal{E}_{\mathcal{K}}$, we also write $[e]$ for the singleton path consisting of the single edge e .

Definition 4. Given a quiver $\mathcal{K}$ and two nodes $a, b \in N_{\mathcal{K}}$, we write $\mathcal{K}(a, b) := \{p \in P_{\mathcal{K}} \mid s(p) = a \wedge t(p) = b\}$.

Definition 5. Given a quiver $\mathcal{K}$, we define the concatenation of paths $\cdot : P_{\mathcal{K}} \times_s P_{\mathcal{K}} \rightarrow P_{\mathcal{K}}$ by recursion on the first path⁴.

In a category, the composition operations means that every path in a diagram denotes a morphism. Let us assume we have the following objects in Coq :

- ◇ `comp : ∀{C : category}, ∀{a b c : object C},
morphisme C b c -> morphisme C a b -> morphisme C a c`
- ◇ `id : ∀{C : category}, ∀(a : object C), morphisme C a a`

Definition 6. Let $\mathcal{K}$ be an annotated quiver. We define a relation $\propto_{\mathcal{K}} \subseteq P_{\mathcal{K}} \times \mathcal{E}_C$ by the following equations :

- ◇ $\forall [n] \in N_{\mathcal{K}}, n \propto \text{id } o(n)$
- ◇ $\forall [e] \in E_{\mathcal{K}}, e \propto m(e)$
- ◇ $\forall p_1, p_2 \in P_{\mathcal{K}}, \forall m_1, m_2 \in \mathcal{E}_C, p_1 \propto m_1 \wedge p_2 \propto m_2 \implies p_1 \cdot p_2 \propto \text{comp } m_1 \ m_2$

If $p \propto m$, then m is a valid annotation for p . Intuitively, a valid annotation for a path p is the composition of all annotated morphisms of the edge of p , parenthesized in any way and potentially with identities composed in the middle. Of these we distinguish *the realization of p* as the morphism which is exactly the composition of the morphism in p associated to the right and with no identities⁵. If the path is empty, its realization is the identity over the annotation of the path source.

When dealing with categories in Coq, we have the associativity of the composition and the unitality with regard to identities as propositional equalities. This means that if $p \propto m$, we can construct an equality between m and the realization of p , normalizing m with regard to associativity and unitality.

During the following development, we will not differentiate between morphisms that are valid with regard to a path. We acknowledge that they are different in Coq, and in the implementation, we have to include equalities between them every time there is a mismatch, but since those equalities can be automatically generated, dealing with this when describing the features would only distract from the intuitions.

3. Having multiple empty paths is necessary for the source and target function to be well defined. However, from an implementation perspective, this means that a path cannot be stored simply as a list of edges, and one needs to store either the source, target or both.

4. This operation makes the set of paths in a quiver a category, with the objects being the nodes of the quiver and the empty paths the identities. More precisely, it makes it the *free category* generated by the quiver.

5. If one of the edges in the path is annotated by an identity, it is not included in the path realization.

7.3 Second level structure

This structure we have just defined corresponds to the intuitive notion of a diagram in a category. However, this is not sufficient to express what we want. When drawing diagrams, we often want to draw *commutative diagrams*, where the composition along any two parallel paths is equal. However, here we want to also represent intermediate states during the construction of such a graph. Therefore, we need to keep track of pairs of morphisms that are equal. Since we are integrating with Coq, we want the existence of a proof of propositional equality between the sides. To keep track of this, we need to have more structure on our quiver.

Definition 7. A 2-quiver (equivalently a 2 globular set) is a quiver $\mathcal{K}$ along with a set F and two functions $left, right : F \rightarrow P_{\mathcal{K}}$ such that :

$$\forall f \in F, s(left(f)) = s(right(f)) \wedge t(left(f)) = t(right(f))$$

We then define $s, t : F_{\mathcal{K}} \rightarrow N_{\mathcal{K}}$ to be $s := f \mapsto s(left(f))$ and $t := f \mapsto t(right(f))$ ⁶.

We refer to the elements of $F_{\mathcal{K}}$ as the *faces* of the quiver $\mathcal{K}$. To annotate them with proofs of equality, we assume that in Coq, here is an equality type $eq : \forall(T : \text{Type}), T \rightarrow T \rightarrow \text{Type}$ ⁷.

Definition 8. An annotated 2-quiver is a 2-quiver $\mathcal{K}$, such that the underlying quiver is annotated, and a function $\phi : F_{\mathcal{K}} \rightarrow \mathcal{E}_C$ such that :

$$\forall f \in F_{\mathcal{K}}, \text{typeof}(\phi(f)) = \begin{array}{l} \text{morphism} \quad \pi_1(o(s(f))) \\ eq \quad \quad \quad o(s(f)) \\ \quad \quad \quad o(t(f)) \\ m(left(f)) \\ m(right(f)) \end{array}$$

We now have a structure that we can manipulate⁸.

7.4 Graph operations

Before listing the operations that are available, we must describe the proof state. There is obviously the Coq proof state, that can progress, but there is also the graph. The initial state of the graph is inferred from the Coq state (see [section 11.1](#)); however, it can be modified by operations. Thus, the operations can modify the graph, the Coq proof state, or both.

In addition to this information, there is some state that is not proof-relevant; that is, it is not necessary to replay the proof, but it is important nonetheless for the user and some operations. The main example is the layout of the graph (see [section 11.3](#)).

◇ Insertion

The most basic operation involves the insertion of nodes, edges, or faces. The user must provide a Coq proof term that is parsed by

6. The previous property implies the choice of taking the source/target of the left or right path is inconsequential.

7. It can live in either `SProp`, `Prop` or `Type`, it does not matter because we will only build them from other equalities.

8. The more categorically minded reader would have noticed we don't need something as strong as an equality type, but only a 2-category structure, but we decided to keep it as is for simplicity's sake. The implementation would actually support any $(2, 1)$ -category structure, that is, a 2-category structure in which the morphisms in the second level are isomorphisms.

Coq and used to annotate the new graph object. When inserting, the plugin tries to be clever, and when adding an edge, it tries to identify if its source and target in the existing graph to add the edge between them. This operation only modifies the graph and not the Coq proof state.

◇ **Merging**

Two nodes, edges or faces can be merged. When doing so, the graph is modified, but it also unifies the terms annotated on the objects, which may progress the Coq proof by instantiating existential variables. To preserve the graph structure, when merging edges/faces, their source/targets/sides will also be merged.

◇ **Solving**

If a face is annotated by an existential variable, one can attempt to refine it using the other faces in the graph. In paper proofs, this is often the step *left as an exercise to the reader*. To avoid manually progressing trivial things, we implemented a small domain-specific solver. This only instantiates existential variables, and does not modify the graph.

◇ **Decomposition**

Decomposition is a way to split a face into smaller faces, and is the main way to progress the proof interactively. See the next section for details on how this works.

◇ **Normalisation**

If an edge is annotated by a morphism, one may want to normalize the annotated morphism modulo associativity and unitality, and cut the edge into multiple edges along the compositions. This operation does exactly this.

◇ **Lemmas**

The user can apply lemmas that *are about categories*. See the [chapter 8](#) for more details regarding this mechanism.

◇ **Terminating**

The plugin can be terminated at any point during the interaction. The graph is then forgotten, but the progress made on the Coq proof state is preserved. This enables going back and forth between the graphical interface for the proof steps that benefit the most from it, and the Ltac language for the other steps.

These operations are the ones currently implemented, and we believe they are sufficient for most transformations that the user may want to perform on the graph and proof state. However, some transformations would be more natural as primitive operations, and we plan to add new ones based on feedback from users.

7.5 Decomposition

Decomposition is a method to reduce the proof obligation on a face to proof obligations on smaller faces. An example of such a decomposition is shown in figure 7.2. More precisely, a decomposition is a sequence of *steps*, each adding a new proof obligation.

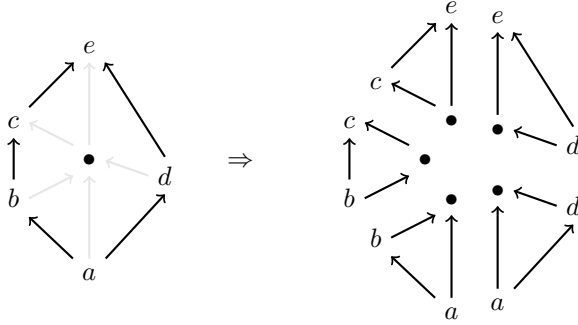


Figure 7.2 An example of a decomposition

Definition 9. A step is a quadruplet $(pre, post, l, r)$ of paths such that⁹ :

- ◇ $s(l) = s(r) = t(pre)$
- ◇ $t(t) = t(r) = s(post)$

We then say that the source of the step is the source of the pre path and its target is the target of the post path. We call the left side of the step the concatenation of the paths pre, l and post and the right side the concatenation pre, r, and post.

A valid decomposition for a face is then a sequence of steps such that :

- ◇ the right side of each step is the same path as the left side of the next step
- ◇ the left side of the first step is the same as the left side of the face
- ◇ the right side of the last step is the same as the right side of the face

You can see a sequence of steps corresponding to the decomposition in figure 7.2 in figure 7.3.

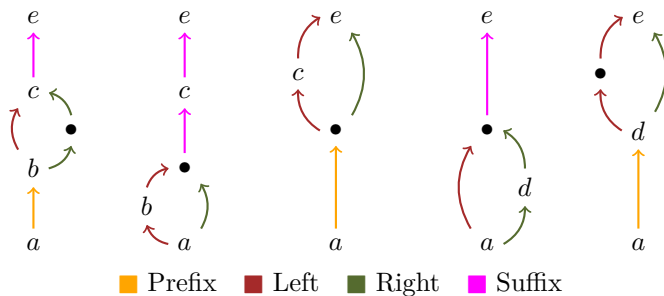
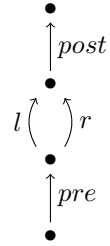


Figure 7.3 A valid sequence of decomposition steps

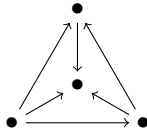
Decomposing the face then means creating a proof obligation $l = r$ for each step, then using the property that $l = r \implies post \circ l \circ pre = post \circ r \circ pre$ to create such an equality from the proof obligation,

9. These conditions means that a step has the following shape :



10. We mentioned in [section 7.4](#) that the layout is not *proof-relevant*, but it is used here for the proof. This is not a contradiction because it is used to *create* the proof, but the proof itself only stores the decomposition as a sequence of steps, which is independent of the layout, and thus can be replayed even if the layout changes.

11. For example, in the following graph, the equality of the outer paths cannot be deduced from the equality of the sides of all the inner faces :



[9](#) A diagram editor in your browser

concatenate all of them, and finally refine the proof obligation of the face with this created equality.

Planar decomposition Manually creating a decomposition for a face is a great deal of work, as there is a lot of information that needs to be entered. However, this information can often be inferred from the graph layout. Indeed, when trying to prove that two paths are equal in a diagram, it is common practice to build up the region delimited by the two paths with new morphisms, and then show that the parallel paths delimitating the innermost regions are equal, which is sufficient to show that the two outside paths are equal. The decomposition in [figure 7.2](#) is one such example. This workflow is implemented in a procedure computing this decomposition¹⁰.

Care must be taken because not all planar decompositions lead to a valid sequence of steps¹¹. The algorithm used is detailed in [section 11.4](#). While we created the algorithm ourselves, the idea of progressing the proof by using planar decomposition is inspired by Ambroise Lafont's work⁹.

8 Lemmas

8.1 Language

The previously described operations on the state graph allow most manipulations that depends only on the categorical structure. However, most proofs are not in abstract categories, but in either specific categories, or categories with more structure.

The additional hypothesis on the graph could be used by going back and forth between Coq and the plugin, performing graph manipulation steps in the plugin, and then applying lemmas from Coq. This is not satisfactory for many reasons. First, despite our efforts, going back and forth still requires a context switch from the user, which interrupts their flow. Second, lemmas that *talk about categories* are more naturally thought of in terms of graphs.

Indeed, many properties are defined as shapes of diagrams commuting, or additional constructions that can be made on a diagram. The main examples are *universal properties*, which are more often drawn than written¹.

The first step is to identify a subset of the Coq lemmas that *talk about categories*. Then, in [section 8.2](#), we describe how to create a graph from them, and in [section 8.3](#), we describe how these graphs can be used to simulate lemma application.

Because the Coq logic is very powerful, we first restrict ourselves to *first order* formulas. More precisely, we consider lemmas which types is a sequence of universal and existential quantifications over types that are *relevant for categories*, and which conclusion is also *relevant for categories*.

Definition 1. A type is said *relevant for categories* if it is the type of categories, a category, an object, a morphism, a functor or an equality between morphisms².

8.2 Associated graph

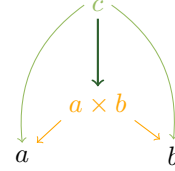
Once we have a lemma that talks about categories, we want to construct a graph associated with it. Here, by *graph*, we refer to the same structure as described in [section 7.1](#), that is, an *annotated 2-quiver*.

The procedure is the same as that for extracting the graph from the proof state, which is described in [section 11.1](#), but with a few differences. The first difference is that, in a quantified formula, the objects under quantifiers depend on the objects above. Therefore, the first step is to bring all the objects in the same scope. To do so, we must eliminate the quantifiers.

For universal quantifiers, this is achieved by creating an *existential variable* of the quantified type, and applying the lemma to this variable, substituting all references to the quantified variable by references to the existential variable³.

For existential quantifiers, we use the fact that Coq is constructive framework in which an existential quantifier is a dependent pair, so we can substitute references to the quantified variables by the first projection of the lemmas⁴.

1. For example, the property of the product indicates that the orange objects exists for all black objects, such that for every light green objects the dark green morphism exists and make the diagram commute :



2. The lack of natural transformation is not due to anything fundamental, it simply has not been implemented yet

3. For example, a lemma $\text{lem} : \forall (x : A), B$ is replaced by $\text{lem } ?x : B[x/?x]$.

4. For example, a lemma $\text{lem} : \exists (x : A), B$ is replaced by $\pi_2 \text{ lem} : B[x/\pi_1 \text{ lem}]$.

By recursively applying these two steps, we obtain a set of terms within the same scope. We can now apply the procedure described in [section 7.1](#) to this set to create a graph associated with the lemma. An example of the application of this procedure is presented in [figure 8.1](#).

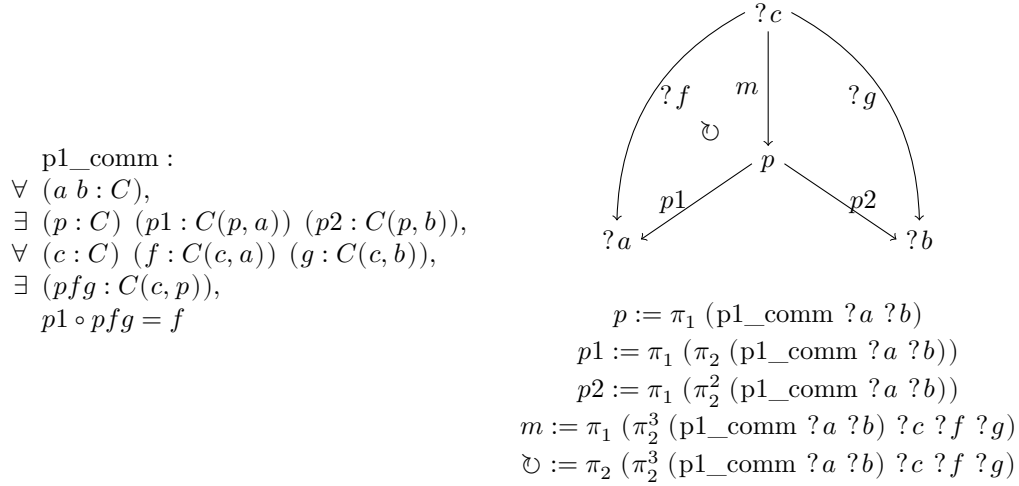


Figure 8.1 Extraction of the graph of the property of the first projection

In contrast to the graphs obtained from the proof state, which has at most one existential from its conclusion, the graphs extracted from the lemmas are filled with existentials, and most of the terms that are not existentials have some in them. Therefore, we call the graph associated with a lemma a *pattern*.

8.3 Application

The ability to represent lemmas as graphs is interesting in and of itself as a visualization aid. However, the notion of applying the lemma is no longer obvious. We found a way to apply lemmas that use only their graph representation, which we believe to be intuitive. We discuss the expressive power of such an operation in more detail in [section 8.4](#).

The idea is to construct a partial matching between the graph of the lemma and the graph of the state. Once this matching is built, the *pushout* of the matching is computed⁵ and the result is the new proof state.

Definition 2. A prematch $\mathcal{M}$ between two 2-quivers $\mathcal{K}$ and $\mathcal{Q}$ is a triplet $\mathcal{M}_N \subseteq N_{\mathcal{K}} \times N_{\mathcal{Q}}$, $\mathcal{M}_E \subseteq E_{\mathcal{K}} \times E_{\mathcal{Q}}$, and $\mathcal{M}_F \subseteq F_{\mathcal{K}} \times F_{\mathcal{Q}}$ such that $\forall (f, f') \in \mathcal{M}_F, |left(f)| = |left(f')| \wedge |right(f)| = |right(f')|$.

This notion of matching is insufficient, because it ignores the quiver structure. Indeed, we also need compatibility conditions stating that if two edges are matched, their source and targets are matched, and similar conditions for faces.

5. Talking about *pushout* implies that we have a span in a category. We will not make it formal here, but we could define a category of annotated 2-quivers such that the operation we describe is exactly the pushout in this category.

Definition 3. Given a prematch $\mathcal{M}$ between $\mathcal{K}$ and $\mathcal{Q}$, we can define $\mathcal{M}_P \subseteq P_{\mathcal{K}} \times P_{\mathcal{Q}}$ the extension of the matching on paths of the same length which component morphisms are either equal or matched by $\mathcal{M}$.

Definition 4. A match between two 2-quivers $\mathcal{K}$ and $\mathcal{Q}$ is a prematch $\mathcal{M}$ such that :

$$\begin{aligned} \diamond \quad & \forall (e, e') \in \mathcal{M}_E, (s(e), s(e')) \in \mathcal{M}_N \wedge (t(e), t(e')) \in \mathcal{M}_N \\ \diamond \quad & \forall (f, f') \in \mathcal{M}_F \wedge \begin{cases} (\text{left}(f), \text{left}(f')) \in \mathcal{M}_P \\ (\text{right}(f), \text{right}(f')) \in \mathcal{M}_P \end{cases} \end{aligned}$$

Such a match now respects the quiver structure, but says nothing about the annotations. To ensure that the entire structure of the annotated 2-quivers is respected, we need additional data and properties for the match.

Definition 5. An annotated match between two annotated 2-quivers $\mathcal{K}$ and $\mathcal{Q}$ is a match $\mathcal{M}$ between them along with a Coq substitution⁷ σ such that :

$$\begin{aligned} \diamond \quad & \forall (n, n') \in \mathcal{M}_N, \sigma(o(n)) = \sigma(o(n')) \\ \diamond \quad & \forall (e, e') \in \mathcal{M}_E, \sigma(m(e)) = \sigma(m(e')) \\ \diamond \quad & \forall (f, f') \in \mathcal{M}_F, \sigma(\phi(f)) = \sigma(\phi(f')) \end{aligned}$$

Before applying a lemma, the user first needs to provide an annotated match between the graph of the lemma and the goal. Of course, the user is not asked for the substitution or to create a complete match. Instead, the user is prompted for a prematch, which can be constructed interactively. The plugin then saturates this prematch into a match. Finally, it reuses the unification features of Coq to compute the most general unifier for all pairs of matched objects. If this step fails, the user is notified and the operation is aborted.

Once such an annotated match is given for a lemma, computing the application is straightforward and involves three steps :

1. Replace the annotations in both graphs with the same annotations substituted by σ .
2. Take the union of the resulting two annotated 2-quivers.
3. Quotient the nodes, edges, and faces of this union by the relations given by $\mathcal{M}_N$, $\mathcal{M}_E$ and $\mathcal{M}_F$ respectively. The definition of a match guarantees that the result is indeed a 2-quiver, and the conditions on the substitution imply that the annotation is compatible with this quotient; therefore, we obtain a valid annotated 2-quiver.

In summary, to apply a lemma which graph we have, the plugin first prompts the user for a prematch (see figure 8.3), then completes it and computes the substitution (figure 8.4), and finally computes the result (figure 8.2).

8.4 Proof flow

Looking again at the previous example from a higher level, we had a goal we wanted to prove and used the lemma to refine it and replace the proof obligation with the lemma's hypothesis. Thus, this is a case of *backward reasoning*, which is what someone would do in Coq using the tactic `apply`.

6. These conditions also imply that $s(f) = s(f')$ and that $t(f) = t(f')$.

7. The notion of substitution we care about is not exactly that of a substitution we can find in the Coq source code, because we want to substitute existential variables, not bound variables, so it would correspond to an `evvar_map`.

8. These three conditions mean that whenever two object are matched together, their annotations must be unified by the substitution σ .

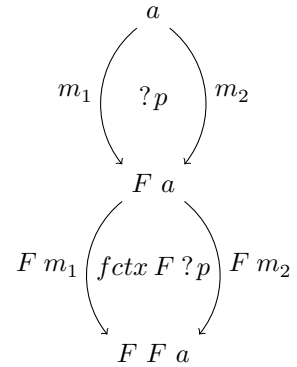


Figure 8.2 Result of the application

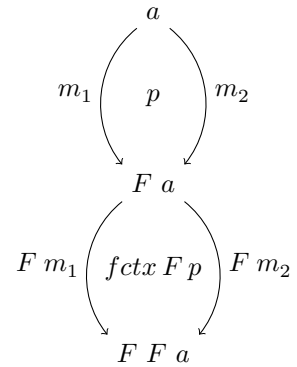


Figure 8.5 Forward application result.

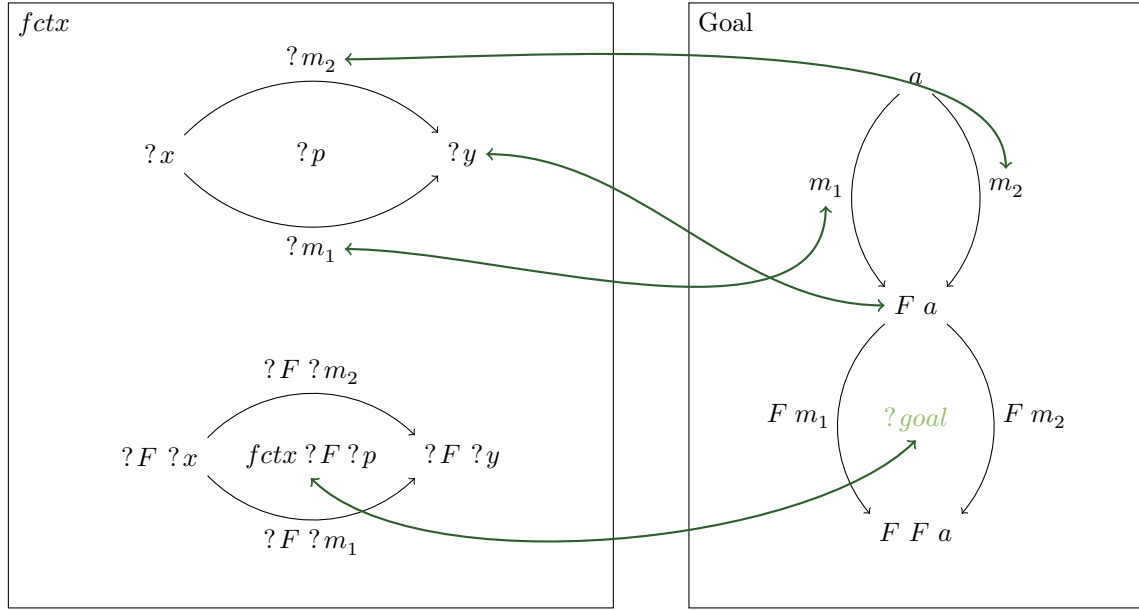


Figure 8.3 A prematch given by the user

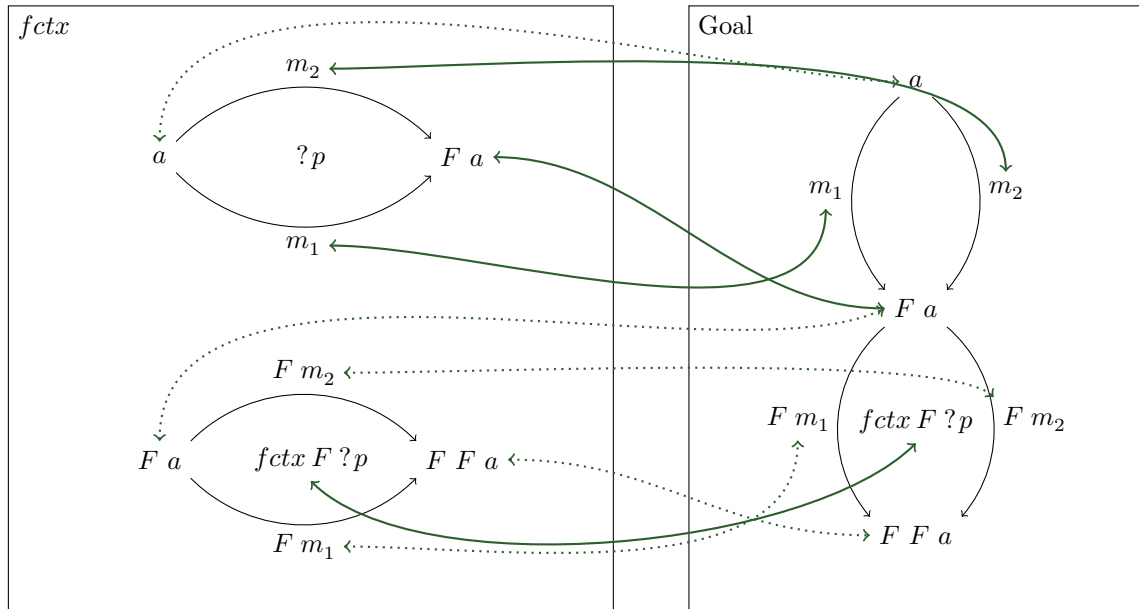


Figure 8.4 The completed match with the substitution applied

However, if instead in the state graph there was an equality p between m_1 and m_2 , and nothing between $F m_1$ and $F m_2$, the exact same mechanism would allow matching p with the hypothesis of the lemma, and computing the pushout would fill the face between $F m_1$ and $F m_2$

(the result of this application is shown in [figure 8.5](#)). Again, looking from a higher level, we apply a lemma to a hypothesis, which is a case of *forward reasoning*.

