# Constructive Category Theory and Applications to Algebraic Geometry

DISSERTATION

zur Erlangung des Grades eines
Doktors der Naturwissenschaften

vorgelegt von
Sebastian Gutsche

eingereicht bei der Naturwissenschaftlich-Technischen Fakultät
der Universität Siegen
Siegen 2017

Gutachter

Prof. Dr. Mohamed Barakat, Universität Siegen
Prof. Dr. Max Horn, Universität Gießen

Tag der mündlichen Prüfung: 29. August 2017

gedruckt auf alterungsbeständigem holz- und säurefreiem Papier

## Preface

I would like to thank all people who supported and helped me in the years I worked on this thesis.

I wish to express my sincere gratitude to Mohamed Barakat, who introduced me to toric geometry, constructive category theory, and many additional projects I worked on during the last years. Without him and his endless help and patience this thesis would not exist.

Special thanks also to my dear colleague Sebastian Posur for countless fruitful discussions and the great time we had developing our software project CAP.

Furthermore, I like to thank Max Horn for many discussions, a lot of input for this thesis, and many hours of working together on other various topics and projects.

I would like to thank all my colleagues in Aachen, Kaiserslautern, and Siegen for the great times I had there. Special thanks to Wilhelm Plesken for letting me keep my office in Aachen for the last years.

Last, I would like to thank my family: my mother, my father, my grandmother, my brothers, my uncle, my girlfriend Maike, and all the others for supporting me during this time.

## Summary

In this thesis we design a framework for computing in (abelian) categories in a structured manner, inspired by constructions in category theory.

We start by giving necessary definitions for a category to be computable in the sense of this thesis. This includes the requirements on the data structure for objects and morphisms, and the specifications of categorical operations which need to be implemented.

As a first example, we provide data structures and algorithms to show how the category of finitely presented graded modules over a graded computable ring can be implemented in this context.

Then we describe the category of Serre morphisms of an abelian category. It provides an example of the flexibility a categorical framework offers for the implementation of abelian categories. The category of Serre morphisms will then be used, together with the previously described implementation of f.p. graded modules, to implement the category of coherent sheaves over a normal toric variety. To achieve this, we present an algorithm to compute the graded parts of a f.p. graded module over a Laurent polynomial ring, the latter graded by a finitely presented abelian group.

As application of this axiomatic computational setup for both f.p. graded modules and coherent sheaves over toric varieties, we describe a categorical algorithm to compute a *grade-compatible presentation* of a f.p. graded module and a coherent sheaf.

A realization of the categorical framework to implement computable categories was created alongside this thesis: CAP (Categories, Algorithms, Programming). All concepts and algorithms presented in this thesis are implemented in CAP. In the last chapter of the thesis, some technical concepts of CAP are explained and motivated.

## Zusammenfassung

In dieser Arbeit definieren wir einen durch die Konstruktionen der Kategorientheorie definierten Rahmen, um abelsche Kategorien auf dem Computer zu implementieren und mit diesen zu arbeiten.

Wir beginnen mit der Definition einer berechenbaren Kategorie im Sinne dieser Arbeit. Dies beinhaltet die Anforderungen an die Datenstrukturen für Objekte und Morphismen und die Spezifikationen der kategoriellen Operationen, die implementiert werden sollen.

Als erstes Beispiel definieren wir Datenstrukturen und geben Algorithmen an, um zu zeigen, dass die Kategorie der endlich präsentierten graduierten Moduln über einem berechenbaren Ring ebenfalls berechenbar ist.

Anschließend beschreiben wir die Kategorie der Serre Morphismen einer abelschen Kategorie $\mathcal{A}$ bezüglich einer dicken Teilkategorie von $\mathcal{A}$. Diese Serre Morphismen Kategorie bietet ein Beispiel für die Flexibilität des kategoriellen Rahmens der Implementation abelscher Kategorien. Wir benutzen die Serre Morphismen Kategorie, zusammen mit der Implementation endlich präsentierter graduierter Moduln, um ein berechenbares Modell kohärenter Garben über torischen Varietäten zu beschreiben. Um dies zu erreichen, stellen wir einen Algorithmus vor, der die Gradschichten eines endlich präsentierten graduierten Moduls über einem mit einer endlich präsentierten abelschen Gruppe graduierten Laurent Polynomring berechnet.

Als Anwendung dieser axiomatischen Implementation der endlich präsentierten graduierten Moduln und kohärenten Garben geben wir abschließend einen Algorithmus an, welcher *Reinheitsgrad-kompatible* Präsentationen endlich präsentierter graduierter Moduln und kohärenter Garben berechnet.