We could also imagine situations where either both the hypothesis and the goal are present in the state or neither. The application would still work in the same way, and would result respectively in an **exact** application or a cut.

We are satisfied with this approach to lemma application, since it allows one to perform either forward or backward reasoning in a completely uniform way. Furthermore, we believe the method is intuitive.

However, it has one major limitation : it cannot apply a lemma under a term. Indeed, if we have an edge annotated with f ($m_2 \circ m_1$), we cannot apply a lemma under f . This is, in general, a bigger limitation with our approach, the fact that it cannot deal with diagrams under terms. To circumvent this, one would need in Coq to do a cut to prove that $m_2 \circ m_1 = \dots$, and then use the plugin to prove that cut, so it can still be used, but it is a bit fastidious.

8.5 Generalisations

Our implementation is slightly more powerful than what we described. In this section, we describe exactly how much and plans to improve it further.

◇ Reversible equality

The first generalization is linked to the fact that our definition of annotated 2-quiver does not exploit the reversibility of equality. Indeed, the faces in our definition are oriented, and this is necessary because equalities in Coq are oriented. However, from a semantic point of view, the orientation of the equality is irrelevant, and it can be used in any direction⁹. As such, partial matchings ([definition 4](#)) should be able to match faces where the left side of the first one is the same as the second side of the second one, and vice versa. Of course, when doing so, one must take care of inverting one of the equalities before matching. Thanks to the way equalities are actually implemented, this is not a problem (see [section 11.6](#)) for more details).

◇ Parameterized lemmas

This generalization comes from relaxing the restriction imposed on the quantified types when we define a lemma that talks about categories in [section 8.1](#). Indeed, with the current definition, the following object is not considered relevant for categories : $\mathbb{N} \rightarrow \text{object } C$ where C is a category in Coq. Such objects are common in some domains of category theory, especially in *homological algebra*. What prevents such a lemma¹⁰ from being used is that $\mathbb{N}$ is not relevant for categories ([definition 1](#)).

However, we could still construct a graph from such a lemma by eliminating quantifiers as described in [section 8.2](#), and then ignoring Coq objects that are not relevant for categories when building the graph. These objects, while not directly present in the graph, would still be present as existential variables in the terms in

9. One more categorical way to formulate that is to say that instead of having diagrams in a weak 2-category, we have diagrams in a weak $(2, 1)$ -category, where the second level lives in groupoids

10. We call lemma any Coq type we are interested in, which is not simply a category, an object, a morphism, or an equality between morphisms. Such a *parameterized object* as $\mathbb{N} \rightarrow C$ is treated as a lemma which graph is a single node

11. For example, an edge annotated by $m : x (1 + ?n) \rightarrow x ?n$ could have his source matched with $x 5$, resulting in the substitution $?n \mapsto 4$, without the plugin needing any knowledge about naturals. Of course, it still suffers from the usual drawbacks of Coq, where $1 + ?n$ could be unified with 5 but not $?n + 1$.

12. In fact, with the previous generalization, it would still be supported, but the objects in the product would not be added to the graph, since the product would be viewed as a single opaque object.

the graph. Thus, while the plugin cannot work directly with them, they can still be refined through unification with other terms¹¹.

Another advantage of this approach is that, since all remaining existential variables are added as Coq goals when terminating the plugin, it would support a lemma that, for example, expects an argument of the form `IsIsomorphism m`, allowing us to use this property and then have to prove that whatever m is unified with is an isomorphism when returning to Coq, without having the plugin know anything about isomorphisms.

A lemma that talks about categories is then simply a sequence of universal or existential quantifiers, where the conclusion is a type relevant for categories. The main drawback of this approach is that applying the lemma may create many proof obligations that are not shown in the interface until the user returns to Coq. This could be mitigated by displaying a list of these obligations in the interface, even if they cannot be interacted with.

◇ More expressive language

While the language for lemmas described in [section 8.1](#) is quite general, especially with the generalization of the previous paragraph, it still suffers from a pretty obvious drawback. Indeed, due to the details of the definition, it would not support a curried version of a lemma, where instead of a sequence of quantification, there is a single quantification with a product of hypotheses¹². Furthermore, if the conclusion is a product, then the lemma would be ignored by the plugin. This is also complicated if the relevant information is hidden in a record.

Adding support for products and records would not change anything from a theoretical point of view, the procedure in [section 8.2](#) would be exactly the same with the addition of splitting quantifiers of products or records into quantifiers of independent objects, and then replacing the usage of the variable by the constructor applied to the independent existential variables. However, this has not yet been implemented, in part because we wanted to keep the Coq term parser simple until it could be replaced by a more powerful alternative (see [section 12.4](#)).

Sums and simple inductive cases can also be supported by generating multiple graphs for a lemma, one for each inductive case. However, this may lead to an explosion in the number of graphs; therefore, some compromise or heuristic may be needed if this is to be implemented.

9 The solver

9.1 The problem

When performing diagrammatic proofs, it is common to build diagrams, write arguments to show that some parallel paths are equal, and then directly conclude without detailing how those equalities give rise to the equality wanted. This last step is considered trivial and purely technical; therefore, it is often *left as an exercise to the reader*.

We would like for trivial steps in paper proofs to be trivial in proof assistants. Therefore, we sought an algorithm that can automatically deduce the equality from the equalities present in the graph. This work was actually done before we started developing the interface for diagrammatic reasoning, and is mostly independent from it, even if it is now available as a specific operation (see [section 7.4](#)) in it.

We need to precisely define the problem that we are attempting to solve. Intuitively, the input should be a diagram with a set of parallel paths assumed to be equal. Implemented as a Coq tactic, the structure is thus exactly the one we have described in [section 7.1](#), the annotated 2-quiver.

We also need a pair of parallel paths in this graph, and we want to try and prove that they are equal from the other equalities and the structural laws of categories. However, this formulation is still imprecise. First, do we want to try and prove that the two paths are equal, and maybe fail, or do we want to succeed whenever the equality can be deduced, and provide a proof of inequality if we fail. Second, we mentioned the *structural law of categories*, but we must specify exactly the one we are talking about.

To prove things, we need to define things more formally. We reuse the [definition 1](#) of quivers in [section 7.1](#). We can then define the *free category* on this quiver.

Definition 1. *The free category over a quiver $\mathcal{K}$, denoted by $F_{\mathcal{K}}$, is the category whose objects are the nodes of the quivers, and arrows are the paths in the quiver. The identities are then the empty paths and the composition the concatenation of paths¹.*

Given a quiver $\mathcal{K}$ and a set of parallel morphisms $\mathcal{S}$ in the free category over $\mathcal{K}$, we can construct a relation $R_{\mathcal{S}}$ on morphisms in the free category.

Definition 2. *$R_{\mathcal{S}}$ is the set of pairs of parallel morphisms containing $\mathcal{S}$ and closed by precomposition and postcomposition in $F_{\mathcal{K}}$ ².*

Definition 3. *We define $\tilde{R}_{\mathcal{S}}$ the reflexive, symmetric, transitive closure of $R_{\mathcal{S}}$.*

Proposition 1. *$\tilde{R}_{\mathcal{S}}$ is compatible with the concatenation of paths :*

$$\begin{aligned} \forall(p_1, p'_1) \in \tilde{R}_{\mathcal{S}}, \forall(p_2, p'_2) \in \tilde{R}_{\mathcal{S}}, \\ s(p_1) = t(p_2) \implies (p_1 \cdot p_2, p'_1 \cdot p'_2) \in \tilde{R}_{\mathcal{S}} \end{aligned}$$

Proof. Immediate by construction of $R_{\mathcal{S}}$ and transitivity of $\tilde{R}_{\mathcal{S}}$. □

1. It would also be possible to define it as the adjoint functor to the forgetful functor from the category of categories to the category of quivers; however, for our purposes, this definition is easier to work with.

2. That means that if we have $(m_1, m_2) \in R_{\mathcal{S}}$ two parallel path and m a morphism in $F_{\mathcal{K}}$ that is respectively precomposable or postcomposable with m_1 and m_2 then we respectively have $(m \circ m_1, m \circ m_2) \in R_{\mathcal{S}}$ or $(m_1 \circ m, m_2 \circ m) \in R_{\mathcal{S}}$.

We finally have all the tools needed to define the quotient of the free category.

Definition 4. We write $F_{\mathcal{K}}/S$ the free category generated by $\mathcal{K}$ quotiented by S . It is the category whose objects are the nodes of $\mathcal{K}$ and whose morphisms are the morphisms of $F_{\mathcal{K}}$ quotiented by $\bar{R}_S$. *Proposition 1* show that the concatenation of paths gives a well defined operation on the equivalence classes, and the identity are the equivalence classes of the identities. We write $q_S : Mph(F_{\mathcal{K}}) \rightarrow Mph(F_{\mathcal{K}}/S)$ the quotient map.

Proof. Since we have more equalities than in $F_{\mathcal{K}}$, the associativity and unitality axiom are still valid. $\square$

Thus, the problem we want to solve is as follows : given a quiver $\mathcal{K}$, a set of parallel morphisms S in $F_{\mathcal{K}}$ and a pair of parallel paths p_1, p_2 in $F_{\mathcal{K}}$, do we have $q(p_1) = q(p_2)$ in $F_{\mathcal{K}}/S$.

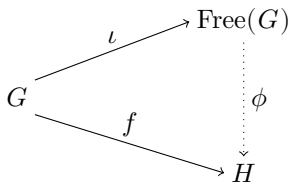
[30] Evans, *Word problems*, 1978

[31] Boone, *THE WORD PROBLEM*, 1958

3. William Boone proves that the word problem for groups is undecidable by encoding *Thue systems* in group representations. We could cut the middle man and directly encode those systems in diagrams, which would be even easier than for groups. However, we could not find the 1947 paper by Emily Post showing the undecidability of such problems by encoding the halting problem in a Thue system. The proof is simple enough, but we think it is better to rely on a paper that is easier to find.

4. Here, we do not have an inverse operation, and $\cdot^{-1}$ should be understood as the creation of a formal inverse.

5. The universal property of the free group over G tells us that for every function f from G to a group H , there exists a single group morphism ϕ that makes the following diagram commute :



9.2 Decidability

This is a type of *word problem*, that are usually undecidable [30]. Therefore, we expect this to be undecidable. Indeed, we can encode the word problem for groups, that is undecidable [31]³.

The word problem for groups is as follows : given a set of generators G , a set of pairs P between words in the underlying monoid of the free group generated by those generators, and a final pair of words, decide whether these two words are equal in the free group quotiented by the set of pairs.

Given this problem, we can create a quiver $\mathcal{K}$ with $N_{\mathcal{K}} := \{\bullet\}$ and $E_{\mathcal{K}} := \{g \mid g \in G\} \cup \{g^{-1} \mid g \in G\}$ ⁴. Because $N_{\mathcal{K}}$ is a singleton, we do not need to specify the source and targets functions.

The set P can also be considered as a set of paths in $\mathcal{K}$. We cannot take it as is because we also need to add equalities to enforce the composition of inverses to be equal.

Definition 5. We define the set S to be :

$$S := \bigcup \begin{array}{l} P \\ \{([g] \circ [g^{-1}], 1_{\bullet}) \mid g \in G\} \\ \cup \{([g^{-1}] \circ [g], 1_{\bullet}) \mid g \in G\} \end{array}$$

Proposition 2. $Mph(F_{\mathcal{K}}/S)$ is a group.

Proof. As the morphism set of a single object category, it is immediately a monoid. Now we need to show that every object has an inverse. We can do that by induction on the length of the paths, using the fact that $1_{\bullet}$ is its own inverse and *definition 5* along with *proposition 1* for the recursive case. $\square$

Proposition 3. $Mph(F_{\mathcal{K}}/S)$ is exactly the free group over G quotiented by P .

Proof. Using the universal property of the free group⁵, one can create a map ϕ from the free group over G to $Mph(F_{\mathcal{K}}/S)$, then show that it

is compatible with P to create a map from the quotiented free group to $\text{Mph}(F_{\mathcal{K}}/\mathcal{S})$.

Expanding the definition, for a word w in the underlying monoid of the free group over G , which is exactly the set of paths in $\mathcal{K}$, we have that the image by ϕ of the class of w is the class of w in $\text{Mph}(F_{\mathcal{K}}/\mathcal{S})$. So for ϕ to be the identity we only need it to be injective.

It is actually easier to directly show that ϕ is bijective by constructing an inverse. To do so, we consider the free group over G as a single object category, and use the universal property of the free category $F_{\mathcal{K}}$ to construct a functor from $F_{\mathcal{K}}$ to the free group. The action on morphism of this functors gives us a group morphism, and we can apply once again the universal property of the free group over G to show that the composition is the identity. $\square$

Proposition 3 immediately gives us the correction of the encoding of the word problem in group in our problem, which is thus undecidable.

9.3 Topological approach

This section describes an attempt to create an algorithm based on topological intuition, using tools from computational algebraic topology. This approach did not succeed, and this section assumes some familiarity with algebraic topology, so it can be skipped.

Intuitively, when trying to show that a diagram is commutative, it is common to draw it in a specific way, then try to fill the innermost faces of the graph. The proof is then completed when no hole remains. Therefore, another way to see it is that the proof is complete when the topological realization of the graph, with equalities realized as surfaces, is simply connected.

We can weaken this objective. Since we are not trying to show that the entire graph is commutative but that two parallel paths are equal, what we need is for these paths to be equal when projected into the fundamental homotopy group of the space.

This space has a structure that makes us believe that it can be used for computation. Indeed, it is the realization of a 2-globule; therefore, it is a 2-globular space. This means that it is in particular a CW-complex, for which we can compute the *cellular homology* [32]. Although this homology can be automatically computed, we cannot deduce the fundamental homotopy group from it. At most, this can yield a negative result⁶.

Furthermore, the information regarding which faces are filled is actually too weak. Indeed, the side of the faces is oriented and just having the two paths be the same in the fundamental homotopy group is not enough to conclude that they are equal⁷. We must have some direction to our paths and faces. Thankfully, a theory of *directed algebraic topology* [33] has been developed.

Our idea at that point is the following :

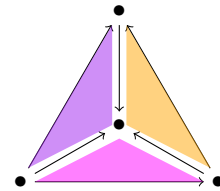
1. Transform our graph into a directed cubical complex.
2. Compute the directed homotopy of this complex.
3. Check if the two paths we want to make equal are the same in the homotopy.

However, we could not progress any further at this step for one reason, which is that in directed algebraic topology, we do not have fundamental

[32] May, *A concise course in algebraic topology*, 1999

6. If two paths are different in the homology group, then they are different in the homotopy grouper. The reverse is not true in general.

7. For example, the following graph is simply connected if seen as a topological space. However, we cannot conclude the equality of the outermost paths from the equality of the innermost paths :



[33] Fajstrup, Goubault, Haucourt et al., *Directed Algebraic Topology and Concurrency*, 2016

8. This limitation is the only approximation step that renders the algorithm incomplete. While such an approximation is necessary because the problem is undecidable, we think that this heuristic is actually quite intuitive, so that the user can guess the correct value to set.

9. This structure is actually slightly different from the standard one because we need to be able to extract proofs from it to feed them back to Coq. See [section 11.2](#) for the details of this structure.

10. This approach is similar to that of the congruence closure algorithm. However, the fact that composition is an open term prevents us from reusing this algorithm. See [section 9.6](#) for a detailed comparison.

11. This loss of features was due to a complete reimplementing of the solver, owing to a change in the architecture of the plugin needed by changes in the goals of the plugin. More details are provided in [chapter 10](#).

12. When doing this, care must be taken that the equality produced is still between morphisms parenthesized in the right way. Depending on whether we parenthesize to the right or to the left, one of these hooks is trivial while the other needs to reparenthesize the two sides of the equality after the composition.

groups but fundamental categories. Looking at the presentation of the category we obtain after all these steps, it turns out to determine if the two paths are the same, we need to solve the exact problem we were trying to solve, and actually we would probably obtain the same problem after all these transformations, in another equivalent presentation.

We did not further explore these ideas. It might be possible to obtain a procedure to simplify the problem before executing the real solver, but we believe it unlikely that the simplification would be worth the complexity of this procedure, both in runtime and implementation effort.

9.4 Algorithm

The algorithm we designed is conceptually simple. First, we enumerate all the paths up to a certain length⁸. We then create a disjoint-set structure⁹ on the paths. We connect the paths for which we have equality, along with a saturation procedure, to handle the compatibility of equality with the composition¹⁰. Finally, we query the structure for an eventual proof of equality for the two parallel paths of interest.

The saturation procedure is tied to the addition of the paths. It works by maintaining a queue of pairs of parallel paths along with an equality proof between them. This set is initialized with the hypotheses equalities. Then, while the queue is not empty, one such pair is extracted from it. If one of the paths is not present in the disjoint set, or if they are already connected, nothing is done. Otherwise, the Coq equality is used to connect them, and the triplet is given to a set of hooks that may add new triplets to the queue.

This procedure terminates because either the number of pairs of paths not connected in the disjoint set decreases (in which case the size of the queue may increase), or it remains the same and the queue is reduced, so each step decreases a value in $\mathbb{N}^2$ with the lexicographic order, which is well founded.

The advantage of this approach is that the hook system is powerful, and allows the encoding of many mechanisms. In the current implementation, only the pre and post composition hooks are implemented, but in the first implementation, all those described afterwards were present¹¹.

◇ Composition hooks

For each edge in the graph, we add two hooks. This first hook checks if the target of the paths added is the same as the source of the edge, and creates a new equality between the two paths extended by the edge at the end. The other hook does the same, but at the beginning of the path¹². These two hooks are responsible for the congruence of the equality relative to the composition.

◇ Isomorphisms

For each edge annotated with a morphism that is an isomorphism, we add four hooks that perform the post and pre composition with the isomorphism and its inverse, which also simplifies when the last or first morphism of one of the paths is the inverse of the

morphism¹³. As we will see in [section 9.5](#), there is a better way to deal with isomorphisms than hooks, but this illustrates the power of hooks.

◇ Epi/mono-morphisms

For each edge that is annotated by a monomorphism, we add a hook that checks whether the two paths added end by the monomorphism, and if it is the case that uses the left cancellative property of the monomorphism to add an equality between the paths without the monomorphism. The same is also done for epimorphism on the other side. While isomorphisms are both epimorphisms and monomorphisms, the handling of isomorphisms means that these hooks are not required.

◇ Functors

For each functor in the context, we add a hook that checks whether the equality added is in the source category of the functor. If this is the case, it adds the image equality by the functor¹⁴.

9.5 Optimisation though rewriting

The disjoint-set structure is really efficient¹⁵; therefore, the execution time is dominated by the application of the hooks, enumeration of paths, and initialization of the disjoint set. The application of hooks is linear in the number of hooks and number of equality to add, which scales with the number of paths in the graph.

The number of paths in the graph can grow rapidly, even exponentially in the size of the graph. For example, a graph with a single node and m different morphisms looping on it will have m^n paths of length n . Of course, this is a worst case situation, and the reason it behaves badly is because it has many short loops. There are two very common cases in which short loops are encountered in category theory. The first is of course identities, and the second is isomorphisms, which, with their inverse, create loops of size 2.

The idea to handle this case is to only enumerate paths modulo a rewriting system that eliminates identities and composition of isomorphisms. The modification of the above algorithm is simple. First, when enumerating the paths, we normalize them and deduplicate the result. After running a hook, when new equalities are added, the sides of the equalities are normalized¹⁶.

They are not implemented as a generic rewriting engine [\[34\]](#), but as an ad-hoc algorithm that performs normalization equivalent to the equational theory described in [figure 9.1](#). Replacing this algorithm with a generic rewriting engine would make the solver extensible. Indeed, as long as the rewriting system is confluent and strongly terminating, and that for every rewrite rule added we have a Coq proof of the equality¹⁷ between the left hand side and right hand side, it can be used with the algorithm of the solver without any change¹⁸.

However, the rewriting engine should not be over morphisms but over paths, since the associativity and composition of inverse morphisms create a system that is not confluent¹⁹. The way to do so is to first normalize modulo associativity to create a path and then implement a

13. These mechanisms automatically cause isomorphisms to behave like epimorphisms and monomorphisms. Indeed, if we have an equality between two paths that start with an isomorphism $i \rightarrow p_1$ and $i \rightarrow p_2$, these hooks will precompose by i^{-1} . The simplification then adds an equality between p_1 and p_2 .

14. This means that we also need to normalize the morphisms on the side of the equality modulo the fact that functors preserve composition. We will discuss this in more detail in [section 9.5](#).

15. Its complexity for union and query is $O(\alpha(n))$ where n is the number of elements and α is the inverse Ackerman function, so in practice it can be considered constant time because $\alpha(2^{64}) \leq 5$.

16. Because we simplify the composition of inverses in the normalization, we do not need the isomorphism hooks anymore because the results of the pre and post composition hooks will be normalized.

[\[34\]](#) Hondet and Blanqui, *The New Rewriting Engine of Dedukti*, 2022

17. We need the Coq equality term because we not only need to normalize the term, but also to construct an equality between the source term and its normalization.

18. This is not exactly true because the current implementation also assumes that the normalization process will not increase the length of a path. However, this is to simplify the implementation, and this assumption can be removed if the need arises.

19. For example, we have the following critical pair that cannot be closed :

$$\begin{array}{ccc}
 & (m \circ m^{-1}) \circ m' & \\
 \swarrow & & \searrow \\
 \text{id} \circ m' & & m \circ (m^{-1} \circ m')
 \end{array}$$

rewrite engine of paths. Currently, our ad hoc implementation has the advantage that it does both simultaneously.

$$\begin{aligned}
 (m_1 \circ m_2) \circ m_3 &\rightarrow m_1 \circ (m_2 \circ m_3) \\
 m \circ \text{id} &\rightarrow m \\
 \text{id} \circ m &\rightarrow m \\
 m \circ m^{-1} &\rightarrow \text{id} \\
 m^{-1} \circ m &\rightarrow \text{id} \\
 F (m_1 \circ m_2) &\rightarrow F m_1 \circ F m_2 \\
 F \text{id} &\rightarrow \text{id}
 \end{aligned}$$

Figure 9.1 Equational theory implemented by the solver

9.6 Comparison with other algorithms

◇ Congruence closure

Our algorithm is a way to compute a congruence closure for equality under composition and the application of functors. A famous and complete algorithm allows the computation of this [35]. Furthermore, because of its relevance in many branches of computer science, it has been improved since, with optimisations like *equality graphs* [12], and has been extended so that proofs can be generated by this algorithm [36]. However, it makes the assumption that the equalities we have as hypotheses are all quantifier free. While the hypotheses we use from the Coq context are indeed closed, we also need to reason modulo associativity and unitality. Therefore, this algorithm cannot be used directly. It might be possible to replace the ASTs used by a structure that makes associativity and unitality tautological by construction (e.g. using lists of morphisms); however, adapting the algorithm to this new structure, if possible, is not trivial and would require more work. This is a direction for future work we may be interested in.

◇ Z3

Owing to its support for uninterpreted functions and constants, Z3 [13], an industry grade SMT solver developed by Microsoft, can be used as an implementation of congruence closure that also supports quantifiers. The absence of dependent types means that we need to use an encoding with a single morphism sort and an explicit error morphism that is used as a result when composing non compatible morphisms. The complete formalization used is presented in figure 9.2. However, this formalization is excessive for Z3, which hangs after consuming all the memory of the computer when ran on this input. Therefore, it cannot be used to replace our algorithm. Furthermore, while Z3 can export a proof, it only exports a classical proof and reconstructing a valid Coq proof from it is an entire project in and of itself [37].

[35] Nelson and Oppen, *Fast Decision Procedures Based on Congruence Closure*, 1980

[12] Egg

[36] Nieuwenhuis and Oliveras, *Proof-Producing Congruence Closure*, 2005

[13] Z3

[37] Armand, Faure, Grégoire et al., *A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses*, 2011

◇ Nuprl

Kozen, Kreitz and Richter designed a Gentzen style deduction system for elementary category theory proofs [38][39]. It describes a higher level first order language to talk about category theory without relying on an encoding into a type system or ZFC. They then implemented this language in the logical framework NuPRL[14] and used this implementation to write proof search procedures tailored for category theory. These procedures are a set of heuristics based on own the authors' own experience with category theory. Most of these are guides for proof tree search; however, similar to our work, they also use a rewriting system to solve equalities.

Their work is much more complete and powerful than what we have done. They can automatically prove much more complicated proofs, and the scope of their approach is much broader. However, to achieve this, they created their own logic specialized for category theory, while our work can be plugged into any Coq development that uses category theory. The approaches are sufficiently different that comparing performance does not really make sense.

◇ Computational category theory

A very different approach to automating some proofs in category theory is to use frameworks such as *Catlab.jl*[15] or *CAP*[16], both libraries in Julia[17], which allow computing with finitely presentable categories. Indeed, some theorems in category theory can be formulated as the commutation of diagrams in a specific free category. Using the previously mentioned framework, such diagrams can be computed, and one can check that all the parallel paths in the diagram indeed compute to the same extent. However, this approach has two limitations. The first is that it cannot reason with abstract categories; it needs concrete, finitely presentable categories to compute with. Furthermore, there is no way to extract a formal proof from such a computation. As such, this approach is meant to solve a different problem than what our algorithm is attempting to solve.

[38] Kozen, Kreitz and Richter, *Automating Proofs in Category Theory*, 2006

[39] Kreitz, *Nuprl as Logical Framework for Automating Proofs in Category Theory*, 2012

[14] NuPRL

[15] Catlab.jl

[16] Categories, Algorithms, Programming

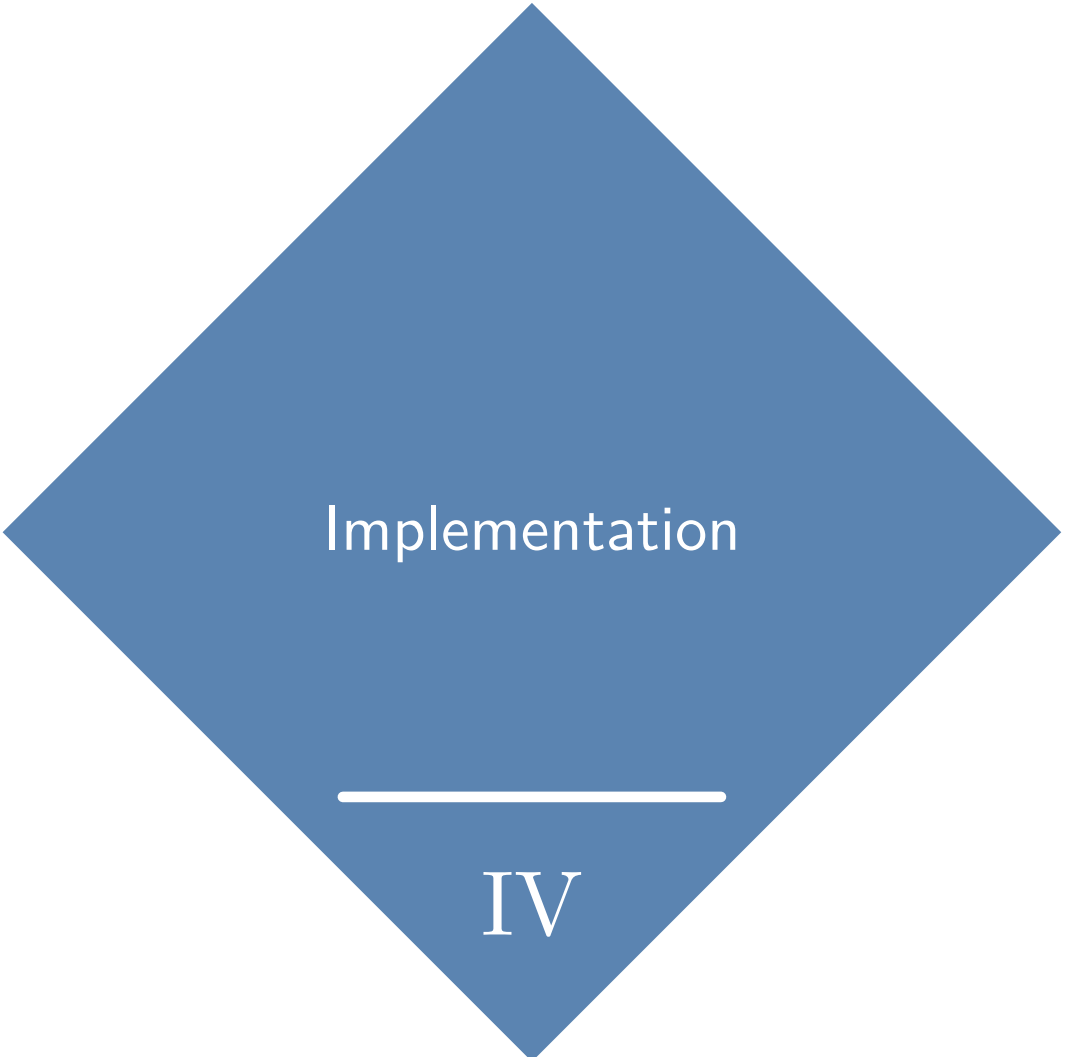
[17] Julia

```

(declare-sort M)
(declare-sort O)
(declare-const e M)
(declare-fun s (M) O)
(declare-fun t (M) O)
(declare-fun i (O) M)
(declare-fun c (M M) M)
(assert (forall ((o O)) (= (s (i o)) o)))
(assert (forall ((o O)) (= (t (i o)) o)))
(assert (forall ((o O)) (not (= (i o) e))))
(assert (forall ((m1 M) (m2 M)) (=> (= (t m2) (s m1)) (= (s (c m1 m2)) (s m2)))))
(assert (forall ((m1 M) (m2 M)) (=> (= (t m2) (s m1)) (= (t (c m1 m2)) (t m1)))))
(assert (forall ((m1 M) (m2 M)) (=> (not (= (t m2) (s m1))) (= (c m1 m2) e))))
(assert (forall ((m M)) (= (c m e) e)))
(assert (forall ((m M)) (= (c e m) e)))
(assert (forall ((m M) (o O)) (=> (= (s m) o) (= (c m (i o)) m))))
(assert (forall ((m M) (o O)) (=> (= (t m) o) (= (c (i o) m) m))))
(assert (forall ((m1 M) (m2 M) (m3 M)) (= (c (c m1 m2) m3) (c m1 (c m2 m3)))))

```

Figure 9.2 Z3 encoding of category theory.



Implementation

IV

10 Architecture

10.1 Split process organisation

This plugin has undergone many steps during its development. Initially, it was only an implementation of an automatic solver for categorical diagrams, as described in [chapter 9](#). Back then, a simple Coq plugin was necessary. A Coq plugin is an OCaml [18](#) development that can declare new tactics¹ implemented as an OCaml function executed inside a specific monad that allows it to inspect and modify the proof states using all the functions in the Coq source code².

This plugin implemented the analysis of the proof context ([section 12.2](#)), the construction of the graph from it ([section 11.1](#)), the union find ([section 11.2](#)), and the algorithm described in [section 9.4](#). It declared a tactic `diagram solver` that would attempt to solve the current goal using the previously mentioned algorithm.

Because this algorithm does not depend on any feature specific to Coq, we attempted to isolate the parts of the code that depended on Coq as much as possible. However, this made the code more complicated, and because other proof assistants have other mechanisms to write plugins, it could not be reused anyway.

When we started thinking of writing an interface to work with categorical diagrams, we did not want to write it in OCaml for two reasons :

- ◇ Due to difficulties with the linking mechanism in Coq, [depending on external libraries from a Coq plugin is complicated](#)
- ◇ This would make all the interface development specific to Coq.

To circumvent this problem, we designed an abstraction over Coq described in [section 10.2](#), designed a protocol described in [section 10.4](#) to interact with this abstraction, rewrote the plugin to implement this protocol and expose it locally, and finally implemented the interface in a separate development in Rust³ that connects to the plugin and only interacts with Coq through this protocol.

The initial abstraction over Coq proved to be too limited for our goals, so we created a new one, more abstract, that allowed greater reuse of the features in Coq, as described in [section 10.3](#)⁴.

The final architecture is illustrated in [figure 10.1](#). A successful execution of the plugin proceeds through the following steps :

- ① The Coq plugin is invoked through the `diagram run` or `diagram edit` tactics.
- ② It looks at the current goal. If it is an equality between morphisms, it constructs the graph from the current proof state, and initializes the model.
- ③ It looks for lemmas, and, whenever possible, extracts the graph for the lemmas⁵.
- ④ It starts the interface as a subprocess, and connects to it through its `stdin` and `stdout`⁶.
- ⑤ The interface does a simplification step on the graph.
- ⑥ The interface attempts to replay the previous proof if there is one. On success it terminates without opening any windows.

[18](#) OCaml

1. A plugin can also declare new top level vernacular commands, but we do not use this in our development.

2. Indeed, there is no identified set of functions that form an API for plugins. Instead, when loaded, the plugins are directly linked against the Coq executable and have access to all the functions in its implementation. While this means that there is no inherent restriction when writing things as a plugin, it also means that every new major Coq version may break the plugins written in OCaml.

3. I am often asked about my choice of Rust for the interface. I needed something reasonably performant because some of the features are quite computationally expensive, but the real reason was that I had been wanting an excuse to learn Rust for a long time.

4. The conception of this new model was quite quick, but converting the codebase to it necessitated almost a complete rewrite, which took over 3 months.

5. This step is responsible for the 3 seconds delay when starting the interface. We would like for this to happen asynchronously with the following steps, but the current implementation does not support such a flow.

6. The current implementation for this step uses the `Unix.open_process_args_full` function, and as such does not work on Windows. However, this could probably be easily fixed by switching to a Windows-specific call, since the rest of the plugin only uses features from the OCaml standard library.

7. The layout step is interactive, so it occurs in real time while the graph is displayed and edited. See [section 11.3](#) for further detail.

- 7. It layouts⁷ and displays the graph. At this step, the user can interact with the interface and progress the proof using the operations described in [section 7.4](#).
- 8. During the interaction, changes have been made to the Coq state. Upon success, these changes are used to refine the proof state, and the plugin returns to Coq.

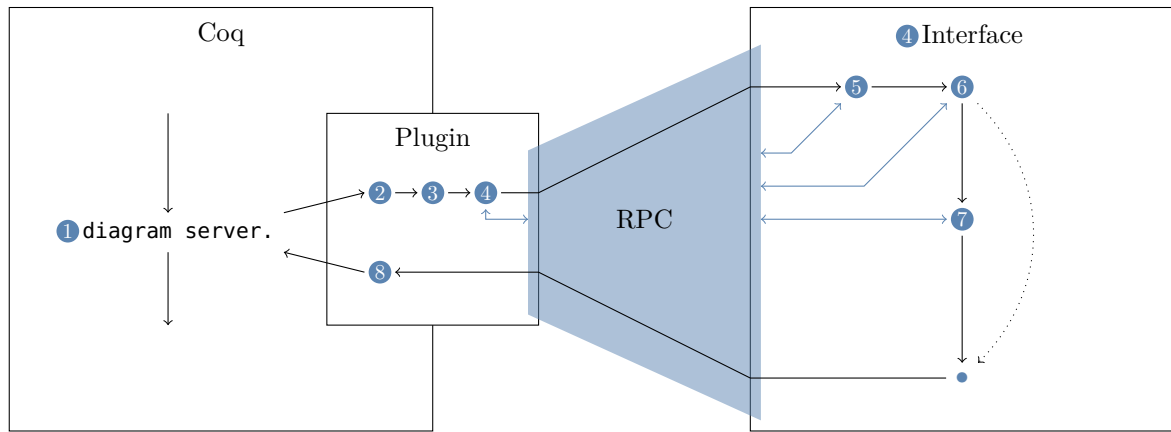


Figure 10.1 The architecture of the plugin with the interface

Another advantage of this infrastructure is that, because the Coq plugin abstracts away the details of Coq, similar plugins implementing the protocol could be developed for other proof assistants, and the interface could be reused as-is directly for those other proof assistants. All the complicated features are implemented in the interface, so it would allow the reuse of a lot of code with a minimal effort⁸.

10.2 The former data model

When designing the abstraction over Coq, we first started with a model based on ASTs, which is described in this section. This approach proved too limited; therefore, we designed another one, more high level, which is described in the next section. What we will describe here is no longer implemented and can be skipped. However, since this approach is the most intuitive, we believe it is important to describe it so that we can understand why it failed.

This approach is based on the creation of domain specific ASTs that the interface and plugin can exchange⁹. The plugin needs to be able to parse a Coq term into such an AST, and realize it back into a Coq term. The AST used is shown in [figure 10.2](#). Any term present on the leaf of an AST is treated as a black box, and corresponds to the *atomic* terms. We need a way to talk about atomics in the interface, where the `EConstr.t`¹⁰ type does not exist. Thus, the plugin maintains an array of such terms, and only sends the interface their index. We also need to represent holes in the AST. To do so, we use existential variables, that need to be present in the AST. To represent existential

8. Despite our efforts to keep it as small as possible, the Coq plugin is still around *2k single line of codes*, so not too large but not so trivial to port. We believe that it should be a few weeks of effort to do something similar for someone used to develop plugins for another proof assistant.

9. We can describe how it used to be serialized and the old protocol, but we will not because it is not really interesting. Describing the current protocol is sufficient to get the idea anyway.

10. The type for Coq terms in the source code.

variables in the AST we use integers; therefore, we use another array of evars¹¹.

One advantage of this approach is that the plugin can perform some deduplicating when adding Coq terms to the atomic array. Each atomic is also stored with its type in a structured manner. It also makes rendering of the atomic text available to the interface. Thus, the information that the interface can have about an atomic is as follows :

- ◊ Its text representation.
- ◊ If it is a category.
- ◊ If it is an object, and the category it belongs to.
- ◊ If it is a morphism, and the category, source and target objects.
- ◊ If it is an equality between morphisms, and the category, source, target, and left and right morphisms.

$a := \text{EConstr } c$	$p_1, p_2 := \text{Atomic } a$
Evar e	Refl m_1
$C := \text{Atomic } a$	Concat $p_1 \ p_2$
$x := \text{Atomic } a$	Inv p_1
FObj $F \ x$	Compose $p_1 \ p_2$
$F := \text{Atomic } a$	Assoc $m_1 \ m_2 \ m_3$
$m_1, m_2, m_3 := \text{Atomic } a$	LeftId m_1
FMph $F \ m_1$	RightId m_1
Comp $m_1 \ m_2$	FId $F \ x$
Id x	FComp $F \ m_1 \ m_2$
	FCtx $F \ p_1$

Figure 10.2 A simplified view of the AST used in the initial model.

Using this information, the interface works without interacting with Coq until it finished. On success, it sends to Coq the ast for an equality term, which is realized as an equality term and refined against the current goal to progress it. Any existential variable still present in the returned term is instantiated as a new goal to be solved.

For the interface to successfully work with these ASTs, it is necessary to reimplement most of the common features one can usually find in a proof assistant, including pretty printing¹², parsing, substitution, and unification. The unification algorithm used is described in [section 11.8](#).

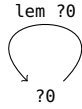
This approach has important limitations. First, we could not reuse the notation system of Coq, neither for parsing nor for pretty-printing. At that point, it is problematic but not a deal breaker because atomics are pretty-printed by Coq, and so are using notations. However, this became problematic when the modifications needed to handle the lemmas were implemented. Second, because we used our unification, it means it does not know anything about $\beta\delta\iota$ reduction, making it even more brittle than the unification of Coq.

The addition of support for lemmas complexified the situation beyond what was acceptable, leading to the new data model and the corresponding refactor. The main problem was that the introduction of

¹¹. In Coq, an Evar is already a wrapper over an integer. We did not reuse this integer because we needed a method to distinguish evars already present in Coq from those created by the interface, and there would have been a risk of collision if they lived in the same namespace.

¹². All this was scrapped before we looked into the details of pretty printing, so calling it *pretty* printing is a bit of a stretch, it was more accurately described as debug printing.

13. The graph :



lemmas meant that we could have existential variables that were arbitrarily deep in Coq terms. For example, according to what is described in [chapter 8](#), a lemma `lem:∀(a : object C), morphism C a a` leads to a graph with a single node and a single loop around it¹³. The loop is annotated by `lem ?0`. Therefore, we need to be able to substitute under the application. To support this, we must expand the structure of the atoms to represent an application. Because our substitution is typed, we need to replace the atomic definition presented in [figure 10.2](#) with that in [figure 10.3](#).

```

a := EConstr c
    | Evar e
    | App a a*
    | Category C
    | Object x
    | Functor F
    | Morphism m1
    | Equality p1

```

Figure 10.3 The new AST for atomic values

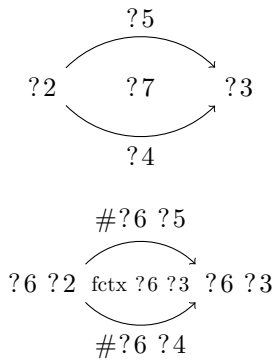
The new constructors are necessary because we can now substitute deep in an atom, injecting categories, or objects, or...However, we have to deal with the fact that our AST can have multiple representations for the same object. For example, if we have a morphism m as an element of our AST, we can always create the element `Atomic (Morphism  $m$ )`, which is also an AST for the same morphism.

These problems can be overcome. However, extending the AST means that the parser and the pretty printer need to be extended. Because we do not support implicit arguments, so the printed terms are unreadable. Furthermore, since existential variables were represented by their number, in the graphs for lemmas where almost all terms are existential variables we would have numbers on all objects and edges, making them incomprehensible, as illustrated in [figure 10.4](#). All these problems could have been solved by introducing new mechanisms, but the implementation effort seemed wasteful¹⁴ when the interface is already relying on Coq, which already has all the desired features, and more. In addition, because the features we need to reimplement are fundamental, we believe it is reasonable to expect that any other proof assistant we may want to support would already have them.

10.3 The data model

All the limitations on the previous came from the fact that it abstracted the proof assistant away too much, preventing the interface from reusing its features. The new model was created to correct this, by giving the interface a higher level view of terms, allowing the protocol to express operations on those terms to be carried away by the proof assistant. In addition, the representation of terms as ASTs made sense for a proof assistant based on the λ -calculus, but it would not have

14. Another consideration against adding too many mechanisms is that they mean that whenever a new constructor is added to the ASTs, for example to support natural transformation, the number of places to be changed in the source code would increase. We want to make it as easy as possible to implement support for new constructors, if we are to support a more significant subset of category theory.

**Figure 10.4** A lemma in the initial model.

made as much sense for a proof assistant like Isabelle. We hope that this new model is more abstract, reducing the overhead of supporting new proof assistants¹⁵.

The main notion on which this model is based is that of an object. An object is an abstract value that represents a Coq term. It is identified by a pair of integers, the first being the context and the second the *id* of the object in this context. A default context of value 0 exists, which is the context of the proof. Almost everything occurs in this context. The other contexts correspond to the lemmas, as the objects living in them might only make sense there.

10.3.1 Querying

Although an object is an abstract value, certain information can be queried. The main information that can be queried is the label of the object, which is a string resulting from the pretty printing operation of the proof assistant. This allows the interface to display the object reusing all the Coq notations. Another string can be queried, the name hint. Indeed, in the interface, objects must be named (the naming mechanism is described in [section 13.1](#)), and this hint allows the name in the interface to match those in Coq as closely as possible, to make the integration between the two as seamless as possible. Another relevant piece of information is whether the object is closed with regard to existential variables, is an existential variable, or contains some existential variables. Because the presence of an existential variable denotes a goal that remains to be solved, having this information allows the interface to display the remaining goals more precisely.

In addition to the information described above, there exists a system of *features* against which an object can be queried. A feature is a tag that identifies the feature along with n -uplet of objects. The value of n depends on the tag and is called its *arity*. Feature tags are listed in [figure 10.5](#). To query an object, the id of the object along with the tag of the feature is sent to the proof assistant, which responds with all the n -uplets that are valid for this object and tag. For example, if we have object 42 which we know is a morphism, we can query it for the tag `ComposeMph`. If the plugin answers with $\{(0, 1, 2, 3, 4, 5)\}$, we know that this morphism is actually a composition in the category 0, of the morphisms 4 and 5, with the source of 4 being 1, its target being 2 and the target of 5 being 3.

The main advantage over the AST approach is the extensibility. Indeed, adding new tags to query more information does not change the object type. Furthermore, if a plugin for a proof assistant has not been updated, it would still work by simply returning the empty set for every tag that it does not understand. Furthermore, this also means that an object can be many things at once. For example, the same object could be seen as a morphism in many categories and be a composition in the first but an identity in the second. Alternatively, an object in a category could also be a morphism in another or even a functor¹⁶.

The drawback of this generality is that the interface must be more careful when manipulating these objects. Assumptions are made to make things easier. First, if we fix the category, then most things must be unique. This includes the source and target of the morphism, if

15. Despite all this talk to support many proof assistants, no attempt has been made to implement a plugin for another, so everything we say in regard to this has not actually been tested.

16. While this will not happen, this is interesting creating categories from objects/morphisms in other categories is something common in category theory, and would allow functors to be seen as both functors and morphisms in the functor category.

it is a composition, and other properties. However, these constraints were guided by pragmatics during the implementation, so there is no consistent list.

Finally, we require a method to compare the objects for equality. Indeed, while the objects are supposed to be deduplicated, modification of the state can make the values of previously different objects equal. However the objects, as integers, are still different. Therefore, we introduce a new query to get the *representative* of an object. If we consider the objects to be quotiented by the equality of values, the representative is an arbitrary object of the class that is consistently chosen for all objects of the class¹⁷.

To summarize, once we have an object, we can obtain the following information about it :

- ◊ The result of its pretty printing.
- ◊ An optional name.
- ◊ Whether it contains existential variables or not.
- ◊ The different features it possess.
- ◊ Its representative.

17. In the current implementation of the Coq plugin, the representative is simply the smallest object by id whose value is equal to the current object. However, no such assumption is made in the interface, to allow the plugin to change its representative selection method if a more efficient method is discovered.

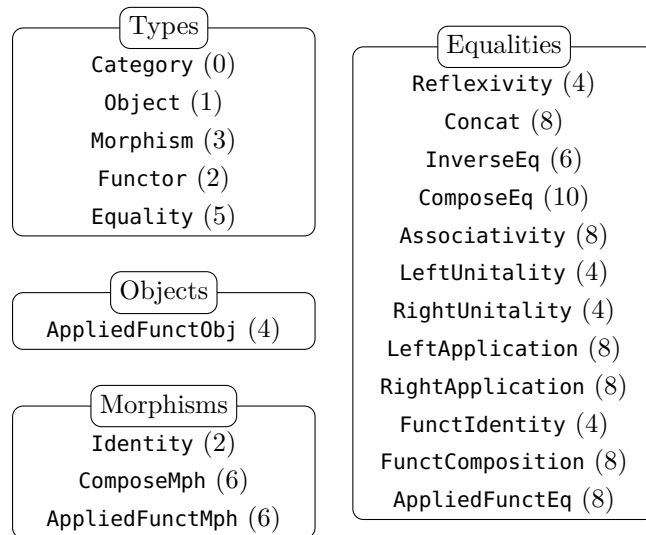


Figure 10.5 The feature tags with their arity, grouped by kind

10.3.2 Building

Until now, we have only described how to obtain information about objects, but not how to obtain those objects in the first place. There are three main methods to obtain new objects.

First, the proof state of the proof assistant is converted to a graph, structure of which is described in [section 7.1](#). This graph can be obtained by the interface. Such a graph is annotated with objects living in the main context.

The plugin exposes the parser in a limited fashion to support the creation of new objects from the interface. The interface can send a string to the plugin, which attempts to parse it. On success, it adds the `EConstr.t` to the object store and sends back the id of the newly created object. On failure, it sends an error message as a string, so that the interface can display the error to the user.

Finally, one can build new objects from the previous ones. To do so, because there is no more AST to describe terms in an abstract way, we need a method to describe how we want to build new objects. We found that *features* can also be used for that. Indeed, one can query an object to check if it has some features, but there is enough information in a feature to build back an object from the feature information. Thus, there is a way to send a feature to the plugin, which will try to build a term from it, and on success, will send back the associated object.

For most features, it is clear which object will be built. For example, the `ComposeMph` feature constructs the composition of two morphisms. However, *type features* do not correspond to specific terms. For example, there is no canonical way to construct a term for the feature `Morphism`. So for the features that describe the type of the object instead of the object itself, what will happen is that the plugin will create an existential variable whose type is the one described by the feature. Thus, using this approach, we have a unified way to construct new terms as well as new existential variables.

10.3.3 State

The state of the proof is the set of objects along with their features. Therefore, any operation that creates or modifies an object is considered to alter the state. Any state modification must have the following three properties :

- ◇ It cannot remove objects.
- ◇ It can add features to objects, but not remove them¹⁸.
- ◇ It can make new objects equal, but cannot make previously equal objects different.

To guarantee that the operations on the state are very restricted, but this is necessary to prevent the interface from having to always check if the objects it holds are still alive. This also allows for aggressive caching of the features of objects¹⁹. The building of objects described in the previous section modifies the state by adding new objects, and they immediately respect the three properties.

Then, there is one more operation needed that modifies the state : unification. The interface can send a pair of objects to the plugin, which will then attempt to unify them. The interface is notified of the success or failure of unification, but nothing is returned. Indeed, the objects are abstract; therefore, there is no notion of substitution or unifier in the model. Instead, when unification succeeds, all the objects present in the state are modified by side effect²⁰. This operation also respects the three properties.

To support undoing operations in the interface, we need methods to save the state at some point, and restore it. When the state is saved, an identifier for this state is sent to the interface, which can request the state to be rolled back to it. When the state is rolled back to a specific state, all the states that were saved after are discarded, so that the saved states form a stack of states, and not a tree. Saving and restoring states only modify the default context, all other contexts remain unchanged through these operations.

18. This is a simplification. Indeed, some features can be replaced by other features with the same tag, where the objects constituting the feature are replaced by objects that are equal to them. Thus, it does not remove features *if they are considered modulo equality*.

19. Theoretically it would allow for more aggressive caching than what is currently implemented. However, getting caching right is difficult, and for now the performance gain has not been felt as necessary.

20. In the case of Coq, the objects do not need to be modified. Instead, new bindings are added to the state and objects are lazily substituted when analyzing them. Further details are provided in [section 12.5](#). Of course, in a term based proof assistant that does not have this kind of mechanism, substituting all terms in the object store is a valid way to implement it.

10.3.4 Lemmas

Each lemma is associated with a context, and for simplicity sake they are represented by the same integer. Thus, a lemma is identified by an integer that is different from 0 because 0 denotes the default context. The interface can obtain the list of lemmas. The information given for each lemma is its identifier, its name as a string, and its namespace as a list of strings.

An additional assumption is that the name plus the namespace uniquely identifies a lemma. The namespace is a string list that allows the interface to display lemmas in a tree, making them easier to find²¹.

It is also possible to obtain the graph of a lemma, as described in [section 8.2](#). There are two ways to do so. The first gives the graph annotated with objects living in the context of the lemma²², while the second gives objects living in the default context. The reason for this distinction is that when the graph is created, new existential variables are created, changing the context. However, if we only want the graph to visualize it, we do not want these existential variables to be added as proof obligations. Furthermore, we want to be able to visualize a lemma and progress the proof simultaneously, so the operations modifying the context must not modify the objects living in the lemma. However, when we apply a lemma, we need the objects annotating its graph to live in the default context so that they can be unified with the objects annotating the goal graph.

21. In Coq, the namespace list is the path in the tree of modules, but we could imagine the plugin creating its own hierarchy instead, especially if it is used in a proof assistant that does not have such a built-in hierarchy.

22. Since the pretty printing of objects is handled by Coq, which stores information about the names of the existential variables, the graph of [figure 10.4](#) now looks like the one in [figure 10.6](#).

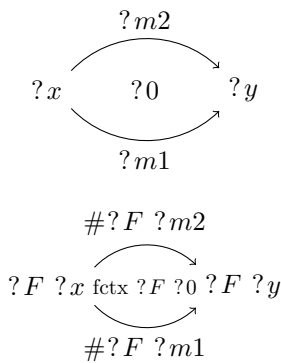


Figure 10.6 A graph lemma in the new data model.