Eine Realisierung dieses axiomatisch-kategoriellen Rahmens zur Implementation berechenbarer Kategorien entstand zusammen mit dieser Arbeit: CAP (Categories, Algorithms, Programming). Alle Konzepte und Algorithmen, welche in dieser Arbeit vorgestellt werden, sind bereits in CAP implementiert. Im letzten Kapitel motivieren und erklären wir zudem einige technische Konzepte von CAP.

# Contents

Chapter I. Introduction 11

Chapter II. Computability of categories 13
1. Computable functions and decidable sets 13
2. Categories with Hom-setoids 15
3. Categories with Hom-setoids vs. classical categories 16
4. Computable categories 18
5. Decidable properties 20
6. Preadditive categories 21
7. Additive categories 23
8. Preabelian categories 26
9. Abelian categories 30
10. Categorical notions 31

Chapter III. Implementation of graded modules 33
1. The category of graded module presentations 33
2. Computability of graded module presentations 36

Chapter IV. Generalized morphisms and Serre quotients 47
1. The category of generalized morphisms 47
1.a Preliminaries 47
1.b Generalized morphisms by cospans 50
1.c Generalized morphisms by spans 53
1.d Generalized morphisms by 3-arrows 56
1.e Conversion between different types of generalized morphisms 63
2. Structure of the category of generalized morphisms 67
3. Generalized and pseudo-inverse 69
4. Serre quotients 76
5. Computability of Serre quotients 79
5.a Computability of Serre morphisms by spans 79
5.b Decidability 90
5.c Computability of Serre morphisms by cospans 91
5.d Computability of Serre morphisms by 3-arrows 94

Chapter V. The category of coherent sheaves over a toric variety 99
1. Preliminaries from toric geometry 99

2. Equivalence of Serre quotient and coherent sheaves 103
3. Deciding membership of the kernel of the sheafification functor 104
3.a Hilbert polynomial for projective spaces 104
3.b Global sections 105
3.c A generating set for $S_0$ 106
3.d Relations between the generators of $S_0$ 108
3.e The homogeneous parts of $S$ 110
3.f A generating set for $S_\alpha$ 111
3.g Relations between the generating monomials of $\mathrm{Mon}\,(S_\alpha)$ 113
3.h Homogeneous parts of finitely presented modules 118

Chapter VI. Application 127
1. Preliminaries 127
2. Bicomplexes 134
3. Internal Hom and Ext 137
4. Grade filtration 139
5. Spectral sequences 142
6. Filtered presentation 145
7. Coherent sheaves 150

Chapter VII. Implementation of computable categories 159
1. The concept of categorical programming 159
2. Main design goal and feature 161
2.a Data structure agnosticism 161
2.b Selection options of basic categorical constructions 161
3. Error messages for categorical operations 162
4. Undecidable realizations 162
5. Ensuring compatibility: `WithGiven` operations 163
6. Caching 165
6.a Pointers and Garbage Collection 165
6.b Caches 166
6.c Caching in `GAP` 167
6.d Caching in CAP 168
6.e Avoiding the setoid 169
6.f Caching for compatibility 170
6.g Caching for performance 170
7. Primitive and derived categorical operations 171
7.a Why the graph is necessary: Circular dependencies 171
7.b The derivation graph 171
7.c Installation and derivation of operations 173
8. Logic Propagation: ToDoLists 174
8.a Property propagation: ToDoLists 174
8.b Creating ToDoList entries: The LaTeXlogic parser 175

Bibliography 177

Appendix A. Programming in CAP 179

Appendix B. Logical theorems in CAP 187

Appendix C. All method names 193

Appendix D. Derivations 199

Appendix E. Final Derivations 287

Appendix F. Installed basic operations 301

Appendix G. Application code 375

Appendix. Index 393

CHAPTER I

# Introduction

Many structures in abstract algebra and computer algebra can be organized as abelian categories. Many algorithms boil down to basic categorical constructions in abelian categories. In this thesis we provide a *framework* to organize the implementation of algebraic structures in a categorical fashion and show the flexibility and computational capabilities of this framework in various examples: Using the notion of *computable categories* we provide algorithms for computable descriptions of *finitely presented graded modules* and *coherent sheaves over normal toric varieties*, the latter modeled as a *Serre quotient category*. Afterwards we present an algorithm to compute the so-called *grade* or *purity filtration* of both a f.p. graded module and a coherent sheaf. As a consequence of the high level of abstraction provided by a categorical implementation of this algorithm, we will be able to use the same algorithm for both f.p. graded modules and coherent sheaves.