10.4 The protocol

This section described how we concretely encoded the previous model. It aims to provide a sufficiently low-level understanding for reimplementing a server for this protocol. However, from a theoretical point of view, all of the ideas were already described in the previous section, so you can safely skip this section.

This section is divided into three sections. The first, [section 10.4.1](#), describes how the bytes making up the messages exchanged between the plugin and the interface are interpreted. This provides a description of the messages in a more abstract structure. In the second section, [section 10.4.2](#), we describe in terms of this intermediate structure how the interface can call procedures in the plugin remotely, and gives a higher level description of the data being exchanged. Finally, in [section 10.4.3](#), we describe the procedures that the plugin must make available to the interface. This section links these descriptions with the high-level descriptions in the previous section.

10.4.1 MessagePack

As the basis of our protocol, we use *MessagePack*, an efficient and easy to parse system for serializing data into bytes. Similar to *JSON*²³ or *XML*; however, unlike systems such as *atdgen*, it can only serialize a predefined set of structures. Therefore, to serialize any structure, they first need to be encoded in structures supported by MessagePack. In this section, we describe the structures supported by MessagePack, and how they are encoded.

23. The *JavaScript Object Notation*, a very common format that can represent most JavaScript objects but is used beyond JavaScript.

MessagePack supports the following :

- ◊ An *unit* object, called *nil*.
- ◊ Two objects for *true* and *false*.
- ◊ Integers, both signed and unsigned, composed of up to 64 bits²⁴.
- ◊ Floating point numbers in double precision.
- ◊ Strings of *almost* arbitrary length.
- ◊ Sequence of objects²⁵.
- ◊ Map of objects to objects.
- ◊ Arbitrary binary data along with an extension mechanism to tag it²⁶.

The complete specification of the MessagePack protocol can be found online⁽¹⁹⁾, so we will not go too much in detail. The main principle of this protocol is that each object is serialized as one byte, acting as a *tag*, giving the structure of the objects, followed by the data. In the cases where the data is of variable length, the number of objects expected is included at the start. For example, a sequence is first the sequence tag, followed by the length of the sequence, and then the serialization of the objects of the sequence concatenated²⁷. For strings, the prefixed length is that of the string in *bytes*, and the string itself must be UTF-8 encoded.

The cleverness of MessagePack comes from the realization that most arrays, strings and integers are small in usually exchanged data. Therefore, instead of representing each integer as 8 bytes, there are different tags for 8, 16, 32 and 64 bit integers. Furthermore, if the integer is sufficiently small, it can be stored directly in the tag. The idea of having the first *bits* of the tag *byte* being enough to determine the structure of the object means that the remaining *bits* can be used to store data. This allows the tag and size of strings less than 31 bytes long to be stored together in a single byte for example.

This allows the serialized objects to be very compact while still being easy to parse. Indeed, the logic to parse the tag byte is only slightly more complicated, rather than looking only at its value, one needs to look at its value with different byte masks applied, but then extracting the value is straightforward. When serializing data, one can choose an encoding that optimizes the size. This is not very complicated, and does not need to be optimal. Indeed, using 9 bytes to encode the value 5 is a bit wasteful, but it is correct, so depending on how much effort one is willing to make, *best effort* encoding is sufficient²⁸.

10.4.2 Remote Procedure Call

Now that we have described how the plugin makes sense of the bytes they exchange, we need to describe the dynamics of these exchanges. We use a *Remote Procedure Call* (RPC) system. In other words, the server exposes a certain set of functions that the client can call through the communication channel.

In the case of our plugin, the plugin itself is the server, exposing functions that allow the interface to query the state of the proof assistant and modify it. The calls are encoded using the format of *msgpack-rpc*⁽²⁰⁾. According to this standard, there are two kinds of message : calls and responses²⁹.

24. This causes difficulties in the Coq plugin. Indeed, because it is written in OCaml, it only natively supports 31 or 63 bit integers because the remaining bit is used to distinguish integers from *boxed* objects. For simplicity, the decoding fails if an exchanged integer cannot be represented in 63 bits, which is sufficient because it will never happen in practice, but it is a bit unsatisfying.

25. Since MessagePack has only one type, which is the type of objects, the elements in the sequence can have any structure. There is no notion of sequence of integers for example. Therefore, it is better understood as an *n* tuple than as an array.

26. We do not use these, the previous notions are sufficient for our needs.

(19) MessagePack protocol

27. This design makes it easier to write efficient parsers that can preallocate memory during parsing. However, this means it is less suitable for applications in which one might want to stream data without first computing it completely. This does not concern our particular usage, and the ease of writing parsers is welcome.

28. For example, we found only one library in OCaml that implemented MessagePack. However, it was just one feature of many in this library, and along with the difficulty of using libraries for Coq plugins, it seemed more efficient to reimplement the support for MessagePack ourselves. The final implementation is only approximately 100 single lines of code long and does not bother finding the most efficient encoding.

(20) MessagePack Remote Procedure Call

29. In fact, there is another kind of message, *Notification*, which allows the server to send a message to the client unprompted and which does not expect any response. We will not discuss this because it was not needed for our plugin.

30. Quadruplets are not part of the `MessagePack` structure, so we mean a sequence of length 4.

A remote call is a quadruplet³⁰ composed of the following information :

- ◊ An integer, that must always be 0 for calls.
- ◊ An unsigned integer that must be unique for each call.
- ◊ A string, identifying the procedure to call.
- ◊ A sequence of arbitrary objects, that are the arguments of the call.

The server can then respond to a call with another quadruplet made of :

- ◊ An integer that must always be 1 for responses.
- ◊ An unsigned integer that must be the same as that of the call this response is answering.
- ◊ An object that is either *Nil* in case of success or another object denoting an error. In our case, we restrict this object to be a string so that the interface can display it to the user.
- ◊ An arbitrary object that is the result of the call. If an error occurs, then it must be nil.

Such a system has a few concrete advantages, in that it supports *asynchronous calls* and *server pipelining*. Asynchronous calls mean that the client can send another call before the server has finished answering the first, enabling the server to work on multiple requests simultaneously. Server pipelining is an additional relaxation that allows the server to answer the requests it receives out of order, which is possible because the responses carry the unique id of the calls³¹.

31. Currently, the implementation does not use any of these possibilities. Moreover, the interface blocks while waiting for the answer from the server. We are currently refactoring the interface to remove this limitation, which is the first step in taking full advantage of the capabilities of the RPC system.

Up until now, we described the *MessagePack-RPC* protocol. However, because it is meant to be generic, it accepts arbitrary objects in quite a few places. In our use cases, we do not need such power, so we describe the structure of the objects we do exchange. There are only six types of objects that we exchange over the protocol :

- ◊ Proof-assistant objects, that are encoded as an unsigned integer identifying them. Proof assistant objects make sense only in a specific context. When the context is not specified, it is assumed to be context 0, which is the goal context.
- ◊ Context or lemmas, that are again encoded as unsigned integer. The integer for a lemma must be non null, as the 0 integer always identify the context of the goal.
- ◊ The evar status, which is an integer of value either 0, 1 or 2. 0 indicates that the queried object is an existential variable, 1 indicates that it is closed with regard to existential variables, and 2 indicates that it is neither.
- ◊ Feature tags that are encoded as strings that name them³².
- ◊ Features, which are a $n + 1$ -uplet where the first element is a feature tag, and n is the arity of this feature tag (see figure 10.5). The other n elements are proof assistant objects.
- ◊ And finally graphs. A graph is a triplet (a sequence of length 3). The first element is a sequence of proof-assistant objects, the nodes of the graph. The second is a sequence of triplets, where the first element is the index of the starting node, the second element is the index of the target node and the third element is a proof assistant object. The third is a sequence of faces. A face is a quintuplet where the first two are the indices of the source and target nodes of the faces. The next two elements are the sequence of indices of morphisms comprising the left and right sides of the face. The final element is the proof assistant object of the equality.

32. We can enumerate the feature tags and encode them as integers. However, this would need to be done more carefully when adding new ones or supporting only a subset in the plugin. We believe that this approach is less bug-prone when the plugins are developed separately from the interface, which is not the case now but may be in the future.

In addition to these objects, strings or sequence of strings are sometimes exchanged directly. Furthermore, sometimes the values are optional, meaning that *nil* is a valid value that represents the lack of an actual value.

10.4.3 The API

Finally, we have reached the point where we can describe the calls that must be supported by the plugin for the interface to work. Since it follows pretty closely what we described in [section 10.3](#), we will also split the calls into four groups : querying, building, state and lemmas.

◇ Querying

- ◇ *info*
Takes two arguments, a context and a proof assistant object, and returns a triplet containing the pretty-printed object as a string, an optional string naming the object³³, and the evar status of the object.
- ◇ *query*
Takes three arguments : a context, a proof-assistant object, and a feature tag. It returns a sequence of features. This sequence of features must include all the features of the object with the queried feature tag, but it can also include additional features³⁴.
- ◇ *repr*
It takes a context and a proof assistant object and returns a proof assistant object, which is the representative of its argument.

³³. This is used to ensure that the name of objects in the interface match those in the proof assistant as much as possible, to reduce the impedance of going back and forth between them.

³⁴. Accepting additional features allows the reduction of the number of queries when computing a specific feature and also gives us some other features for free. We have not used this to date.

◇ Building

- ◇ *goal*
Takes no arguments and returns a graph. The objects in this graph must be valid in context 0. This graph is extracted from the proof context, and the existential variables appearing in it are proof obligations.
- ◇ *parse*
Takes a string as an argument, and returns an optional proof assistant object³⁵. The returned object is in context 0.
- ◇ *build*
Takes $n + 1$ arguments, where the first argument is a feature tag, the n following arguments are proof assistant objects, and n is the feature's arity. Return a proof assistant object³⁶.

³⁵. As it is now, a failure to parse does not transmit an error message to the interface; therefore, the interface can only say that the parsing failed but it cannot explain why the parsing failed. This could be improved in future versions.

³⁶. We consider that every time *build* is called, it is called with arguments that make the build possible, so that if the build fails, for example, owing to a type error, it is a bug in the interface. As such, the call always returns an object, and in the case of failure we use the RPC protocol way of handling errors, which will cause a panic in the interface.

◇ State

- ◇ *unify*
Takes a sequence of pairs of proof assistant objects and returns a boolean. If the returned value is true, all objects in context 0 are modified so that the unified pairs are equal. If the returned

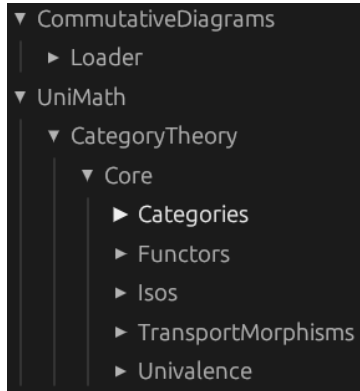
value is false, it means that at least one pair is not unifiable and the context 0 is unchanged.

- ◇ *save_state*
Takes no arguments and returns an unsigned integer identifying the saved state.
- ◇ *restore_state*
Take an integer as argument, which must be a value previously returned by *save_state*, and returns nothing. It restores the state by side effect.
- ◇ *finish*
Takes a boolean as arguments, and returns nothing. It notifies the plugin that the interface has been finished. If the argument is true, it means that the interface succeeded, and the changes should be committed to the proof state. Otherwise, the tactic should fail.

◇ Lemma

- ◇ *lemmas*
Takes no argument and returns a list of triplets. Each triplet represent a lemma. The first element is an integer that identifies the lemma and its context. The second element is the string that names the lemma. Finally the third element is a list of strings that provide a namespace for the lemma. This allows displaying the lemmas in a tree³⁷. We expect the pair of namespace and name to identify the lemma uniquely.
- ◇ *pattern*
Takes an integer identifying the lemma, and returns the graph of the lemma where the objects must be interpreted in its context.
- ◇ *instantiate*
Works the same as *pattern*, but the graph must be interpreted in context 0.

37. *UniMath* example :



11 Algorithms and data structures

This chapter presents an inventory of most of the interesting algorithmic problems and algorithms that we had to implement or consider during our development. The sections are mostly independent from each other. Some of the algorithms described here were never implemented, either because of a lack of a satisfying solution or a lack of time.

11.1 Graph extraction

Even when our development was composed of only the solver (see [chapter 9](#)), the structure we operated on was that of a graph, or as described more precisely in [chapter 7](#) of an annotated 2-quiver. However, when our plugin starts, it only has access to Coq's proof context, which we need to transform into a graph.

The context of Coq can be accessed sequentially, as a list of variable definitions, each of which is a name, a type, and optionally a value¹. The goal is then given by its type and an existential variable.

Therefore, for each object we have a pair of Coq terms, a value, and its type. For hypotheses, the value will simply be the variable, and for the goal, the value will be the existential variable. When dealing with lemmas, because we need to support objects under quantifier, we wrap Coq terms.

Therefore, our graph extraction algorithm is parameterized by an arbitrary type for terms. We need some structure on this type. The structure we want is such that this type is ordered, which enables us to create maps from this type. Furthermore, we do not want the equality obtained from this order structure to be the structural equality of the type itself, but to correspond to a more semantic equality, in the case of Coq terms to the equality modulo reduction². However, there is no obvious way to order Coq terms in a way that is compatible with reduction. To circumvent this, we create an array that stores all the Coq values we encounter, and whenever we encounter a new value, we first check if it is present in the array³. Then, we can use the array index instead of the Coq values, which is ordered and the equality corresponds to the equality procedures we used to check the presence of the value in the array⁴.

We need a structure that supports the addition of new terms. If the type of the term is an object in a category⁵, we add a new node; if it is a morphism, we add two new nodes corresponding to the source and target values, and then an edge for the morphism. Finally, if it is an equality between morphisms, we split the two sides of the equality along morphisms, add all the components as new edges, and finally add the equality as a new face, with sides being the list of morphisms. This procedure is illustrated in [figure 11.1](#). To simplify this procedure, the nodes, edges and faces of the graph are simply lists, and are referred to by their index⁶.

Each list of objects is enriched with a map from the objects to their index in the list. Thus, whenever a new object is added, we can quickly check if it is already present and prevent adding multiple nodes annotated by the same object, for example. This means that the graph we build is that with minimal deduplication, which may not

1. We do not need or use the value of variables because we always refer to them by name. When we identify Coq objects modulo reduction, the body is used by the Coq reduction engine.

2. We simply say modulo reduction, but there are many kinds of reductions in the Coq engine. The reason we keep it imprecise is that it does not matter to the algorithm itself and that it is subject to change with our understanding of the Coq internals and performance trade-offs.

3. How this is done is described in more detail in [chapter 12](#).

4. It has the additional advantage that the order is really efficient to compute, as we do not need to traverse the entire AST of the values to compare them.

5. The exact way we parse types to recover this information is described in [chapter 12](#).

6. In contrast to, for example, an adjacency list representation, where to refer to an edge you need both the index of its source and its index in the list of outgoing edges for this node.

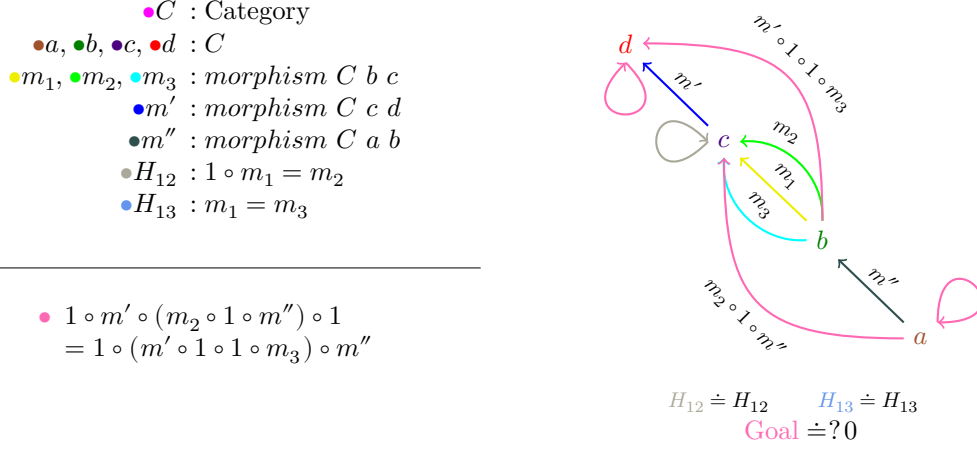


Figure 11.1 Example of the procedure. Each object in the graph is coloured according to the step it is added. The steps correspond to the variables in the hypothesis and to the goal, the color of each is indicated by the colored bullets.

7. An example of a case where minimal deduplication is not desired is when taking the product of an object with itself. The current procedure gives the graph in figure 11.2 whereas the expected graph is shown in figure 11.3.

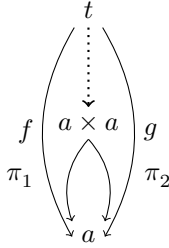


Figure 11.2 Deduplicated self product

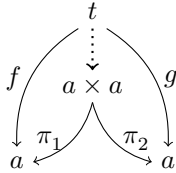


Figure 11.3 Self product graph

8. See section 10.1 for a description of the distinction we make between plugin and interface.

[40] Tarjan and Van Leeuwen, *Worst-case Analysis of Set Union Algorithms*, 1984

always be what is desired by the user⁷. Indeed, the graph contains more information than simply the context, and we decided to make this choice. If it does not correspond to what the user wants, they can manually fix it through the interface.

There is one final limitation that must be addressed. When splitting the side of an equality along compositions, we can do so in a naive way, that is, recursively splitting only on the side of the default associativity, or we can recurse on both sides. Furthermore, we can simplify the identities when doing so. However, when doing so, we would need to change the equality term because such a process would amount to normalizing both sides of the equality modulo associativity and unitality. Because we want to keep all the complicated algorithms in the interface and not the plugin⁸, we decided against it. To compensate for this, the interface performs a normalization step upon receiving the graph, where it removes all identities⁹. We decided to keep the edges annotated by composition instead of automatically splitting them up, because we interpret a sequence of morphisms with parentheses around them is an indication of intent from the user that they should be considered as an atom. If this is not the case, it is very easy to split the morphisms in the interface.

11.2 Proof relevant disjoint set

The *disjoint set* or *union find* structure is a highly efficient structure for modeling a partition of a set. It supports two operations : the union of two subsets, identified by any of their objects, and finding a representative for each subset. If implemented correctly, both of these operations are in $O(\alpha(n))$ worst case amortized time where n is the number of elements in the set and α is the inverse Ackermann function [40]. Therefore, the complexity of these operations can be considered constant for all purposes¹⁰.

The basic structure consists of an array of cells. Each cell stores an object, which is referred to by the index of its cell. Each cell also stores an integer that is the index of its parent cell. The idea is that every equivalence class of the partition¹¹ is represented by a tree, where each element stores its parent in the tree and the root is the element whose parent is itself.

Finding the representative of an element consists of following the parents until the root is found, and taking the union of two equivalence classes is making the parent of one of the roots the other root. To achieve complete performance, two optimizations are required, which are not described here.

The reason this algorithm is not sufficient for our needs is that we do not simply need to find the representative, will also need an equality proof between the queried object and the object of its parent. To do so, more information needs to be stored in each cell. In addition to the index of the parent, each cell stores an equality proof between the object of the current cell and the object of its parent. Because the objects we consider are paths in the graph, the equality must be between the canonical realization of the path as morphism¹².

Initialization of the structure is still possible; since every object is its own parent at the start, we can take the reflexivity as a valid equality proof. Then, when querying a representative for an object, we use transitivity to construct an equality proof that concatenates the proofs along the way.

Finally, when connecting two objects, we must provide an equality proof between them. However, we do not connect these objects but their representatives, so we need is an equality between the representatives. We construct it by applying the equality between the first object and its representative in reverse, which we can compute, then the provided equality, and finally the equality between the second object and its representative.

Thus, we can always provide an equality proof between any two connected objects. Furthermore, it is interesting to note that the axioms we needed were the three axioms of an equivalence relation : reflexivity, transitivity, and symmetry.

11.3 Graph layouting

11.3.1 First approach

After the combinatorial structure of the graph has been extracted from the proof goal, we need to layout it to display it to the user. Initially, *graphviz*⁽²¹⁾ was used. Graphviz is a command line tool that takes a description of a graph in a textual format, and can output a representation in many formats. The most useful is a *JSON* export, which describes the coordinates of the nodes and edges. The edges are described as sequences of cubic Bezier curves¹³. By reading this information, we can store it in the graph structure and then not only display it ourselves, but also detect clicks on specific parts of the graph.

However, this approach has two limitations. First, the layouting of the graph is fully non interactive. It is impossible for the user to place some things manually, and then have graphviz place the rest. Second,

9. This explains some unusual behavior in the displayed graph. Indeed, if there are both a m and a $m \circ \text{id}$ edges in the graph created by the plugin, after normalisation there would be two parallel m edges. It is easy to deduplicate them interactively, but we did not implement a procedure to do so automatically during the normalization step.

10. Indeed, $\alpha(2^6 4) \leq 5$, and because the memory usage is linear in the number of elements, we encounter other problems before we reach that point.

11. Every equivalence relation corresponds exactly to a partition of a set; therefore, we will use the two terms interchangeably.

12. Section 11.6 describes a representation of equality directly between lists of morphisms.

(21) Graphviz

13. This is something that is not clear in graphviz documentation. An edge is described by a sequence of points, whose length is always $3k + 4$. The points in positions multiple of 3 are the endpoints of the curves, and the two points between them are the control points of the Bezier curve. It is thus a sequence of cubic Bezier curves, but since the curve is continuous, the endpoint of each curve is the starting point of the following curve and is not repeated.

[41] Kobourov, *Force-Directed Drawing Algorithms*, 2013

14. In practice, it is a gross overstatement to call it a physical simulation. Indeed, while the intuition is physical, and the *forces* applied take inspiration from physical systems, the force vectors are applied directly to the position of nodes at each step, instead of having each node have a speed and forces changing this speed. Therefore, nodes have no inertia, breaking the analogy with physical systems.

15. We say *mostly* because the simulation itself is linear, so the complexity depends on which forces are to be applied, and thus depends on the specific flavors of the force directed algorithm chosen.

16. A non converging layout is one where a node overshoots its position at each step, which results in it oscillating around a position.

17. This kind of trade-off is very common in optimization problems, and is sometimes solved using *simulated annealing*. In simulated annealing, the speed of the simulation is controlled by a parameter called *temperature*, which decreases as the simulation progresses. It enables the simulation to go fast when it is far from its target state, and becomes more precise as it gets closer. We did not implement this method, but it should not be too complicated.

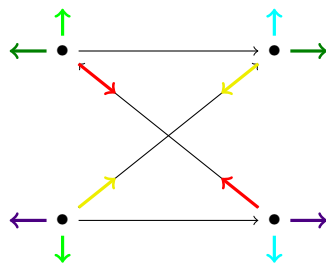


Figure 11.4 A square stuck in a non-optimal layout. The forces trying to bring the edges closer along the diagonals and the force pushing them away along the top and bottom sides compensate each other perfectly. Symmetric forces between pairs of nodes have the same color.

[42] Kamada and Kawai, *An algorithm for drawing general undirected graphs*, 1989

it was discontinuous. This means that every time something changed in the graph, graphviz was called again and a completely new layout was created, making it difficult to understand the progressive changes.

11.3.2 Force directed approach

Therefore, we decided to implement our own graph layouting algorithm to better integrate it with the interface. There are many approaches to graph visualization and layouting. We decided to use a *force directed* approach [41]. Force directed approaches are based on the idea of placing constraints on nodes and performing a physical simulation¹⁴ to let it converge to a hopefully good layout. It has many advantages :

- ◊ It is simple to implement.
- ◊ By displaying the intermediate states of the simulation, the user can see the graph change shape continuously from a previous state to a new one, resolving the previous problem of continuity.
- ◊ It is flexible, and new constraints can be added to the simulation to direct the resolution.
- ◊ Its complexity is mostly quadratic¹⁵ but the constants are low; therefore it is quite fast for small graphs.

However, this approach has some limitations. The main one is that the quality of the layout is lower than what would be obtained with something like graphviz. If one is unlucky, it can become stuck in position similar to the one depicted in [figure 11.4](#). Furthermore, it can be quite fiddle and struggle to converge with some set of forces¹⁶. Finally, it is quite difficult to find the right simulation speed. Indeed, a higher speed means that the graph will converge faster, but it will also make the simulation less precise, sometimes resulting in a non-converging layout¹⁷.

Another problem is non-determinism. Indeed, across restarts of the interface, the resulting layout is not always the same. There are two sources of non-determinism. The first is the handling of nodes in the same positions. Indeed, some forces scale with the inverse of the distance, so we need a way to avoid an explosion of forces when the nodes are too close. In this case, an arbitrary direction is chosen to push the nodes away. At first we used a random direction; however, to make the algorithm more deterministic, we now use a *pseudo-random deterministic* method¹⁸ to compute the direction. However, this is not sufficient to eliminate the non determinism. Indeed, to maintain a constant speed on screen, the simulation speed is scaled by the time elapsed since the previous step; therefore, it will depend on the refresh rate of the interface. Of course, this will have some random variations, resulting in deviations between runs. We cannot do much about this, and since the layout itself is not proof relevant, we did not really try to find a solution. Anyway, as soon as the user interacts with the graph by dragging nodes, for example, deviations will be created.

11.3.3 Forces

We experimented with spring based and barycentric approaches, but we could not obtain reliable results. We settled on an approach based on *graph-theoretic* distance [42]. The idea is that we compute the graph distance between any two nodes¹⁹, and then between any pair of

connected nodes we add a spring force²⁰ where the nominal length of the spring is proportional to the graph theoretic distance between the two nodes. This approach provides a good layout that converges quite quickly, but it does not add any force between disconnected nodes, so it only works on connected graphs.

To address this limitation, we compute the connected components of the graph, and store the position of each node relative to the position of the connected component. That way, the previous forces provide a layout for each connected component, and we now apply forces to the positions of the connected components to place them relative to each other. The idea is to compute the bounding box off all the nodes and control points of edges of each connected component. Then, if two bounding boxes overlap, we determine the direction in which the overlap is smaller and add a force that pushes them away proportionally to the size of the overlap. This is illustrated in figure 11.5.

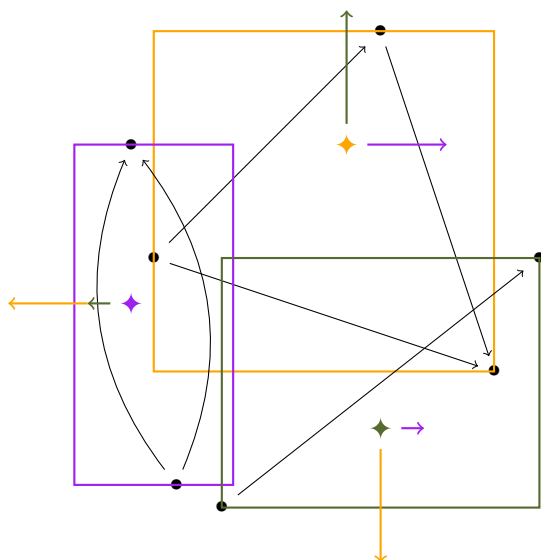


Figure 11.5 Forces on the connected components. The origin of the connected components is denoted by a coloured $\blacklozenge$. The forces on the origins are coloured by the connected component causing this force. Notice how the origin of the connected component is not necessarily the center of its bounding box.

Until now, we have only described the placement of nodes. To keep things simple, we restrict the shape of the edges to quadratic Bezier curves²¹. A quadratic Bezier curve is parameterized by its endpoints, which are the positions of nodes, and a single control point. Three types of force are applied to the control points :

- ◇ A logarithmic²² force towards the midpoint between the two extremities of the edge. This pushes the edges into straight lines between their endpoints. These are the orange forces in figure 11.7. If the edge is a loop, we also add a force pushing it away from the node to prevent the edge from becoming too small.
- ◇ Between any pair of parallel edges, we compute the projection of their control point along the bisection of their endpoints²³. We then apply a force that pushes them away along the bisection proportional to the inverse square root of the distance of the projection²⁴. There

18. This is a pedantic way to say that we took a few integers parametrizing the node, summed them together, multiplied them by 17 and took the rest of the division by 31. Indeed, we do not need cryptographic grade randomness, so the actual solution is sufficient for our needs.

19. Of course, we have a directed graph, but we disregard that for this computation. It is defined as the minimal number of edges, taken in any direction, that connects the two nodes.

20. What we call a spring force is a force that follows *Hooke's law* : a force proportional to the difference between the actual distance between the two objects and a nominal length. If the objects are closer than the nominal length, they are pushed away; otherwise they are pulled together.

21. There one limitation with quadratic Bezier curve : they cannot represent a loop where the two endpoints are the same. We decided that for the simulation, it was sufficient. However, when drawing a self loop, we convert the quadratic Bezier curve into a cubic one. This conversion is shown in figure 11.6.

22. If the control point is too far, we make the force linear to accelerate convergence.

23. If the endpoints are the same, bisection is not well defined. In this case, we simply push the control points away linearly, without projecting.

24. There is a special case when the projections are too close to avoid propelling them away from each other with too much force.

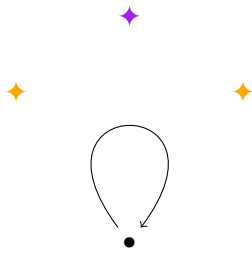


Figure 11.6 Drawing a self loop. The control point of the quadratic Bezier curve is in purple and the control points of the computed cubic curve are in orange. The resulting direction of the loop is arbitrary.

- are the purple forces in [figure 11.7](#). These forces tend to make parallel edges form *eyes* between the endpoint nodes.
- ◇ Finally, we push away the control point of any pair of non-parallel edges with a force proportional to the inverse of the distance between them. These are the olive forces in [figure 11.7](#). This tends to make the graph *bubbly* since it means that single edges between nodes will be pushed outwards. It is useful when the graph is dense to create more space inside by pushing the outside edges to the outside. However, it prevents straight edges, which we sometimes want; therefore, these forces can be enabled and disabled interactively.

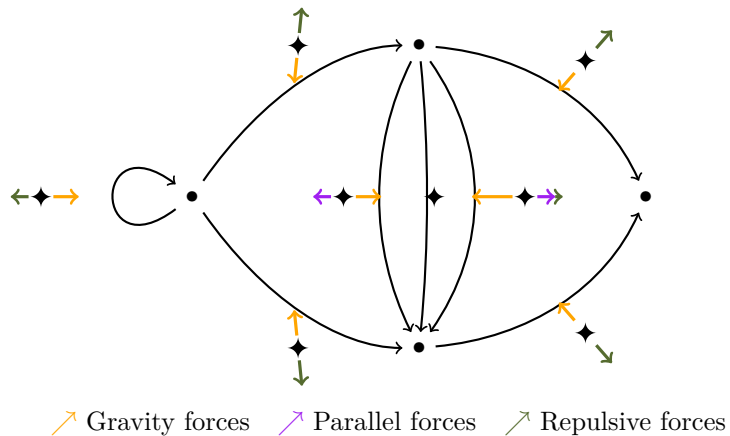


Figure 11.7 The different forces on edges. The control points are marked by $\blacklozenge$ symbols, and only the sum of each kind of force are drawn. Forces too small are not drawn either.

11.3.4 Structure

To implement this algorithm, we created a structure that stores the simulated points. Each point can have forces applied to it, and we have a procedure that computes the next position of each point from the forces applied on it. This structure also stores some precomputed information, such as connected components and the distance on the graph between nodes.

The graph then stores indexes into this structure instead of directly storing the coordinates of the points. This method has several advantages. First, when converting to positions, computations can be performed, such as adding the position of a node to that of its connected component. It also allows the creation of simulated points that do not correspond to graph objects, such as the origins of the connected components.

When computing the next step, the structure also supports the use of a predicate on points. When this predicate is true on a point, the position of this point is not changed by the simulation. This enables pinning nodes and edges, but also preventing the simulation from moving an object while it is being dragged by the user.

11.4 Planar decomposition

Given a layouted directed graph, along with two parallel paths on this graph, we want to compute a valid sequence of steps from the first path

to the second (see [section 7.5](#) for the precise definition of steps). We want this decomposition to be such that there is a step for every planar face in the region delimited by the two paths. Of course, this is not always possible, because some planar face may have its sides oriented in such a way that a step cannot be created from it, but our algorithm should give a decomposition in faces that are as small as possible.

Another expectation is that it also succeeds on graphs that are not planar. Indeed, because the user can layout the graph any way they want, we cannot guarantee that the graph will be planar. We can add a procedure to check for planarity, but it turns out that our algorithm still works in non planar situation²⁵. [Figure 11.8](#) shows some graph and the expected decomposition.

25. Work here means that it terminates and produces a valid sequence of steps, but because there is no notion of face for a non planar graph, we do not guarantee the result in this case.

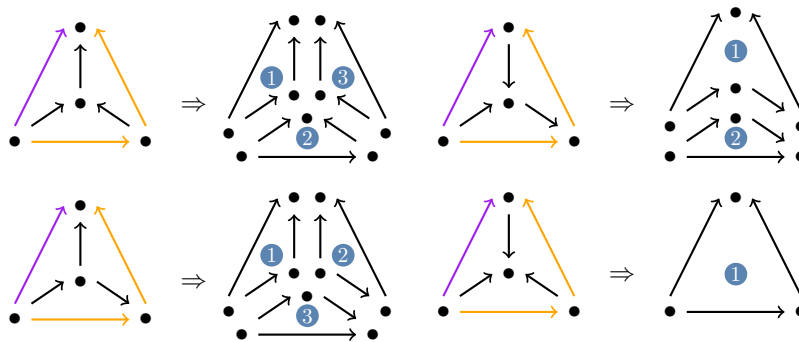


Figure 11.8 Examples of planar decompositions. The parallel paths are drawn in purple and orange on the input graph. The numbers in the result give the order the corresponding steps are applied.

The algorithm we are going to describe makes two additional assumptions regarding its input :

- ◊ It expects the given two parallel paths to be given in topological orientation. This means that if you go along the first one, and then along the second one in reverse, you must have made a counter-clockwise turn. This is not a major limitation because we can simply compute the orientation of the turn and switch the two paths if the orientation is incorrect. In this case, after the algorithm has run, we also need to reverse and flip the steps; however, this operation preserves the validity of a sequence of steps. If the graph is not planar, the orientation does not matter significantly.
- ◊ It assumes there are no loops along any of the paths. Again, this is not very restrictive, as we can simply go over each path and add a step for every encountered loop that contracts the loop. We then conjugate the returned sequence of steps by those steps to obtain a valid new sequence between the paths with loops.

While we took the idea of using this procedure to progress the proof from Ambroise's editor⁽⁹⁾, we did not reuse his algorithm. Instead, we have created our own. As described in the previous section, the edges in our layout are quadratic Bezier curves. Because we do not check edges for intersections, the only information that matters is the angle of the edge at its endpoints. This is easy to compute because the tangent of the curve at its endpoint is given by the endpoints and the control point.

⁽⁹⁾ A diagram editor in your browser

26. Because we may not go over the entire graph during this procedure, we compute these lazily as we encounter new nodes. To simplify the description of the algorithm we will assume they are all precomputed.

27. This is well defined because there are no loops along the path; therefore there is exactly one outgoing edge for every node along the path.

We begin the procedure by computing the angle of each outgoing edge for each node, and reordering the outgoing edges by angle²⁶. We do not precisely determine the origin of the angles as they are not important, because we only consider the relative angles between edges. We also store, for the nodes on the second path, the angle of the outgoing edges along this path²⁷. We name these angles the *maximum angles*, because they are the angles we must not cross when iterating; otherwise we would go out of the region delimited by the two paths.

We then start from the first path and try to create a step that pushes it along the trigonometric direction. Once we have found such a step, we store it and start again with the other side of this step, until we have reached the second given path. At that point, we have the sequence of steps that we were looking for. To do so the algorithm is divided into two procedures : `find_parallel_path` and `find_step`.

11.4.1 `find_parallel_path`

This procedure takes a node and a pair of angles, and attempts to find a path from this node that reaches again the starting path. First, it checks whether it has already encountered its argument node. If this is the case, it fails. Doing so avoids this procedure from getting stuck in loops and guarantees that the found path does not have loops. Then, it checks if the current node is in the starting path, in which case it succeeds with the empty path only if it is not the node it started from.

Once this is done, it marks the current node as seen and starts looping over all the outgoing edges from the node that have their angle strictly in the range delimited by the pair of angles²⁸ ordered by the trigonometric order of rotation. For each of these edges, it recursively calls itself on the next node. The range of angles the recursive call is given starts at the ingoing angle, and goes either back to it or to the maximum angle of this node if it is on the target path. If the recursive call succeeds, it prefixes the returning path with the considered edge and returns this new path. Otherwise, if none of the recursive calls succeed, the procedure fails. [figure 11.9](#) illustrates this procedure.

28. Angles wrap at 2π , so the range $(1.5\pi, 0.5\pi)$ is not empty.

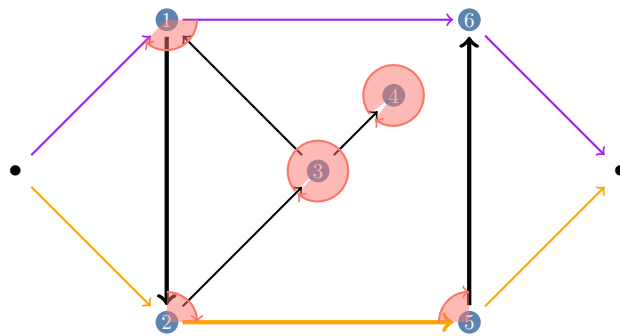


Figure 11.9 Finding a parallel path. The starting path is in purple, the target path in orange, and the nodes are numerated by the order in which they are travelled through by `find_parallel_path`. We display the angle range the procedure is called with at each node. The returned path if the thick one.

While the intuition behind this procedure only makes sense when the graph is planar, because it can only visit each node a finite number of times, one for each entrant edge at most, it always terminates.

11.4.2 find_step

This procedure takes a node along the first path, an angle for this node, and either fails or succeeds with a decomposition step between its node and the last node of the path. It starts by recursively calling itself to find a step as far as possible along the starting path. The reason for this is that if there is a step starting from the current node and a step starting from a node later in the starting path, and if the graph is planar, the latter step is either contained in the current one (see [figure 11.10](#)) or independent from the current one (see [figure 11.11](#)). In the first case, we must first apply the later step to obtain a decomposition in the smallest possible steps. In the second case, it does not matter which is applied first. Therefore, it is always correct to first apply the later step.

The recursive call is performed on the following node along the path by giving it as angle either the maximal angle at this node, if it is also a node of the right path, or the angle of the incoming edge. Finally, when we reach the last node of the path, we fail. Indeed, because we do not want steps with loops, there can be no step where the starting and ending nodes are the same.

There are now two cases :

- ◇ If the recursive call succeeds, we prefix the step with the edge from the current node to the next node before succeeding with the resulting step.
- ◇ If the recursive call fails, we call `find_parallel_path` on the current node with an angle range going from the angle of the outgoing edge to the angle taken as argument. If a parallel path is found, we return the step with no prefix, the left side being the part of the path going from the current node to the end node of the found parallel path, the right side being the computed parallel path, and the suffix being the suffix of the path starting from the end node of the parallel path. Otherwise we fail.

The algorithm is illustrated in [figure 11.12](#). Again, the intuition behind this algorithm only works on planar graphs, but if the graph is not planar, it would still terminate because it only visits each node along the starting path once, and on each of these, it only calls `find_parallel_path`, which always terminates.

11.4.3 Putting it all together

The algorithm then consists of taking the starting path, calling `find_step` on it, and repeating with the right side of the found step until we reach the target path. If, at any point, `find_step` fails, then the algorithm fails. On a planar graph, the target path will always be reached because every intermediate path closing it with the target path is always a valid step that can be found within the given angle bounds²⁹.

11.5 Equality term simplification

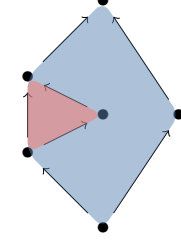


Figure 11.10 A later step contained in the previous one.

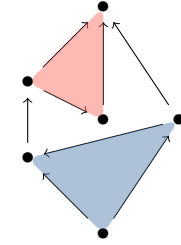


Figure 11.11 A later step independent from the previous one.

²⁹ The termination is not so obvious in the case of a non planar graph. The current implementation assumes it terminates even in this case, but we have yet to prove it. If it turns out not to be the case, we will add a check that the layout is planar, and refuse to execute otherwise.

```

 $p_1, p_2 := \text{Atomic } a$ 
| Refl  $m_1$ 
| Concat  $p_1 \ p_2$ 
| Inv  $p_1$ 
| Compose  $p_1 \ p_2$ 
| Assoc  $m_1 \ m_2 \ m_3$ 
| LeftId  $m_1$ 
| RightId  $m_1$ 
| FId  $F \ x$ 
| FComp  $F \ m_1 \ m_2$ 
| FCtx  $F \ p_1$ 

```

Figure 11.13 AST of equality terms. See [figure 10.2](#) in [section 10.2](#) for the complete AST.

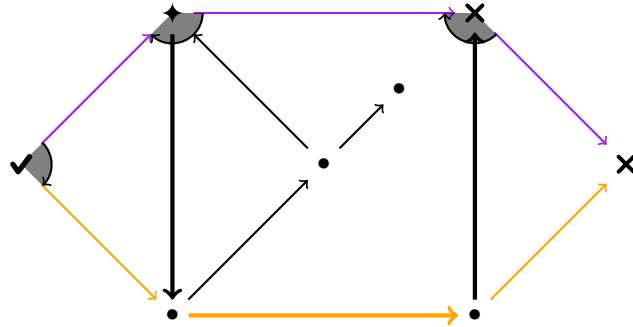


Figure 11.12 Finding a step. The starting path is purple, the target path is orange, and the parallel path found is thick. The nodes are annotated by ✗ if procedure failed on them, ♦ if a parallel path was found at that step and by ✓ if the recursive step succeeded.

We progress the proof state by generating an equality term that is used to refine the goal. As such, most procedures that manipulate the proof state generate equality terms. At first, we used an algebraic data type for equality, whose AST is displayed in figure 11.13. One limitation of this approach is that there are many ways to represent equivalent equalities. For example, $\text{Compose } p \text{ (Refl } m)$ is the same as p . However, this type of subtree is present everywhere in the generated AST to keep the parts of the code generating the equality terms simple. The problem is that having an excessively large AST results in longer typechecking in the proof assistant, and in the need of more data to be transmitted between the AST and the proof assistant.

We replaced this AST with a more abstract structure when manipulating equalities in the interface, which is reduced to the AST when sent to the proof assistant. This structure is described in the next section (section 11.6). Before resorting to this, we attempted a simpler approach. The idea was to keep the code generating ASTs as-is and apply a procedure that simplifies it before sending it to Coq.

In his famous paper on naturality [43], Mac Lane proved that in a monoidal category, all morphisms composed only of tensors, associators, and unitors are equal if a few equalities on them are true³⁰. Our intuition is that these morphisms resemble the equality structure we are manipulating. Indeed, the tensor operation looks like the `Compose` operation, the composition the `Concat` operation, and the associator and unitor look like the `Assoc`, `LeftId` and `RightId` constructors³¹.

Our hope was that we could take inspiration from the proof in the paper to create a simplification procedure, even if it is not optimal³². However, it does show that any two natural transformations can be made equal by filling them with commutative squares, with a proof looking more like the confluence proof. Therefore, while it could have been helpful if we wanted to construct an equality between equalities to show our simplification is correct, it does not help us find the target equality.

At that point, we implemented a naïve procedure that rewrites some bad patterns in a single pass on the AST. It performs the following operations :

- ◊ It associates concatenations to the right.
- ◊ It removes concatenations with reflexivity.

[43] Lane, *Natural associativity and commutativity*, 1963

30. The famous pentagon and triangular equalities.

31. This is no coincidence. Indeed, a monoidal category is a special case of 2-category, with a single object, while the equalities on morphisms of a category also give rise to a 2 categoretical structure on the underlying category. This remark forms the basis of the intuition behind the new structure described in the next section.

32. It turns out that finding a minimal equality term, even if we simply have the tensor and associator, is very non trivial. This is discussed in section 11.7.

- ◇ It simplifies the compositions with reflexivities³³.
- ◇ It pushes down the inverses, and simplifies the concatenation of mutually inverse morphisms and the inverse of reflexivity into reflexivity.

While doing so definitely resulted in a smaller proof term, it still struggles with the interaction of `Concat` and `Compose` and fails to simplify the proof term further in those cases. Furthermore, because it was meant to be temporary, the algorithm we implemented also failed to apply some of the previous operations³⁴.

Because we do not ask for *Uniqueness of Identity Proofs*, one may wonder if the transformations we do on the equality term could be problematic. Indeed, under a framework such as *Homotopy Type Theory* [17], one can observe a difference between equality terms; therefore such a procedure may change the semantics of the generated term. There are two answers to this remark, theoretical and pragmatic :

- ◇ From a theoretical point of view, for each transformation, there exists an equality between the transformed term and its source. This is true for most equality types in Martin-Löf type theories³⁵. Therefore, the transformation does not change the equality in a manner that can be observed internally.
- ◇ From a pragmatic point of view, the equality term that is being transformed has been generated by our algorithms, and we do not make much guarantee on its shape, so it does not make sense to consider it more *primitive* than the transformed one. One could even argue that because the simplification procedure tends to remove *bureaucratic* details in the proof terms and give more dominance to its ‘true’ semantic content, the transformed term might be a better reference.

11.6 Equality structure

11.6.1 Intuition

The complications with the attempts to simplify equality terms (see the previous section) led us to question the relevance of representing equality terms as an AST. Indeed, while the AST is a good representation for sharing the low-level structure of the equality term with the proof assistant, when we manipulate it in the interface, we are bogged down by having to insert normalizations everywhere. Therefore, we sought a more abstract combinatorial structure to represent equality terms in the interface, that we could then compile to an AST before sending it to the proof assistant³⁶.

The main intuition comes from the work on *homotopy type theory* [17]. The initial realization of this work is that every type behaves like a specific kind of category, ∞ -groupoids, where the morphisms are the equalities. Thus, if we look at the definition of a category in a proof assistant such as Coq, the equality between morphisms adds a higher structure on the category³⁷. If we squint a bit, it appears that an equality term is a 2-morphism in a 2-category³⁸. This is interesting because we know how to represent two morphisms in a 2-category in a way that focuses on the 2-morphisms using string diagrams.

Indeed, while string diagrams are mostly famous for their application in *monoidal categories*, this is only a special case of the fact that

33. To do so, the actual AST we use is slightly more complex than that in figure 11.13. Indeed, in addition to the `Compose` constructor, we have the `LAp` and `RAp` constructors that are equivalent to composing with reflexivity on one side or the other. In other words, those are the left and right applications of equality. In this way, we can rewrite a composition with reflexivity into an application, and an application with reflexivity into a reflexivity.

34. Because it has indeed been scraped, we believe it was a good idea not to spend too much time on it while looking for a better theoretical solution.

[17] Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*, 2013

35. This is because all the transformations use only the groupoid structure of the equality, which is present in most MLTT settings.

36. Of course, using two representations leads to some other problem. For example, when we need to unify two equality terms, because the proof assistant cannot use the higher-level structure, we must compile them to ASTs before unifying, and then somehow deduce the result on the higher structure from the result on the ASTs. To avoid this problem, we ensure that, whenever we unify equalities, one of the two is an atomic existential variable.

[17] Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*, 2013

37. We discuss the challenges posed to formalization by this remark in section 1.3.

38. To make this formal, we need to account for the fact that the composition of equalities is not strictly associative, but only associative up to propositional equality. Thus, a formalization needs at least another level, but this remark is also valid for this higher level. Therefore, we need to keep going, and we end up with a weak $(\infty, 1)$ category, or equivalently, a category enriched in ∞ groupoids.

[44] Selinger, *A survey of graphical languages for monoidal categories*, 2010

a monoidal category is a 2-category with a single object, and string diagrams work on any 2-category. However, string diagrams do not provide a data structure. Indeed, a big part of their appeal is that the manipulation of string diagrams is of topological nature [44].

11.6.2 The data structure

To design a data structure for string diagrams, we must first try to understand precisely what a string diagram is. As figure 11.14 illustrates, a string diagram can be defined as the dual, in the sense of planar graphs, of a commutative diagram. This makes objects into regions of spaces delimited by morphisms, which remain edges, and equality into nodes. However, it is often more legible to represent nodes as boxes, as shown in figure 11.15.

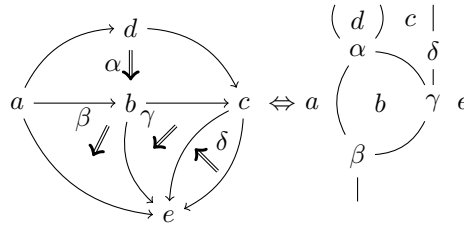


Figure 11.14 String diagram as the dual of the commutative diagram.

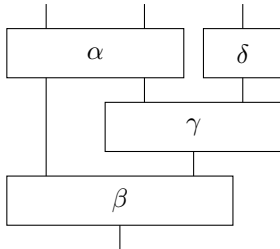


Figure 11.15 The same string diagram as in figure 11.14 displayed with boxes.

If we place each box on its own line, we start to see a representation as a vector of boxes, where each box has an input path, an output path, and a sequence of morphisms to its left and right. However, doing so seems wasteful because it fails to account for the fact that some box *commutes*, in the sense that they are parallel, the output of the first one and the input of the second one are disjoint.

Thus, we represent the equality as a vector of *slices*, where a slice is a triplet of an input path, an output path, and a vector of boxes with an offset of the boxes in the input and output paths. Boxes are treated as atomic equalities. They are made of either an equality term, of the inverse of an equality term, or of a reflexivity to handle cases where there is a composition on an edge that needs to be split into multiple edges³⁹.

Of course, to work with this structure, we need the ability to concatenate equalities, invert them and create reflexivities. Interestingly, instead of being between morphisms, those equalities are between paths in the graph. As such, the representation of equalities remains unchanged when the associativity of the input or output morphisms changes.

The reflexivity is simple, it is simply the equality with no slices. Inverting the equality means reversing the vector of slices, and in each slice exchanging the input and output morphisms, and inverting each block. The interesting case is the concatenation. We could simply concatenate the vectors of slices, which would be correct, but does not account for the fact that blocks in the second equality might commute with slices in the first equality.

Therefore, we must deal with the interaction between a slice and a block. We can easily check whether a block commutes with a slice by

39. There is also a fourth case to handle the image of an equality by a functor. There are many things that could be done to integrate better functors with this structure, which we have not had the time to investigate.

checking if a morphism on the input of the block is on the output of a block in the slice. If this is not the case, we need a procedure that inserts the block in the slice, and a procedure that commutes the two, by changing the input and output morphisms of the slice to reflect the application of the block on top of it. The intuition behind these operations is very graphical; therefore, please refer to [figure 11.16](#).

Once we have that, we can deduce a procedure to append a block to an equality. While the block commutes with a slice, compute the commutation. Once we have found a slice with which the block does not commute, we insert the block in the next slice. If it does not commute with any slice, we create a new slice at the bottom with the block inside (see [figure 11.17](#)). Now, we can append a slice to an equality, simply by applying the previous procedure to every block in the slice. Finally, we can concatenate two equalities by applying the previous procedure to all slices in the second equality.

11.6.3 Future work

This representation is useful because it allows us to delay the insertion of associativity and unitality until the last moment. Furthermore, it automatically handles the interactions between `Compose` and `Concat` in a high level manner. Therefore, we decided to use it in our implementation. However, we believe that this representation enables more opportunities. While we did not have the time to implement them, we describe them in this section.

◇ Functors

Now, we deal with functors simply by treating the constructor `FunctCtx` as an atomic equality, built from another equality. However, this operation has structural laws that are well behaved with regard to string diagrams [\[45\]](#). Therefore, it might be interesting to look into this in order to better integrate functors in our representation, in the hope of simplifying the terms further.

◇ Display

While we choose this structure for combinatoric reasons, it would be straightforward to display it to the user. Indeed, because the intuition is based on string diagrams, we could have a simple procedure to display the representation⁴⁰. If we decide to display them in the style of Hotz [\[46\]](#), using boxes for atomic equalities, we do not even need a layout because since we can simply reuse the structure to display the equality slice by slice.

◇ Inspection

Once we can display the equalities, it opens up new opportunities. The first comes from the fact that the user can now inspect the term generated without it being obscured by more *bureaucratic* parts, enabling them to gain insights into the proof⁴¹. This can be made easier by making the display interactive, for example, by

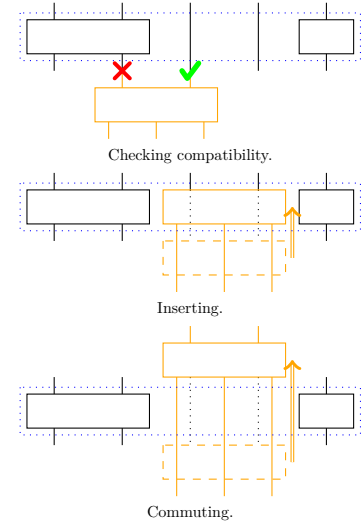


Figure 11.16 Operations between a slice and a block with a path of size 2 as input and 3 as output.

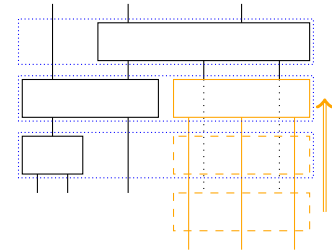


Figure 11.17 Inserting a block into an equality. Slices are the blue boxes and the inserted block is in orange.

[\[45\]](#) Mellies, *Functorial Boxes in String Diagrams*, 2006

⁴⁰. A significant advantage of this display is that it compensates for a weakness of the current display : loops. Indeed, if one side of an equality has loops, the currently implemented display does not provide much information. On the other hand, displaying the equality as a string diagram allows the user to precisely inspect the input and output paths of the equality.

[\[46\]](#) Hotz, *Eine Algebraisierung des Syntheseproblems von Schaltkreisen I*, 1965

41. When we get equality terms from Coq, we always treat them as atomic. However, if such a display were to be implemented, it would be interesting to parse the Coq proof terms and display them using this system. This would enable the interface to be used to analyze Coq terms in a graphical and abstract manner.

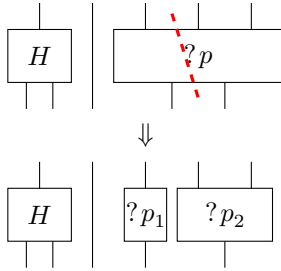


Figure 11.18 Splitting an existential along a common object on the source and target paths. The red stripe show the action the user is doing.

42. The distinction between path and morphism is important to understand. A morphism is a term that is understandable by the proof assistant. A path is a sequence of composable morphisms. We say that a morphism is compatible with a path when normalized the morphism is equal to the straight composition of morphisms along the path. A more formal definition is provided in [section 7.2](#), more precisely in [definition 6](#).

43. Once again, the odd case is the functor case, since it stores an equality it does not trivially have an input and output morphism. However, we can choose one arbitrarily when compiling the embedded equality, so it can fit in our description.

44. This requires the previous point, and means that the block, when converted to an AST, can be directly applied to the chosen morphism without needing normalization steps.

highlighting the relevant part in the main graph when some part of the equality is hovered.

Interaction

While we decided against making an interface that supports both commutative and string diagrams because their differences would increase the scope of things to implement too much, having some ways to progress the proof through the displayed equalities might be interesting. [Figure 11.18](#) displays how one such interaction could work.

11.6.4 Compiling to ASTs

While this structure makes it easier to build equality terms without needing to consider associativity, it cannot be understood by Coq or other proof assistants. Therefore, we must compile it to an AST whenever we need to send it to the underlying proof assistant. Although this may not be optimal, we proceed slice by slice and then glue the result together.

For each slice, we first compute an input and output morphism⁴². The exact procedure is not very relevant, the important points are the following :

- ◇ The computed input and output morphisms are compatible with the input and output paths.
- ◇ For each block, the input and output subpaths must be parenthesized together.
- ◇ Because a block is, in most cases, an atomic equality term⁴³, it has specific input and output morphisms. We require that the input and output subpaths are parenthesized such that the input and output morphisms of the block are valid subterms of the input and output morphisms of the equality⁴⁴.

We then create an AST by creating a reflexivity for each morphism that is not the input or output of a block, compile the blocks to ASTs, and compose all those equalities together. In the current implementation, we choose the input and output morphisms such that when there are multiple adjacent morphisms that are not involved in blocks, we simply create a single reflexivity on their composition, instead of composing multiple single reflexivities. Furthermore, we use the `RightApplication` and `LeftApplication` constructors to avoid composing with reflexivities.

Now that we have compiled each individual slice, we want to create a complete AST by concatenating them. However, this does not work trivially because the output morphism of a slice may not be that of the following slice. However, if we normalize these morphisms, we obtain the same morphism. Therefore, as shown in [figure 11.19](#), we insert normalization equalities between each slice to create the complete AST. Because the usual use case for the compilation is to generate an AST to unify it with an existential variable, which has specific input and output morphisms, our compilation also takes the expected input and output morphisms, and includes normalization steps at the beginning and the end of the generated AST in order to create an equality AST with the expected type.

The main advantage of this process is that the simplifications we would like to make regarding the interactions of composition, concatenation, and left and right applications are handled automatically by the structure we are using. However, there are still a few optimization opportunities that have not yet been fully explored or implemented.

First, our normalization step always fully normalizes each morphism and then concatenates these equalities, with one in reverse. This also means that even if the two morphisms are equal, we still create a huge equality term doing and then undoing the normalization. We could simply add a special case for when they are equal. However, we could also look to create a minimal renormalization equality between the two morphisms, which would be small if the morphisms are almost the same. This is explored in more detail in the next section.

Once we have this optimization, we could wonder if we can change the choices we make when choosing the input and output morphisms to minimize the renormalization steps. However, we believe that this would not be worth the gains. Indeed, we would need to optimize the choices across the entire equality to account for the connection between the input and output of a slice, which would probably be very costly in terms of computational power for a minimal reduction in term size.

Another optimization would be to create the AST across slices by cutting it up in independent blocks to handle the fact that sometimes there are parts of the equality that are independent across multiple slices (see figure 11.20 for such an example). Again, such an optimization would probably require a lot of non-trivial work to truly reduce the generated term size, and may not be worth the computational power.

11.7 Minimal associativity

When compiling the structure described in the previous section into an AST, the parts dominating the size of the generated AST are the normalizations. Therefore, one may wonder if we could optimize the generation of equalities between morphisms that are parenthesized differently, but otherwise equal. Of course, a complete solution should also handle identities and functors; however, as a first approximation we only focus on the application of associativity.

If we squint a bit, we can see that our AST is a binary tree, where each node corresponds to a composition, and the leaves to atomic morphisms⁴⁵. Looking at it through this lens, applying associativity to a subterm corresponds exactly to applying a tree rotation (see figure 11.21 for an illustration) at the root of a subtree. Therefore, finding the minimal number of associativities between the two morphisms corresponds to finding the rotation distance between the two binary trees.

The good news is that the problem of the rotation distance between trees has already been studied and was first defined in 1982 [47]. However, despite the progress made in terms of bounds [48] and equivalent formulations⁴⁶, the computational complexity of the rotation distance is still unknown, and the best currently known algorithms to solve this are exponential [49].

However, although it would be better to have an optimal solution, any improvements over the current situation would already be good. Approximation algorithms exist in the literature [50]; however they are very complicated. We designed a simple algorithm that may be bad

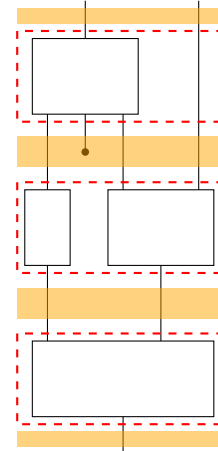


Figure 11.19 Compilation of an equality. The slices in red are compiled independently, and then normalisation steps are inserted in the orange regions.

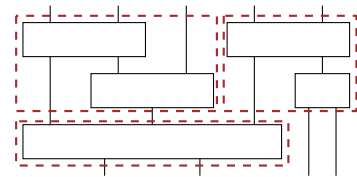


Figure 11.20 A decomposition in independent blocks of an equality term.

⁴⁵ Or identities, or functors, or ... For the sake of our discussion in the chapter, we consider any morphism that is not a composition atomic.

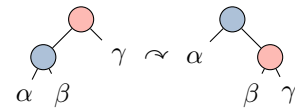


Figure 11.21 A tree rotation.

[47] Culik and Wood, *A note on some tree similarity measures*, 1982

[48] Sleator, Tarjan and Thurston, *Rotation distance, triangulations, and hyperbolic geometry*, 1988

[49] Li and Xia, *An $O(3.82^k)$ Time FPT Algorithm for Convex Flip Distance*, 2023

[50] Cleary and John, *A Linear-Time Approximation Algorithm for Rotation Distance*, 2010

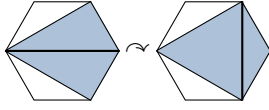


Figure 11.22 A flip between triangulations.

in the worst case, but handles much more gracefully good cases. The idea of the algorithm is based on the following remark : a composition, no matter what the associativity is, can always be identified by the morphisms on its left and right. In other words, in a tree, a node is always uniquely identified by the atomic morphisms to its left and right. Furthermore, rotations preserve this invariant. Thus, given two trees, we can look at the root node of the second tree. If it is the same as the root node of the first tree, we do nothing. Otherwise, we search for the corresponding node in the first tree, and apply rotations to raise this node to the root. We then repeat the process recursively on the left and right subtrees. This process is illustrated in [figure 11.23](#). In good cases, that is, if the trees are equal, or if only one application of associativity is necessary, it will indeed be optimal. However, we did not have the time to implement it.

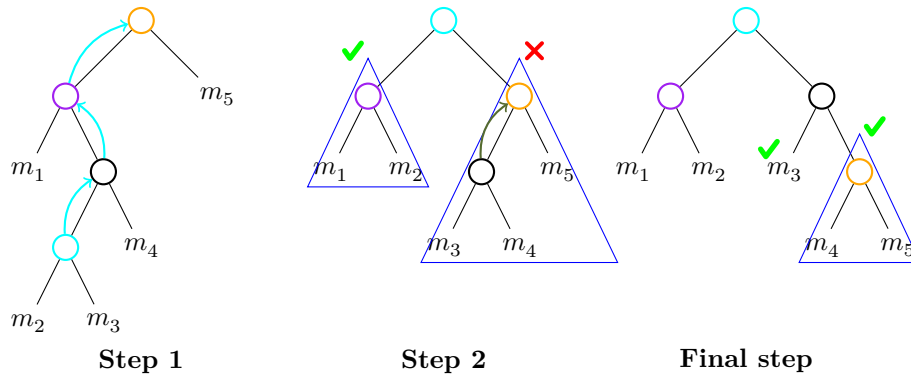


Figure 11.23 Finding a sequence of 4 rotations between two trees by successively rising the root node.

46. A particularly interesting reformulation is in terms of flip distance. Here, we create a regular polygon with $n + 1$ sides if the considered morphism is composed of n atomic morphisms. Then, parenthesizing the morphism is the same as triangulating the polygon, where each triangle indicates the morphisms that are grouped together. Applying associativity is the same as performing a flip, as illustrated in [figure 11.22](#). Thus, the rotation distance is then the same as the flip distance between the corresponding triangulations. Interpreting composition as triangulation is interesting because it is exactly the horn closing condition of a Kan complex. A Kan complex is a geometric interpretation of $(\infty, 1)$ categories, which is exactly the structure we are considering when looking at a category in a proof assistant where the equality gives a ∞ -groupoid structure to types.

[51] Cleary, *Restricted rotation distance between binary trees*, 2002

While we have not thought about how to integrate functors in such a procedure other than by treating them as atomic, some things need to be said about identities. The first remark is that, wherever they are and whichever way they are parenthesized, identities can always be inserted and removed. Therefore, an option would be to first remove all the identities, apply the previously described algorithm, and then insert the missing identities. Another way would be to look at where the identities are in the path, and only remove those that are not present in the target path. Then treat those that are as atomic morphisms while applying associativity and finally insert the missing identities. It is not obvious which of these two approaches is better, because while the second one reduces the need to remove and insert identities, more associativities might be applied. Maybe a more precise criterion could be designed to select identities that can be efficiently kept.

Until now, we have treated all applications of associativities as the same. However, the deeper the corresponding rotation is in the tree, the larger the term will be because it needs all the composition congruence applications. Therefore, maybe we can restrict ourselves to rotations at the top level. The good point is that this approach results in a decidable distance [51]⁴⁷. However, this also means that if a single rotation would be needed deep in the term, it would instead apply many associativities to raise it first, so it is not obvious if this is a

real gain. If it is cheap enough to compute, it might be interesting to implement both this approach and the one described above, run both, and choose the smaller result each time.

47. Of course, allowing only rotations at the root is not sufficient, we also allow rotations on the right child of the root node.

11.8 Linear unification

As described in [chapter 10](#), the first iteration of the interface uses ASTs for terms and implements its own unification. Efficiently implementing unification is not trivial; therefore, we used an algorithm devised by Paterson and Wegman in 1978 [52] that allows unifying a pair of terms in linear time. We will describe some of the challenges and key points of the implementation of this algorithm, but will not spend too much time explaining the algorithm itself, so please refer to the paper for details about the algorithm.

[52] Paterson and Wegman, *Linear unification*, 1978

Instead of working on ASTs, this algorithm operates on an acyclic graph of terms, to allow for the sharing of subterms. It even assumes that the sharing is maximal. The unification then works on a graph that contains all subterms of the two terms. It remembers what terms should be made equal, and cleverly propagates them to go through the graph at most twice. Such a graph is shown in [figure 11.24](#). Outside of the implementation of the unification algorithm, the main challenge was to efficiently deduplicate the terms of the AST.

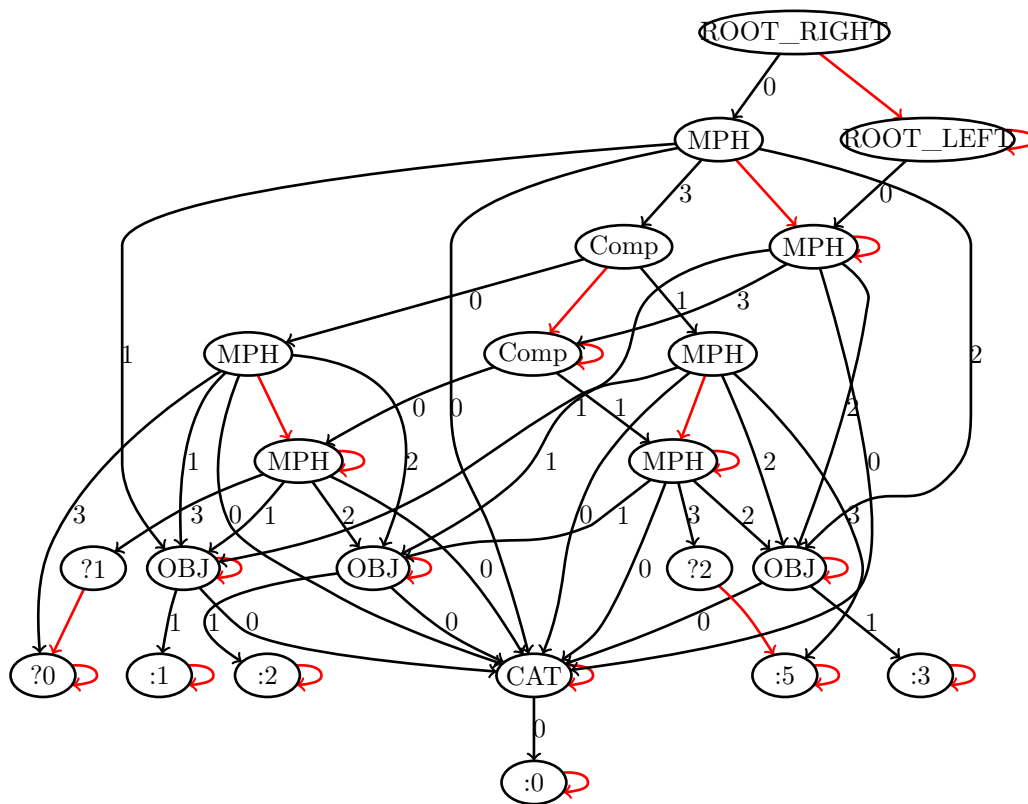


Figure 11.24 A deduplicated graph of an AST. The black edges represent subterms, and red edges the subterms made equal.

`<22>hashconsing crate`

[53] Conchon and Filliâtre, *Type-Safe Modular Hash-Consing*, 2006

48. The language described in [section 6.2](#) can be considered as the inheritor of these considerations.

[54] Bahiense, Manić, Piva et al., *The maximum common edge subgraph problem: A polyhedral investigation*, 2012

49. For example, both $?x \approx f$ and $?x \approx g$ can be solved independently, but jointly it fails due to the fact that $f \neq g$.

50. We performed this migration when migrating to the second data model described in [section 10.3](#).

51. Obviously, the number of matchings between two graphs can be very high, so enumerating them all is really bad from a complexity point of view. Our hopes was that the unification constraints would prune a lot of the matching, and hopefully prune them early.

52. We believe that it better captures the idea of a maximally applied lemma than simply the number of nodes or edges matched.

The usual way to achieve this is to use *hash-consing*. Instead of implementing our own, we reused the rust library *hashconsing*`<22>`, which is based on a paper by Filliâtre and Conchon [53]. Using this library, we replaced the AST structure with an acyclic graph with maximal sharing. Because the library also exposes a unique integer for each hash-consed term, it makes it easy to associate data with terms, which we used in the structure for unification.

11.9 Automatic lemma application

At the beginning of the development of our plugin, we aimed to create a language to manipulate diagrams⁴⁸. We already had a blurry view of how we wanted to use the lemmas. However, constructing partial matchings in a purely symbolic way seemed too complicated to be useful, so we were thinking of creating a procedure that could choose such a matching automatically.

Because there are many such matchings, we wanted one that was maximal in some sense. We examined the *Maximal Common Edge Subgraph* problem, which given two graphs intends to find a common subgraph with a maximal number of edges. The bad news is that this problem is NP-complete, and the best known algorithm uses linear integer programming to solve it [54]. Furthermore, our situation is more complicated, because our graph is annotated with terms that must be unified. Even worse, two pairs of terms might be unifiable independently but not jointly⁴⁹. Therefore, we require an algorithm that can handle these problems.

The first idea we considered was to encode the unification constraints in integer linear programming constraints and then adapt the previous encoding of the MCES problem to include those constraints. Thus, the solver solves the unification constraints and the graph matching simultaneously. However, this approach only works as long as we directly manipulate an AST for the proof assistant terms, and at that point, we already had the ambition of migrating to reuse the proof assistant unification⁵⁰. Therefore, we sought another solution to this problem.

The second idea, which we implemented, was a simple backtracking algorithm that tries to match nodes and their outgoing edges together until the unification fails. To manage the time complexity, it was designed in such a way that it would progress until it finds a matching, and then return it. The process can then be restarted from its stopping point. Thus, all the matchings can be enumerated without having to store them all, and the process can be stopped if it takes too much time⁵¹. Furthermore, it enables the possibility of selecting the *best* matching after the fact, using any criterion. One particular criterion that we think might be interesting is the number of existential variables instantiated⁵².

With the implementation of the interactive graphical matching, we found we do not really need this procedure. The effort to maintain it across the multiple refactors has not proven worth the effort, so it was dropped in the current version of the code.

12 Coq plugin architecture

This chapter is dedicated to the implementation details of the Coq plugin. It discusses all of the interesting choices that were made when implementing our interface. This should be relevant for anyone implementing similar software. See the next chapter for the implementation details of the interface.

12.1 Interface communication

As described in [section 10.1](#), our development is split into two parts that communicate through a full-duplex connection¹. When the tactic is started from Coq, the plugin code is executed. It then must start and connect to the interface. In our organization, the plugin is the server and the interface is the client.

The first step in starting the interface is to locate its executable. To do so, the plugin first checks whether there is a `COMDIAG_ENGINE` environment variables set to the path to the plugin². If it is not present, it looks for the `commutative-diagram-engine` binary in the `PATH` directories³.

When the binary is found, it is started, and its standard input, output, and error output are bound to channels⁴ used to communicate with it. The standard input and output are then used in each direction for communication with the interface, as described in [section 10.4](#). Any content sent to the error output is displayed to the user using the Coq log system. In this manner, error messages from the interface are propagated to the user. This is especially useful in the case of an interface crash, which often displays a stack trace and debugging information when crashing, which is then displayed and can be used to debug it. In a normal, run there should not be any message displayed in this manner.

This approach should be contrasted with Actema⁵. First, in Actema's protocol, the interface acts as the server and the plugin as the client. Owing to this design, the interface must first be opened and kept open during the proof, and whenever Actema's tactic is used, the plugin code connects to the interface, which listens to a local port. On a new connection, the interface opens a new tab for the new proof.

The approach used by Actema has the advantage of enabling an interaction in which the interface is opened before Coq. While using the usual workflow for working on a proof in Coq, both approaches are equally viable. However, when working on supporting a web version ([section 13.4](#)), we needed a method for the entire interaction to be initiated by the interface. To support this, we added a new channel for communication : if there is an environment variable `COMDIAG_SOCKET` containing the path to a Unix socket⁶ when the plugin is run, it connects to this socket instead of starting the interface. This enables the interface to be started, bind a Unix socket, and wait for the Coq plugin to connect to this socket and answer on it.

12.2 Term indexing

Because terms are opaque objects for the interface, we need a method to name them so that the interface and plugin can communicate about

1. Full-duplex means that both the Coq plugin and the interface can send messages to the other at the same time. The current implementation, however, only uses the ability to communicates in both directions, only one message will be sent concurrently. As such, a *half-duplex* connection would be sufficient.

2. `server.ml`

3. The `PATH` environment variable is a colon separated list of directories used to search for executables on Unix-like systems.

4. Channels are the abstractions used in OCaml for `input` and `output` streams of data.

5. Actema : une interface graphique et gestuelle pour preuves formelles

5. Actema is a Coq plugin that enable graphical interaction for predicate reasoning.

6. A Unix socket is a *socket*, so a primitive exposed by the operating system supporting full-duplex communication that, instead of being bound to a *port*, is bound to a path in the filesystem.

7. The reason we chose to use an array instead of a list is that during the interaction, terms will be queried more often than created, so we chose the type that is optimised for indexing instead of appending. Furthermore, we only append at the end, which can be implemented in amortized constant time for arrays.

8. `hyps.ml`

9. Sorting the array means having an order on the terms. Coq does not provide any, because having an order compatible with the usual operation on term is complicated. However, we only need any order on the terms to perform an efficient lookup, so we could have created one.

10. At many points, the interfaces checks that when composing two morphisms they are indeed composable. These checks are used to detect bugs in the implementation, as the structure of the graph should guarantee that only composable morphisms are ever composed. To verify this, it is necessary to compare the source and target objects for equality modulo $\beta\delta$, thus the notion of representative.

11. Shifting is the operation of increasing the value of all free variables when using a De Bruijn representation of terms. It is the mechanical implementation of weakening because it transforms the term into another, which refers to an extended context.

12. Forgetting to do so leads to some interesting bugs, as the interface considered terms equal, while they had completely different meanings and types, breaking all the assumptions that allows it to work.

them. As described in [section 10.4](#), the names we decided to use are integers. Therefore, we need a way to provide each term that we encounter as a unique integer.

To achieve this, the plugin maintains an array⁷ of all the terms it encountered⁸. When encountering a new term, it first looks into this array to determine if the term is already present. If not, the term is simply pushed at the end of the array. The integer associated with each term is its index in the array.

Because adding a new term requires first looking into the array to check if it is not already present, it would be more efficient to keep the array sorted⁹. However, we want to keep the indices constant because the interface may refer to them, thus new insertions cannot be made without changing the indices or breaking the ordering. Another relevant question is which equality is used when comparing the terms. If we use the equality of the term representation in Coq, we can store many duplicated versions of what should be semantically the same term. On the other hand, comparing modulo $\beta\delta$ is more efficient in terms of storage but makes each comparison more expensive. The current implementation uses comparison modulo $\beta\delta$.

The protocol includes the notion of a representative. As the proof progresses and existential variables are instantiated, terms that were previously different, even modulo $\beta\delta$, may become equal, and the interface needs to know that. Therefore, when the interface queries the representative of a term, the plugin performs a linear search in the array for another term equal modulo reduction to it under the new context, and returns the first one it finds. This approach allows the plugin to compute a representative of the equivalence class modulo $\beta\delta$ of a term, which is then used by the interface when performing typechecking checks¹⁰.

Each lemma also has its own array of terms used to identify terms in its context. A separate array is used because lemma terms are to be understood in the context of all its quantified variables, so `var 0`, while equal, could have very different meanings for different lemmas. Lemma terms can then be introduced into the global context by substituting all variables with existential variables or terms that are themselves in the global context. One particular complexity of dealing with the context of a lemma is that it needs to store the conclusion of the lemma itself, but also that all types of quantified objects need to be stored because we need them to construct the graph of the lemma. However, each of these types lives in a specific context, which is that of all the previously quantified variables, but not the following. To ensure that all terms live in the same context, we need to consider *shifting*¹¹ terms before adding them to the array¹².

12.3 Term analysis

In the previous section, we described how the plugin attributes numbers to Coq terms such that the interface can manipulate them abstractly. However, for the interface to do anything interesting, it needs to access at least some information on the structure of the terms. In the protocol, this is implemented through the *features* system, as described in [section 10.3.1](#). When a term is queried, its structure must be analyzed by the plugin.

Coq exposes an interface for deconstructing a term into its syntax in the calculus of constructions. However, the features we are interested in are much higher level : we care about whether a morphism is an identity or a composition, not whether it is an application, an abstraction, or any other term former. To reconnect the two, our code¹³ deconstructs the term as needed, depending on the specific feature we are interested to recognize patterns in its syntax tree. For example, when trying to look for a composition, it checks if the term is an application of the composition projection of the category type¹⁴ applied to other terms.

However, this approach has several limitations. First, the patterns that we look for are hardcoded in the plugin. If the user wants to use our plugin with another library, they will have to rewrite the query functionality. This is mitigated by the fact that all the term names we use are isolated in a separate file¹⁵; as long as the shape of the patterns themselves does not change, it should not be too complicated to adapt the plugin to another library. However, changing the code of the plugin and recompiling it does not lead to a good user experience. At least the names of the constants used in our analysis should be configurable at runtime.

However, even if the constants can be easily changed, the current approach supports only one set of constants simultaneously. This makes sense if we consider that a specific development will only use one formalization of category theory. However, our interface can be used with anything that remotely looks like categories, such as Coq types and functions¹⁶. Thus, the user could want to change the set of patterns used for each invocation of the tactic, depending on the kind of proof they are doing.

Even if the constants used in our parsing were easily changed, the rigidity of the patterns would be problematic. While most formalisations are similar, some are using very sophisticated approaches to deal with the complex hierarchy of structures that one can add one categories for example²³. A more fundamental problem appears when dealing with specific categories : the constructors may compute. For example, morphisms in the product category reduce to products of morphisms. Thus, the type of morphisms is not of the form `morphism ....` Dealing with these cases involves adding new patterns for common reductions¹⁷. However, if the user defines their own specific categories, there would not be any patterns to recognize them after reduction, severely reducing the usefulness of our interface. Therefore, we want a way to make the patterns extensible at runtime, if possible, with a better syntax to express them.

We also want the building of terms to be extensible, so any system we use should also handle this. However, building terms is much less complicated, since we can always build terms using generic builders and let Coq reduce them to the specific cases : when building the type of morphisms in the product categories, we do not need to know it reduces to the product of morphisms.

12.4 ELPI

This section presents plans to address the drawbacks of our term analysis method, which is described in the previous section. We conclude

13. [query.ml](#)

14. In Coq, the fields of a record type are called primitive projections and are handled in a surprising manner in the syntax of the language. Each projection has an associated constant that, when applied to enough parameters, changes nature in the syntax tree, becoming a projection.

Remark 1. *Our analysis of a term structure is guided by the type of the term, and we often need information that is only directly available on the type, particularly when dealing with primitive projections. However, this often leads us to call the pretyper, and often on the same terms, to check whether it is a category, a morphism, an object... To avoid redoing the work too many times, we use the array mentioned in the previous section to cache the type and data about the type of the terms we analyzed.*

15. [env.ml](#)

16. We have not fully explored whether using our interface for Coq types and functions is possible, and we expect some difficulties due to universe handling, but there is no fundamental reason to believe that it should not be possible. This means that our interface may have uses, even outside of category theory developments.

²³ [Hierarchy Builder](#)

17. However there is a problem in general that when encountering a product of morphism, there is no way to know whether the user intended it as a morphism in the product category or not : the intention is lost through the reduction. A more powerful pattern system might recover a lot, but it is fundamentally going to be guessing ways to fold the concrete terms it encounters.

⟨24⟩ Soufflé

[55] Guidi, Sacerdoti Coen and Tassi, *Implementing Type Theory in Higher Order Constraint Logic Programming*, 2017

18. A Higher-Order Abstract Syntax or HOAS is an abstract syntax that deals with binders by reusing bindings from the ambient language. Instead of using names or De Bruijn indices to deal with variables and bindings, bindings are abstracted in the source language, and there is no need to include a variable case. These syntaxes are convenient for implementing interpreters because the substitution is simply the evaluation of the language. However, they complicate syntax analysis because most languages do not support pattern matching under binders.

⟨25⟩ Coq-Elpi : Coq plugin embedding ELPI

that to make the term analysis truly useful, it needs to be fully extensible, and if possible, using a language adapted to find patterns in abstract syntax trees. A language that has emerged as a concise and efficient language to write syntax analyses is *Datalog*. For example, there has been considerable effort to implement a Datalog engine fast enough to prototype program analyzers on huge codebases through the development of *Soufflé*⟨24⟩.

While reimplementing a Datalog engine is beyond the scope of our project, there is already an OCaml library implementing a λ Prolog in OCaml : ELPI [55]. λ Prolog is a variant of Prolog that includes mechanisms for reasoning under binders, making it possible to write straightforward analyses of higher-order abstract syntaxes¹⁸. Furthermore, because ELPI was developed to provide a more convenient language for implementing pretyping in Coq, there is a development that provides a definition of Coq terms as HOAS in ELPI, along with functionalities to import and export terms from Coq in ELPI through this interface⟨25⟩.

We intend to adapt our plugin so that the analysis and building of terms are done by ELPI, while keeping the rest of the plugin as is. Because ELPI is interpreted, it means that the user can define its own rules by providing Prolog sources that define or add rules for the building and analysis predicates. Finding the right types for the predicate and the right interface would probably require some work, but ultimately we believe it could make our plugin much more extensible and powerful. However, while ELPI itself is meant to be used from OCaml, Coq-ELPI is meant to be used directly from Coq and does not export its definition as an OCaml API. There are three possible methods to circumvent this problem :

- ◊ Rewrite the entire plugin in ELPI. However, communication with the interface through channels require some low level APIs that are not exposed to ELPI. Furthermore, the caching of terms in an array is fundamentally imperative and reimplementing it in ELPI would probably lower the performance. Rewriting the entirety of the plugin in ELPI is also a time investment that should not be neglected.
- ◊ Extend Coq-ELPI to expose an OCaml API. While this is the simplest solution for us, the maintainer of Coq-ELPI, is not interested in maintaining an OCaml API.
- ◊ We maintain a local fork of Coq-ELPI that adds the OCaml API. This is the approach we will likely pursue.

12.5 Context handling

The plugin requires some way to store and restore the Coq context. Because the only way the context is modified through the interaction is by unification, we do not need to care for the entire Coq proof context, but only for the underlying `evvar_map`. The `evvar_map`, as the name indicates, stores information about all the *existential variables* in the context, whether they have been refined with a value and the value if they have¹⁹.

Our first instinct was simply to store the `evvar_map` when saving the context, and to restore it when a specific context needs to be restored. However, throughout its evolution Coq started to store more

19. For this reason, most functions inspecting terms in Coq also take an `evvar_map` as argument : existential variables are substituted lazily using this map, to avoid having to do the substitution on all the terms in the context every time a variable is refined.

information in the `eval_map`. More specifically, it is also used to store universe constraints for universe polymorphism²⁰. Simply rolling back the `eval_map` causes problems with the state of the universe checker, which we never really manage to identify.

To circumvent this, we simply remember which variables were present, and which were instantiated when storing the state. When rolling back, we use the working `eval_map` and remove and deinstantiate the variables as needed. We are not really happy with this approach, as it may allow state to be kept that we wanted to discard, but it does solve the problems we had with the universe solver.

²⁰. Universe polymorphism is a feature of Coq that hides universes from users. For example, when a user writes `T : Type`, `Type` refers to a specific universe level. Coq creates a universe variables and maintains a set of constraints between universe variables as terms are elaborated, that are solved at the end. This is really until it fails, at which point the user does not have any way to inspect the state of the solver.

13 Interface architecture

This chapter is dedicated to the implementation details of the interface. The interface is interesting because, despite being only meant to allow graphical manipulation of diagrams and being layered on a proof assistant, it needs to reimplement some features of proof assistants. More specifically, while the context and existential variable handling are delegated to the underlying proof assistant, it still needs to support its own high level tactic language to store the interaction¹. Furthermore, because it stores its own state in addition to the Coq proof state, rollbacks of actions must be supported.

Regarding more concrete aspects, the interface is implemented in *Rust*⁽²⁶⁾ and *egui*⁽²⁷⁾ is used to draw the interface.

13.1 Tactic language

We use a tactic language that we designed ourselves. Because it has not been a priority, and as we explain in [section 13.3.2](#), long term plans are to replace it with a real programming language, it is pretty barebone. It is a line-based language, where each line is either an action, an empty line, or a comment². Empty lines and comments are ignored. An action starts with its name, and then takes a space separated list of arguments. Currently, there is no unified syntax or type system for the arguments of the actions, as each action has a specific parser for its arguments³.

The language refers to objects in the context by their name. The names are resolved at runtime. Names should not be confused with *labels*. The label of an object is the textual representation of its value, and is generated by the underlying proof assistant. A name has a more restricted range of accepted values, as it cannot include spaces, for example, and it must be unique for each object. Names are automatically attributed by the interface in a deterministic manner⁴, as a change in an object name can result in a failure to execute a stored script. However, while deterministic, the names can be changed by small variations of the context, so a more robust way to refer to objects would be better⁵.

13.2 Supporting rollbacks

Supporting undoing is important, not just for convenience. Any action that refines existential variables is irreversible, so without the ability to undo them it would be possible to get stuck and have to restart the interface from scratch to conclude a proof. In order to be able to store interactions as a script, everything the interface does is through *actions*⁶. An action is a high-level description of a modification of the proof state. Each action has a textual equivalent in the tactic language. Thus, tactic scripts are precisely a list of actions⁷.

Therefore, whenever the user does something, the script is appended at the current execution point with the representation of the action, and the action itself is added to a list of executed actions, stored in the state⁸ of the interface and the action is executed. The actions themselves are too high level, and they do not store enough information to be

1. We discuss the choice of using a tactic language instead of a binary format to store the interaction in more detail in [section 6.2](#). As it is now, it is not meant to be written by hand, but we will discuss in [section 13.3.2](#) plans to make it powerful enough to share scripts across proofs and maybe even proof assistants.

(26) The Rust programming language

(27) Egui

2. [vm/parser.rs](#)

3. A consequence of this approach is that there is no typing step. Each error is either a parsing error or a runtime error. Furthermore, while we report errors to the user, we have yet to make them user friendly. So instead of getting an error like *The action solve expected an integer as first argument but got a name instead*, the errors are more like *Expected a digit, got 'a' at line...* Furthermore, since the parsing stops as soon as it fails to parse a character, the error range always includes all the rest of the script from the point of the error.

4. [vm/namer.rs](#)

5. The tactic language for Coq, LTac, also suffers from this problem. This is usually mitigated by manually naming all introduced objects and by using tactics that find hypotheses based on their semantic content instead of their name. Manually naming things is not really an option for us, as it would make the graphical interaction much slower; therefore, a more semantic way to identify objects should be developed.

6. [vm/ast.rs](#)

7. Actually, some actions can be represented in multiple manners, but going from actions to tactics and back to actions is always guaranteed to be the identity.

8. [vm/vm.rs](#)

reversible. Furthermore, they are often contextual, with non-trivial logic used to execute them.

13.2.1 Instructions

The solution we found to support rollbacks is to have an intermediate notion of action, which we call *instructions*⁹. Instructions describe low level operations on the state. Furthermore, they store sufficient information to be executed and reverted. Abstractly speaking, they are a way to store a diff of the state. Now, whenever the user does something, the interface first interprets it as a list of actions, which are stored to the script and *compiled*¹⁰ to instructions. The instructions are stored and executed¹¹. The interface remembers which action maps to which instructions and to which part of the script.

Now, whenever the user edits the tactic script, causing a rollback, or undoes, the interface has enough information to find the actions that need to be reverted. To revert an action, its list of instructions is read in reverse and each one is reverted. Finally, both the action and the reverted instructions are removed from state of the interface. However, they are not removed from the script. Thus, if the user wants to *redo* an action, the script is simply read forward and executed again¹². Any action present in the script that was not executed when finishing is commented before saving the script.

One aspect we have yet to mention is how the instructions deal with the underlying proof assistant state. Indeed, because we abstract away from the proof assistant, we have no way of storing a diff of its state in the interface. This is where the API to deal with state described in [section 10.4.3](#) is used. When compiling an action, whenever the interface uses the `unify` protocol call, it also emits a `DirtyState` instruction. The execution of this instruction results in the interface using `save_state` to store the state before the unification, and uses `restore_state` when the action is rolled back¹³.

To avoid querying the underlying proof assistant several times, the interface has a *cache* system^{14,15}, which stores information about objects. However, whenever two objects are unified, the store needs to be marked as stale, as the information may have changed, especially the label of the objects. Thus, the `DirtyState` instruction is also used by the interface to know determine the cache needs to be marked as stale. However, the previous cache is not discarded, and the structure of the cache is stratified such that new information can be removed efficiently when `DirtyState` is reverted¹⁶.

13.2.2 Interactive actions

A limitation of this system is that it assumes that every graphical operation maps onto a single action. It is not complicated to add actions if an operation requires multiple actions¹⁷. However what if we construct a single action using multiple graphical operations. The initial example for this approach was the application of lemmas, described in [section 8.3](#) and illustrated in [section 6.1](#).

Applying a lemma should be a single action. However, we want the user to be able to construct the matching interactively. More importantly, every time the user adds something to the partial matching,

9. [vm/asm.rs](#)

10. [vm/compiler.rs](#)

11. [vm/interpreter.rs](#)

12. A limitation of this approach is that if a graphical interaction leads to multiple actions being created, undoing and redoing will only act on one action at a time. However, because actions represent high-level operations, they are designed such that they map nicely to graphical operations.

13. The exact manner in which `save_state` and `restore_state` work depends on the underlying proof assistant and the plugin implementing the API. See [section 12.5](#) for a description on how we do it for Coq.

14. [data/store.rs](#)

15. [vm/store.rs](#)

16. The entire interaction of the instruction system with the handling of the state of the proof assistant has been developed organically, and requires careful synchronization of many kinds of states. The addition of the interactive actions described in [section 13.2.2](#) makes it even worse. Thus, this system has been a recurrent source of bugs, and it should be simplified and refactored.

17. Alternatively, we could have a notion of grouping actions. However we have not yet felt the need for this.

we want to try to unify it to display the results of the unification or fail early, without canceling the entire construction of the partial matching. Finally, the entire interaction can be either canceled, or the lemma can be applied.

To support this type of workflow, we introduced the notion of *interactive action*¹⁸. An interactive action is an additional state that is stored in the interface. An interactive action can execute instructions that are added to the list of instructions, but do not correspond to any stored action. Then, when the interaction is completed, the interactive action must provide an action that *if compiled, would result in the same state change*. This action is then added to the action list, and all the instructions added during the lifetime of the interactive action are associated with it.

18. `ui/vm.rs`

Interactive actions are canceled, and all their instructions reverted, whenever another interactive action is initiated or when any action is executed. In this way, there can be at most one interactive action at any time, and the instructions for the interactive action, if any, are always the final instructions to have been executed.

This system enables actions to be constructed interactively. The additional stored state specific to the action is also used by the user interface to show information about the current interactive actions. Things such as the window for the current lemma applied, or the object we want to match, are all stored in this state. Thus, interactive actions are also used for any action that needs to display additional information in the interface during its construction, even if it results in a single action at the end. For example, insertion, which opens a pop-up for the term to be inserted, is implemented as an interactive action¹⁹.

19. `ui/vm/insert.rs`

13.3 Extensibility

One major limitation of our interface is that it only knows the axioms of basic category theory. However, commutative diagrams usually have many additional annotations for displaying structures. Monomorphisms are displayed with hook arrows $\hookrightarrow$, epimorphisms with double ended arrows $\rightrightarrows$, pullbacks with $\lrcorner$...

Not only are those annotations useful for visualizing the properties of the manipulated objects, but it is also common to reason directly with them. For example, pullback squares are composable²⁰. Alternatively, through their interaction, such as how the pullback of a monomorphism is a monomorphism or how the fact that a morphism is a monomorphism can give us new equalities of morphisms. Furthermore, these properties are also proven through diagrammatic reasoning, and not just equalities of morphisms.

20. Given two pullback squares that share an opposite side, for example the left side of one being the right side of the other, the square obtained by pasting them along their common side is also a pullback.

Therefore, it would be interesting to support many of those properties in our interface to enable graphical reasoning not just on the equality of morphisms but on any of those properties. However, making an inventory of all the relevant properties, along with how they should be displayed, to support all of them would be too ambitious. Furthermore, giving the user some control over the notations is useful because depending on the context and field they may want to use different notations for the same objects.

This points to the need to have an extension mechanism that allows the user to define new structures, choose how they are displayed, and even provide some high level operations that may be performed on them. Extensibility in this way would require three developments :

- ◊ A data model that can describe a category with arbitrary abstract structures. The current model uses a graph with a set of parallel path to represent equalities. It is unclear how such a representation can be extended. We discuss our plans to do so in [section 13.3.1](#), using a theory for representable categories called *sketches*.
- ◊ A way for the proof assistant plugins to describe the structures they support. This should not require any theoretical work, only an update to the protocol used to communicate between the interface and plugins, which would be necessary anyway to support the new data model. Considering the Coq plugin we implemented, we want it to be extensible to easily support new structures. However, we believe that the plan to use ELPI, described in [section 12.4](#), would already make it sufficiently extensible.
- ◊ A method for graph drawing and interaction to be scripted. This would require the interface to support a more powerful scripting language, and expose an API that allows the language to extend the graph drawing and manipulation. We describe our plans to do this in [section 13.3.2](#).

13.3.1 Sketches

Our objective is to find a way to describe categories with arbitrary additional structures in a finite and computationally tractable manner. We also need to specify operations on these structures. We believe that the theory of *generalized sketches* to be a good approach.

Let us start with the original *sketches* [56][57]. They provide a graphical notion of *theory*.

Definition 1. Given a directed graph²¹, a diagram is a subgraph.

Definition 2. Given a directed graph and a diagram, a cone on this diagram is an object of the graph along with a collection of edges from the object to each object of the diagram. A cocone on the diagram is an object along with a collection of edges from each object of the diagram to this object.

In general a cone or cocone in a directed graph are the information of a diagram and respectively a cone or a cocone on this diagram.

Definition 3. A sketch is a quadruple composed of :

- ◊ A directed graph.
- ◊ A set of diagrams on this directed graph.
- ◊ A set of cones on this directed graph.
- ◊ A set of cocones on this directed graph.

The intuition behind the notion of sketches is to describe a basic shape that must be present in the category through the directed graph, with the additional conditions that the given diagram should commute, the cones should be limit cones, and the cocones should be colimit cocones. This is made formal by the notion of *model* of a sketch.

[56] Ehresmann, *Esquisses et types de structures algébriques*, 1968

[57] Bastiani and Ehresmann, *Categories of sketched structures*, 1972

21. To be understood as a quiver, as defined in [definition 1](#).

Definition 4. A model of a sketch is a category C with a graph morphism from the graph of the sketch to the underlying graph of the category, such that the image of each diagram is a commutative diagram, the image of each cone is a limit cone and the image of each limit cocone is a limit cocone. C is usually taken to be $\mathbf{Set}$.

In a way, sketches are already a generalization of the structure we manipulate in our interface. Indeed, we have the underlying graph, and a set of commuting diagrams. However, instead of representing the commuting diagrams in their generality, we only support them if they have a *globe* shape²², since we interpret them as equalities. However, any commuting diagram can be decomposed into the set of all its pairs of parallel paths.

Furthermore, sketches are manipulable as data structures as long as the graphs and sets considered are finite. They already provide a more general description of categories, because they can describe limit cones and colimit cocones. However, this is still too restrictive. Furthermore, while pullbacks and pushouts are limits and colimits, we want to identify them because they have their own graphical representations.

Another problem with this description is that it does not provide a way to describe operations on sketches in an abstract manner.

Makkai describes a generalization of the notion of sketches that resolves our problems [58]. His approach has even been used as a basis for concrete programming language modeling [59]; therefore it should be computationally tractable.

Sketches are generalized in two ways. First, instead of being based on graphs, they are abstractly defined in a category G . Second, they can have an abstract notion of features with associated shapes, whose meaning is determined later. Furthermore, it comes with a notion of *sketch axioms* that can be used as a description of actions.

Definition 5. Let G be a category and $(K_i)_{i \in I}$ an indexed family of objects of G . A generalized sketch S in G of signature K is an object $|S|$ of G along, for each $i \in I$, a set $K_i[S]$ of morphisms from K_i to S .

G should be interpreted as a category of shapes, for example the category of graphs. Each i is the name of a kind of feature in the sketch, and K_i is the shape of this feature. Then, a sketch is a shape along with for each i a set of instances of the shape K_i in the sketch. In non-generalized sketches, the shapes were commutative diagrams, limit cones, and colimit cocones. Now we describe the shape data without specifying anything about its semantics.

Definition 6. Given two generalized sketches S and S' over the same G and K , a sketch morphism from S to S' is a morphism m in G from $|S|$ to $|S'|$ such that for every $i \in I$ and every $k \in K_i[S]$, $m \circ k \in K_i[S']$.

In other words, a sketch morphism is a morphism of the underlying object that preserves features. We write $G \mid K$ the category of sketches over G and K using this notion of morphisms.

Remark 1. The models of a sketch form a category, where morphisms are the functors between categories that preserve the interpretation of the different objects. This category may not have an initial object; therefore, we cannot define the notion of free category over a sketch. Sketches are meant to describe structures in categories, such as group objects, and not the categories themselves.

22. A globe is a pair of two parallel morphisms. The name globe comes from their usual representation :



[58] Makkai, *Generalized sketches as a framework for completeness theorems. Part I*, 1997

[59] Diskin and Wolter, *Generalized Sketches : A Universal Logic for Diagrammatic Modeling in Software Engineering*, 2007

Definition 7. A sketch axiom is a morphism $r : R \rightarrow R'$ in $G \mid K$. A sketch $S \in G \mid K$ is said to satisfy r , written $S \models r$ if for every $\phi : R \rightarrow S$, ϕ factors through r :

$$\begin{array}{ccc} R & \xrightarrow{r} & R' \\ & \searrow \phi & \swarrow \text{dotted} \\ & S & \end{array}$$

Intuitively, this implies that for every instance of R as a sub-sketch of S , it can be extended in the manner described by r . We refer you to Makkai's work [58] for a more complete description of this notion and examples.

While Makkai then uses G as the category of graphs, and encodes all the axioms of categories as sketch axioms, we believe that for both performance and convenience, it is better to keep those axioms handled at a lower level. Therefore, we must define our own category to better capture the objects we manipulate in the implementation.

Definition 8. Let G be a finite graph. We name $\mathcal{F}(G)$ the free category over G . We name $\mathbb{G}$ the category of finite graphs where morphisms between G and G' are the graph morphisms from G to the underlying graph of $\mathcal{F}(G')$. The free property guarantees that these are exactly the functors from $\mathcal{F}(G)$ to $\mathcal{F}(G')$, so they have a well defined notion of composition and identity.

A sketch in $\mathbb{G}$ is then a graph sketch, but edges in shapes can be mapped to paths in the base graph. Using paths instead of explicit composition allows us to manipulate them modulo associativity and unitality.

Then we should have a collection of feature names I that is extensible. To support this, we would describe the map of features as a partial map. Each feature name is associated with a feature shape. The proof assistant plugin must send a graph to the interface, and instead of equalities, a partial map of feature names to sets of feature instances in the graph. Equality can be handled as a specific kind of feature with a globular shape.

Actions can be described through named sketch axioms. Application would work a bit like lemmas, where the source sketch needs to be mapped into the current goal, and a pushout is constructed that adds or merges new features. The plugins on the proof assistant side can then advertise the axioms verified by the sketch. For example if we have an hypothesis that the category we are considered is cartesian, the plugin could advertise that the sketch axiom that allows the construction of the product of two object is verified by the sketch.

13.3.2 Julia

Supporting a better scripting language has several advantages. It enables more generic proof scripts, that can be reused across proofs and proof assistants. It allows the user to extend functionality without having to edit and recompile the Rust source code. It could even allow some features to be moved from the Rust code to the scripting language, making it easier and faster to add new features and interactions.

[58] Makkai, *Generalized sketches as a framework for completeness theorems. Part I*, 1997

Remark 2. Another way to formulate definition 8 is to say morphisms from G to G' maps nodes to nodes and edges in G to paths in G' in a way compatible with the source and target of respectively edges and paths.

Remark 3. Instead of allowing arbitrary graphs as shapes, we may want to create a simpler algebra of shapes. For example, supporting globules, lines, triangles, rectangles, cones, and cocones over another shape...Doing so allows the interface to offer a higher level of control over the shapes. Keybindings or actions could be used to easily select the top of a cone, or the left side of a rectangle.

Remark 4. The lemma system described in chapter 8 is more powerful than the system of sketch axioms; therefore, it is not obvious that we actually need to implement and support sketch axioms. However, we think of the sketch axioms not as a replacement for lemmas but as a system to make the actions supported extendable. So, what they lose in power they should gain in ease of use and integration. Furthermore, because the names of the sketch axioms are the same across all the proof assistants, they can form the basis of proof scripts that are more independent of the underlying formalization, and so can be shared more easily. Of course, this is all very theoretical, as we have not started to implement any of it.

Creating a new language is a complex task, and it would be unfamiliar to the users. However, reusing an existing language comes with challenges. We need it to be deterministic, so that proofs can be replayed, and the state of the interpreter needs to support rollbacks. An obvious choice for an embedded language is *Lua*⁽²⁸⁾. This language has been made to be easy and efficient to embed. Furthermore, it provides low level access to its interpreter as a library; therefore, it should be feasible to support rollbacks.

However, having a relevant ecosystem and being familiar to the relevant people is also important. Another language seems to perform much better on this front : *Julia*⁽¹⁷⁾. Indeed, it has an entire ecosystem for *algebraic computation*⁽²⁹⁾, including two libraries for computational category theory, *CatLab.jl*⁽¹⁵⁾ and *CAP*⁽¹⁶⁾, a library for categorical graph rewriting, *AlgebraicRewriting.jl*⁽³⁰⁾ and a library for abstract algebraic computation, *Metatheory.jl*⁽³¹⁾.

Our hope is that if we were to use Julia as the scripting language, our interface could serve as an interface between those developments in computational category theory and proof assistants, enabling the creation of powerful proof generating tactics. However, the question of how to ensure that the execution is deterministic and that rollbacks are supported is no longer obvious. Most of the work has already been done through the development of *Pluto.jl*⁽³²⁾, an interactive notebook for Julia aimed at scientific computing. They ensure determinism and on-the-fly recomputation as the code is edited, so it is possible to have these features with Julia. We may even be able to reuse their code when embedding Julia into our interface.

13.4 Web version

Installing a Coq plugin can be complicated. Indeed, the plugin needs to be for the exact Coq version, but must also be compiled with the same version of the OCaml compiler that was used to compile Coq. In our case, we also need to install the interface, developed in Rust. While Opam²³ and Cargo²⁴ can take care of most of this, there is still trouble to ensure the Rust version is recent enough for our interface, and that the right system libraries have been installed.

Although we provide a *Nix*⁽³³⁾ script to install all of it, it comes with its own set of complications. A common request to circumvent this problem is to provide a web version. Rust can be compiled to Web Assembly, the *graphical framework*⁽²⁷⁾ we use support that, and Coq has a *JavaScript version*⁽³⁴⁾, so it should be possible.

However, integrating our plugin with CoqJS seems complicated, particularly when handling the communication between the plugin and the interface compiled to web assembly. Therefore, we decided to organize it differently. The web version of the interface, when started, begins a *websocket*²⁵ connection to the server. We implemented a simple websocket server in Rust that starts the Coq process on an incoming connection, and channels all communication with it through the websocket. In this way, Coq and the plugin do not need to be compiled to JavaScript because they are executed on the server. The architecture is illustrated in [figure 13.1](#).

One limitation of this approach is that there is no way for the user to interact with the Coq script. The script must be written on the

Remark 5. One not trivial question we have yet to consider is how the names of features and sketch axioms are attributed. If the interface itself is not extendable, it is easy : the interface code attributes the names, then there is a single source of truth and no risk of collision. Each plugin then selects the features that it can implement from the description. However, if as we hope the features can be added by external scripts that are loaded in the interface, there is a risk that two scripts use the same name for different features. If this is ever implemented, we need to define a good way to attribute names or scope them to avoid problems in the long term.

- (28) Lua
- (17) Julia
- (29) Algebraic Julia
- (15) Catlab.jl
- (16) Categories, Algorithms, Programming
- (30) AlgebraicRewriting.jl
- (31) Metatheory.jl
- (32) Pluto.jl

23. The OCaml package manager.

24. The Rust package manager.

(33) Nix

(27) Egui

(34) The Coq Proof Assistant in your browser

25. Websockets are a technology for real time communication between a client and a server.

server, and only our interface is exposed to the user. However, this seems sufficient to demonstrate our interface, and it may be useful for educational purposes. Indeed, the teacher of a mathematical course could create exercises in Coq and direct students to a web page where they can play with the graph to solve the problem.

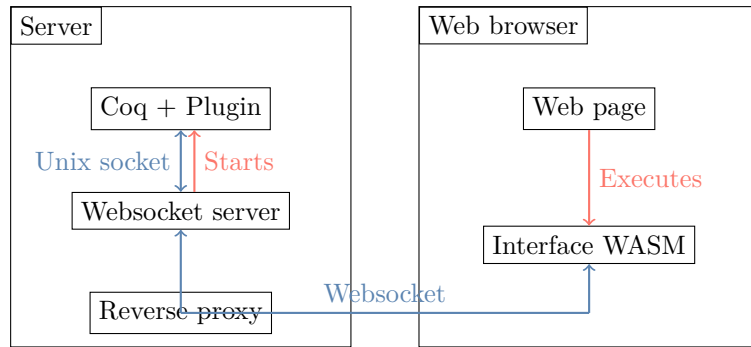


Figure 13.1 Architecture of the web version.

We believed that all that was needed was to change the connection method of the interface to use web sockets. However, we used a synchronous communication channel, and in the browser it is only possible to use asynchronous communication. Thankfully, Rust has a good builtin support for asynchronous coroutines²⁶. However, we still need to rewrite much of the interface code in an asynchronous style. We are in the process of doing so, so we cannot provide a link to test the interface.

²⁶. Asynchronous coroutines means turning functions into a state graph that can be interrupted and resumed on different states.

Remark 6. While the motivating aspect of doing this rewrite is to support a web version, asynchronous communication has other benefits. It makes the interface more resilient to hangups in the plugin. Indeed, with the current implementation, if the plugin takes a few seconds to compute something, the interface is frozen during that time. With the asynchronous version, interaction will be limited but still possible.

A Software list

- ⟨1⟩ Lambdapi, a proof assistant based on the $\lambda\Pi$ -calculus modulo rewriting
Frédéric Blanqui
🌐 <https://lambdapi.readthedocs.io/>
</> <https://github.com/Deducteam/lambdapi>
- ⟨2⟩ Quiver: a modern commutative diagram editor
🌐 <https://q.uiver.app/>
</> <https://github.com/varkor/quiver>
- ⟨3⟩ Globular
</> <https://github.com/jamievicary/globular>
[60] Globular: an online proof assistant for higher-dimensional rewriting
- ⟨4⟩ homotopy.io - A proof assistant implementing the theory of associative n-categories
🌐 <https://homotopy.io/>
</> <https://github.com/homotopy-io/homotopy-rs>
- ⟨5⟩ Actema : une interface graphique et gestuelle pour preuves formelles
Pablo Donato, Pierre-Yves Strub; Benjamin Werner
🌐 <https://actema.xyz>
</> <https://gitlab.com/prover-ui/prover>
[61] A drag-and-drop proof tactic
- ⟨6⟩ ProofWidgets
🌐 <https://lean-lang.org/lean4/doc/examples/widgets.lean.html>
</> <https://github.com/leanprover-community/ProofWidgets4>
- ⟨7⟩ Commutative Diagrams Interface
Luc Chabassier
</> <https://github.com/dwarfmaster/commutative-diagrams>
- ⟨8⟩ The Rock Proof Assistant
🌐 <https://rocq-prover.org/>
</> <https://github.com/rocq-prover/rocq>



⟨9⟩ A diagram editor in your browser

Ambroise Lafont

🌐 <https://amblafont.github.io/graph-editor/index.html>

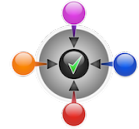
</> <https://github.com/amblafont/graph-editor-web>

⟨10⟩ Dedukti, a logical framework

🌐 <https://deducteam.github.io/>

</> <https://github.com/Deducteam/Dedukti>

[62] Dedukti : a Logical Framework based on the $\lambda\Pi$ -Calculus Modulo Theory



⟨11⟩ Coq LSP

Ali Caglayan, Emilio J. Gallego Arias, Shachar Itzhaky

</> <https://github.com/ejgallego/coq-lsp>

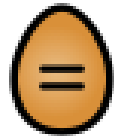
⟨12⟩ Egg

Max Willsey, Chandrakana Nandi, Yisu Remy Wang et al.

🌐 <https://egraphs-good.github.io/>

</> <https://github.com/egraphs-good/egg>

[63] egg: Fast and Extensible Equality Saturation



⟨13⟩ Z3

🌐 <https://www.microsoft.com/en-us/research/project/z3-3/>

</> <https://github.com/Z3Prover/z3>

[64] Z3: An Efficient SMT Solver

⟨14⟩ NuPRL

Robert Lee Constable

🌐 <https://nuprl-web.cs.cornell.edu/>

[65] The Nuprl Open Logical Environment

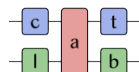
[66] Implementing mathematics with the Nuprl proof development system

⟨15⟩ Catlab.jl

🌐 <https://algebraicjulia.github.io/Catlab.jl/dev/>

</> <https://github.com/AlgebraicJulia/Catlab.jl>

[67] Compositional Scientific Computing with Catlab and SemanticModels



⟨16⟩ Categories, Algorithms, Programming

🌐 https://homalg-project.github.io/CAP_project/CAP/

</> https://github.com/homalg-project/CAP_project

[68] On the Syntax and Semantics of CAP

⟨17⟩ Julia

🌐 <https://julialang.org/>

</> <https://github.com/JuliaLang/julia>

[69] Julia: A Fast Dynamic Language for Technical Computing



⟨18⟩ OCaml

🌐 <https://ocaml.org/>

</> <https://github.com/ocaml/ocaml>



⟨19⟩ MessagePack protocol

🌐 <https://msgpack.org/index.html>

</> <https://github.com/msgpack/msgpack>

⟨20⟩ MessagePack Remote Procedure Call

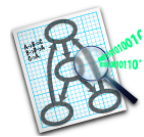
</> <https://github.com/msgpack-rpc/msgpack-rpc>

⟨21⟩ Graphviz

🌐 <https://graphviz.org/>

</> <https://gitlab.com/graphviz/graphviz>

[70] Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools



⟨22⟩ hashconsing crate

🌐 <https://docs.rs/hashconsing/latest/hashconsing/>

</> <https://github.com/AdrienChampion/hashconsing>

[53] Type-Safe Modular Hash-Consing

⟨23⟩ Hierarchy Builder

Cyril Cohen, Kazuhiko Sakaguchi; Enrico Tassi

</> <https://github.com/math-comp/hierarchy-builder>

[71] Hierarchy Builder: algebraic hierarchies made easy in Coq with Elpi

⟨24⟩ Soufflé

🌐 <https://github.com/souffle-lang/souffle>

</> <https://github.com/math-comp/hierarchy-builder>

[72] Soufflé: On Synthesis of Program Analyzers



⟨25⟩ Coq-Elpi : Coq plugin embedding ELPI

Enrico Tassi

</> <https://github.com/LPCIC/coq-elpi>

[73] Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λ Prolog dialect)



⟨26⟩ The Rust programming language

🌐 <https://www.rust-lang.org/>

</> <https://github.com/rust-lang/rust>



⟨27⟩ Egui

🌐 <https://www.egui.rs/>

</> <https://github.com/emilk/egui>

⟨28⟩ Lua

🌐 <https://lua.org/>

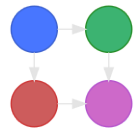
[74] Lua 5.1 Reference Manual



⟨29⟩ Algebraic Julia

🌐 <https://www.algebraicjulia.org/>

</> <https://github.com/AlgebraicJulia>

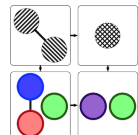


⟨30⟩ AlgebraicRewriting.jl

🌐 <https://algebraicjulia.github.io/AlgebraicRewriting.jl/dev/>

</> <https://github.com/AlgebraicJulia/AlgebraicRewriting.jl>

[75] Computational category-theoretic rewriting



⟨31⟩ Metatheory.jl

🌐 <https://julasymbolics.github.io/Metatheory.jl/v0.6/>

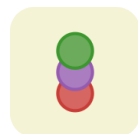
</> <https://github.com/JuliaSymbolics/Metatheory.jl>

[76] Metatheory.jl: Fast and Elegant Algebraic Computation in Julia with Extensible Equality Saturation

⟨32⟩ Pluto.jl

🌐 <https://plutojl.org>

</> <https://github.com/fonsp/Pluto.jl>



⟨33⟩ Nix
Eelco Dolstra
🌐 <https://nixos.org/>
</> <https://github.com/NixOS/nix>
[77] The purely functional software deployment model.



⟨34⟩ The Coq Proof Assistant in your browser
🌐 <https://jscoq.github.io/wa/>
</> <https://github.com/jscoq/jscoq>

B Bibliography

- [1] P.T. Johnstone, *Sketches of an elephant: a Topos theory compendium* (Oxford Univ. Press, New York, NY, 2002). (p. 3)
- [2] B. Fong and D.I. Spivak, *Seven Sketches in Compositionality: An Invitation to Applied Category Theory*, (2018). (p. 3)
- [3] J.C. Baez and M. Stay, Physics, Topology, Logic and Computation: A Rosetta Stone, *arXiv:0903.0340 [quant-ph]* **813** 95–172 (2010). (arXiv: 0903.0340) (p. 3)
- [4] V. Voevodsky, The origins and motivations of Univalent Foundations, *The Institute Letter* (2014). (p. 3)
- [5] M. Dowson, The Ariane 5 software failure, *SIGSOFT Softw. Eng. Notes* **22**(2), 84 (1997). (p. 3)
- [6] K. Appel and W. Haken, Every Planar Map is Four Colorable, *Contemporary Mathematics* (1989). (p. 3)
- [7] P. Scholze, Liquid Tensor Experiment, *Experimental Mathematics* **31**(2), 349–354 (2022). (p. 3)
- [8] B. Ahrens, C. Kapulkin, and M. Shulman, Univalent categories and the Rezk completion, *Mathematical Structures in Computer Science* **25**(5), 1010–1039 (2015). (arXiv: 1303.0584) (p. 3 and p. 13)
- [9] J. Gross, A. Chlipala, and D.I. Spivak, Experience Implementing a Performant Category-Theory Library in Coq, *arXiv:1401.7694 [cs, math]* (2014). (arXiv: 1401.7694) (p. 3 and p. 13)
- [10] J.Z.S. Hu and J. Carette, Formalizing of Category Theory in Agda, *arXiv:2005.07059 [cs]* (2021). (arXiv: 2005.07059) (p. 3 and pp. 12–13)
- [11] E.W. Stark, Category Theory with Adjunctions and Limits, *Archive of Formal Proofs* (2016). (p. 3 and p. 11)
- [12] S. Eilenberg and S. MacLane, General theory of natural equivalences, *Transactions of the American Mathematical Society* **58**(0), 231–294 (1945). (p. 9)
- [13] F.-W. Lawvere, *An Elementary Theory of the Category of Sets* (University of Chicago, 1964). (p. 9)
- [14] J. Lambek, P.J. Scott, and P.J. Scott, *Introduction to higher order categorical logic*, no. 7, Paperback ed. (with corr.), reprinted ed. (Cambridge Univ. Press, Cambridge, 1994). (p. 9)
- [15] P. Martin-Löf, An Intuitionistic Theory of Types: Predicative Part, In H.-E. Rose and J.-C. Shepherdson (Eds.) *Studies in Logic and the Foundations of Mathematics*, Vol. 80, pp. 73–118 (Elsevier, 1975). (p. 12)
- [16] P. Martin-Löf, A Theory of Types, *Unpublished* (1971). (p. 12)
- [17] T.U.F. Program, *Homotopy Type Theory: Univalent Foundations of Mathematics* (<https://homotopytypetheory.org/book>, Institute for Advanced Study, 2013). (p. 12, p. 30, p. 32, and p. 101)
- [18] T. Streicher, *Investigations into intensional type theory* (1993). (pp. 31–32)

- [19] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg, Cubical Type Theory: a constructive interpretation of the univalence axiom, *arXiv:1611.02108 [cs, math]* (2016). (arXiv: 1611.02108)
(p. 32 and p. 39)
- [20] T. Altenkirch, C. McBride, and W. Swierstra, Observational equality, now!, In Proceedings of the 2007 workshop on Programming languages meets program verification - PLPV '07, p. 57 (ACM Press, Freiburg, Germany, 2007).
(p. 32)
- [21] D. Cousineau and G. Dowek, *Embedding Pure Type Systems in the lambda-Pi-calculus modulo*, (2023). (arXiv:2310.12540 [cs])
(p. 32)
- [22] A. Bauer, G. Gilbert, P.G. Haselwarter, M. Pretnar, and C.A. Stone, Design and Implementation of the Andromeda Proof Assistant, *arXiv:1802.06217 [cs]* (2018). (arXiv: 1802.06217)
(p. 35)
- [23] M. Hofmann, Conservativity of equality reflection over intensional type theory, In G. Goos, J. Hartmanis, J. Leeuwen, S. Berardi, and M. Coppo (Eds.) *Types for Proofs and Programs*, Vol. 1158, pp. 153–164 (Springer Berlin Heidelberg, Berlin, Heidelberg, 1996). (Series Title: Lecture Notes in Computer Science)
(p. 36 and p. 41)
- [24] N. Oury, Extensionality in the Calculus of Constructions, In J. Hurd and T. Melham (Eds.) *Theorem Proving in Higher Order Logics*, pp. 278–293 (Springer, Berlin, Heidelberg, 2005).
(p. 36 and p. 41)
- [25] T. Winterhalter, M. Sozeau, and N. Tabareau, Eliminating reflection from type theory, In Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, pp. 91–103 (Association for Computing Machinery, New York, NY, USA, 2019).
(p. 36, p. 39, and p. 41)
- [26] N.-G. De Bruijn, Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem, *Indagationes Mathematicae (Proceedings)* **75**(5), 381–392 (1972).
(p. 36)
- [27] D. Annenkov, P. Capriotti, N. Kraus, and C. Sattler, Two-Level Type Theory and Applications, *arXiv:1705.03307 [cs]* (2019). (arXiv: 1705.03307)
(p. 39)
- [28] V.-I. Glivenko, Sur quelques points de la logique de M. Brouwer, *Roy. Belg. Bull. Cl. Sci.* **15** 183–188 (1929).
(p. 39)
- [29] P. Dybjer, A general formulation of simultaneous inductive-recursive definitions in type theory, *Journal of Symbolic Logic* **65**(2), 525–549 (2000).
(p. 40)
- [30] T. Evans, Word problems, *Bulletin of the American Mathematical Society* **84**(5), 789–802 (1978).
(p. 68)
- [31] W.W. Boone, THE WORD PROBLEM, *Proceedings of the National Academy of Sciences* **44**(10), 1061–1065 (1958).
(p. 68)
- [32] J.P. May, *A concise course in algebraic topology* (University of Chicago Press, Chicago, 1999).
(p. 69)
- [33] L. Fajstrup, E. Goubault, E. Haucourt, S. Mimram, and M. Raussen, *Directed Algebraic Topology and Concurrency* (Springer International Publishing, Cham, 2016).
(p. 69)
- [34] G. Hondet and F. Blanqui, *The New Rewriting Engine of Dedukti*, (2022). (arXiv:2010.16115 [cs])
(p. 71)
- [35] G. Nelson and D.C. Oppen, Fast Decision Procedures Based on Congruence Closure, *J. ACM* **27**(2), 356–364 (1980). (Place: New York, NY, USA Publisher: Association for Computing Machinery)
(p. 72)

- [36] R. Nieuwenhuis and A. Oliveras, *Proof-Producing Congruence Closure*, Vol. 3467 (2005). (Journal Abbreviation: Lecture Notes in Computer Science Pages: 468 Publication Title: Lecture Notes in Computer Science) (p. 72)
- [37] M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner, A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses, In J.-P. Jouannaud and Z. Shao (Eds.) *Certified Programs and Proofs*, pp.. 135–150 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2011). (p. 72)
- [38] D. Kozen, C. Kreitz, and E. Richter, Automating Proofs in Category Theory, In D. Hutchison, T. Kanade, J. Kittler, J.M. Kleinberg, F. Mattern, J.C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen *et al.* (Eds.) *Automated Reasoning*, Vol. 4130, pp.. 392–407 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2006). (Series Title: Lecture Notes in Computer Science) (p. 73)
- [39] C. Kreitz, Nuprl as Logical Framework for Automating Proofs in Category Theory, In *Logic and Program Semantics* (2012). (p. 73)
- [40] R.E. Tarjan and J. Van Leeuwen, Worst-case Analysis of Set Union Algorithms, *Journal of the ACM* **31**(2), 245–281 (1984). (p. 92)
- [41] S.G. Kobourov, Force-Directed Drawing Algorithms, In *Handbook of Graph Drawing and Visualization* (2013). (p. 94)
- [42] T. Kamada and S. Kawai, An algorithm for drawing general undirected graphs, *Information Processing Letters* **31**(1), 7–15 (1989). (p. 94)
- [43] S.M. Lane, Natural associativity and commutativity, *Rice University Studies* (1963). (p. 100)
- [44] P. Selinger, A survey of graphical languages for monoidal categories, Vol. 813, pp.. 289–355 (2010). (arXiv:0908.3347 [math]) (p. 102)
- [45] P.-A. Mellès, Functorial Boxes in String Diagrams, In D. Hutchison, T. Kanade, J. Kittler, J.M. Kleinberg, F. Mattern, J.C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen *et al.* (Eds.) *Computer Science Logic*, Vol. 4207, pp.. 1–30 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2006). (Series Title: Lecture Notes in Computer Science) (p. 103)
- [46] G. Hotz, Eine Algebraisierung des Syntheseproblems von Schaltkreisen I, *J. Inf. Process. Cybern.* **1** 185–205 (1965). (p. 103)
- [47] K. Culik and D. Wood, A note on some tree similarity measures, *Information Processing Letters* **15**(1), 39–42 (1982). (p. 105)
- [48] D.D. Sleator, R.E. Tarjan, and W.P. Thurston, Rotation distance, triangulations, and hyperbolic geometry, *Journal of the American Mathematical Society* **1**(3), 647–681 (1988). (p. 105)
- [49] H. Li and G. Xia, An $O(3.82^k)$ Time FPT Algorithm for Convex Flip Distance, *Discrete & Computational Geometry* (2023). (p. 105)
- [50] S. Cleary and K.S. John, A Linear-Time Approximation Algorithm for Rotation Distance, *Journal of Graph Algorithms and Applications* **14**(2), 385–390 (2010). (p. 105)
- [51] S. Cleary, Restricted rotation distance between binary trees, *Information Processing Letters* **84**(6), 333–338 (2002). (p. 106)
- [52] M.-S. Paterson and M.-N. Wegman, Linear unification, *Journal of Computer and System Sciences* **16**(2), 158–167 (1978). (p. 107)

- [53] S. Conchon and J.-C. Filliâtre, Type-Safe Modular Hash-Consing, In ACM SIGPLAN Workshop on ML (Portland, Oregon, 2006). (p. 108 and p. 127)
- [54] L. Bahiense, G. Manić, B. Piva, and C.C. de Souza, The maximum common edge subgraph problem: A polyhedral investigation, *Discrete Applied Mathematics* **160**(18), 2523–2541 (2012). (p. 108)
- [55] F. Guidi, C. Sacerdoti Coen, and E. Tassi, *Implementing Type Theory in Higher Order Constraint Logic Programming* (2017). (p. 112)
- [56] C. Ehresmann, Esquisses et types de structures algébriques, *Buletinul Institutului Politehnic Din IASI* **14** (1968). (p. 118)
- [57] A. Bastiani and C. Ehresmann, Categories of sketched structures, *Cahiers de Topologie et Géométrie Différentielle Catégoriques* **13**(2), 104–214 (1972). (Publisher: Dunod éditeur, publié avec le concours du CNRS) (p. 118)
- [58] M. Makkai, Generalized sketches as a framework for completeness theorems. Part I, *Journal of Pure and Applied Algebra* **115**(1), 49–79 (1997). (pp. 119–120)
- [59] Z. Diskin and U. Wolter, Generalized Sketches : A Universal Logic for Diagrammatic Modeling in Software Engineering (2007). (p. 119)
- [60] K. Bar, A. Kissinger, and J. Vicary, Globular: an online proof assistant for higher-dimensional rewriting, In *Leibniz International Proceedings in Informatics*, Vol. 52, pp.. 34:1–34:11 (2016). (p. 125)
- [61] P. Donato, P.-Y. Strub, and B. Werner, A drag-and-drop proof tactic, In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp.. 197–209 (ACM, Philadelphia PA USA, 2022). (p. 125)
- [62] A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard, *Dedukti : a Logical Framework based on the $\lambda\Pi$ -Calculus Modulo Theory*, (2016). (p. 126)
- [63] M. Willsey, C. Nandi, Y.R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, Egg: Fast and Extensible Equality Saturation, *Proceedings of the ACM on Programming Languages* **5**(POPL), 1–29 (2021). (arXiv:2004.03082 [cs]) (p. 126)
- [64] L. de Moura and N. Bjørner, Z3: An Efficient SMT Solver, In C.R. Ramakrishnan and J. Rehof (Eds.) *Tools and Algorithms for the Construction and Analysis of Systems*, pp.. 337–340 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2008). (p. 126)
- [65] S.F. Allen, R.L. Constable, R. Eaton, C. Kreitz, and L. Lorigo, The Nuprl Open Logical Environment, In D. McAllester (Ed.) *Automated Deduction - CADE-17*, pp.. 170–176 (Springer, Berlin, Heidelberg, 2000). (p. 126)
- [66] R.L. Constable and C. University, *Implementing mathematics with the Nuprl proof development system* (Prentice-Hall, Englewood Cliffs, N.J, 1986). (p. 126)
- [67] M. Halter, E. Patterson, A. Baas, and J. Fairbanks, *Compositional Scientific Computing with Catlab and SemanticModels*, (2020). (arXiv:2005.04831 [cs, math]) (p. 126)
- [68] S. Gutsche, S. Posur, and Ø. Skartsæterhagen, On the Syntax and Semantics of CAP, In *CICM Workshops* (2018). (p. 126)
- [69] J. Bezanson, S. Karpinski, V.B. Shah, and A. Edelman, *Julia: A Fast Dynamic Language for Technical Computing*, (2012). (arXiv:1209.5145 [cs]) (p. 127)

- [70] J. Ellson, E. Gansner, E. Koutsofios, S. North, and G. Woodhull, Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools, (2004). (p. 127)
- [71] C. Cohen, K. Sakaguchi, and E. Tassi, Hierarchy Builder: algebraic hierarchies made easy in Coq with Elpi, In FSCD 2020 - 5th International Conference on Formal Structures for Computation and Deduction, pp.. 34:1–34:21 (Paris, France, 2020). (Issue: 167) (p. 127)
- [72] H. Jordan, B. Scholz, and P. Subotić, Soufflé: On Synthesis of Program Analyzers, In International Conference on Computer Aided Verification (2016). (p. 127)
- [73] E. Tassi, *Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λ Prolog dialect)* (2018). (p. 128)
- [74] L.H.D. Figueiredo, *Lua 5.1 Reference Manual* (Roberto Ierusalimsky, 2006). (p. 128)
- [75] K. Brown, E. Patterson, T. Hanks, and J. Fairbanks, *Computational category-theoretic rewriting*, (2023). (arXiv:2111.03784 [cs]) (p. 128)
- [76] A. Cheli, Metatheory.jl: Fast and Elegant Algebraic Computation in Julia with Extensible Equality Saturation, *Journal of Open Source Software* **6**(59), 3078 (2021). (Publisher: The Open Journal) (p. 128)
- [77] E. Dolstra, *The purely functional software deployment model.*, (PhD thesis) (2006). (p. 129)

While the use of proof assistants in mathematical research has yet to become the norm, an increasing number of results have been formalized. The prevalence of category theory in recent mathematical research indicates the need for its formalization in proof assistants. However, category theory is challenging to formalize for reasons we will explore.

The first difficulty is due to the fact that proposition equality in dependently typed theories as a structure that interacts in a complicated way with categories formalised inside those theories. We describe those challenges, and propose some solutions we explored to this problem.

The second difficulty is more pragmatic. One big advantage of category theory is its ability to be reasoned with using graphical languages. However, most proof assistants are text based; thus their notation systems cannot express any graphical language. We implement a plugin to graphically progress category theory proofs in the Coq proof assistant. After a quick demo, we will describe the different challenges in the implementation of such a software.