In Chapter II we start by defining the type of category we use as a base for the definition of computable categories: *Categories with Hom-setoids*. We show how this type of category relates to the classical definition of a category. Then we define what we call a *realization of a category*. The realization will state all requirements we have for the data structure of a computable category. Using this realization, we state a minimal set of algorithms to be implemented for a given realization of a category $\mathcal{A}$ in order to render $\mathcal{A}$ *computable abelian*.

In Chapter III we describe the *category of graded module presentations* over a computable graded ring $S$. This is simultaneously our first involved example of a computable abelian category together with a realization and all necessary algorithms. The category of graded module presentations is a computable model of the category of f.p. graded modules over the ring $S$. The algorithms for this category will show the necessity of the definition of a computable category as category with Hom-setoids. Furthermore, we are going to use this computable description of f.p. graded modules to define a computable model of the category of *coherent sheaves over a toric variety*.

In Chapter IV, to show the flexibility of the developed categorical framework for implementing abelian categories, we give an instance of constructing a computable category out of another one: the *generalized morphism category* $\mathrm{G}(\mathcal{A})$ of a computable abelian category $\mathcal{A}$. While the generalized morphism category $\mathrm{G}(\mathcal{A})$ is already interesting by itself, for example to constructively perform diagram chases in the computable abelian category $\mathcal{A}$, we use it to describe another level of abstraction: the *Serre quotient category* $\mathcal{A}/\mathcal{C}$ of an abelian category $\mathcal{A}$ with respect to a thick subcategory $\mathcal{C} \subseteq \mathcal{A}$. The Serre quotient category $\mathcal{A}/\mathcal{C}$ is computable abelian if $\mathcal{A}$ is computable abelian and the membership of objects in $\mathcal{C}$ is decidable. We call the membership of objects in $\mathcal{C}$ decidable if there is an algorithm which decides for any object in $\mathcal{A}$ whether it is in $\mathcal{C}$. We describe three distinct data structures for generalized morphisms and establish the computability of the Serre quotient category for each of the three data structures.

In Chapter V, using the Serre quotient category we show that the category of *coherent sheaves over a normal toric variety* is computable abelian. For a normal toric variety $X$ with no torus factors every coherent sheaf is the sheafification of a f.p. graded module over the Cox ring $S$ of $X$. To model the category of coherent sheaves over $X$ we use the Serre quotient of the category of f.p. graded $S$-modules with respect to the thick subcategory of modules that sheafify to zero. Whether or not a f.p. graded module sheafifies to zero in this setup can be *decided* by computing the 0-th degree parts of certain finitely presented graded modules over Laurent polynomial rings. We give an algorithm to compute all graded parts of a f.p. graded module over such a ring. This algorithm establishes the decidability of the thick subcategory of f.p. graded $S$-modules that sheafify to zero, so we can prove that the category of coherent sheaves over a toric variety modeled by the said Serre quotient is computable abelian.

As an application of the computable versions of the categories of *f.p. graded modules* and *coherent sheaves* we give a purely categorical algorithm to compute the *grade* or *purity filtration* of a f.p. graded module and a coherent sheaf in Chapter VI. We will see that ensuring the computability of all involved categories in our categorical framework, we can apply the same algorithm to compute the *grade filtration* in both the f.p. graded module and the coherent sheaf context. So we see that the abstraction provided by the categorical framework leads to *highly abstract algorithms*, mimicking the constructions in category theory proofs.

The notion of computable categories and the defined constructions which mirror the existential quantifiers from the definition of an (abelian) category led us to the implementation of a categorical programming language, which itself is implemented as the `GAP` ([**GAP17**]) package CAP ([**GSP17**]). Such an implementation posed several challenges due to the many possible choices of data structures for categories, but also due to the undecidability of certain problems. We are going to address these challenges and provide possible solutions in Chapter VII. We will also highlight certain features implemented in CAP that help to provide a universal framework which still allows efficient computations, including the following:

- *Derivation of categorical constructions*, ensuring that only a small set of algorithms has to be implemented for an abelian category $\mathcal{A}$ to provide the possibility to computationally carry out all categorical constructions possible in $\mathcal{A}$.
- The *caching of results* of computations to make categorical constructions compatible, fast, and mimic "paper mathematics" as far as possible on a computer.

The categorical framework and all computable categories discussed in this thesis are implemented in the `GAP` package CAP. Parts of the code of the implementations can be found in the Appendices D to G.

CHAPTER II

# Computability of categories

This chapter sets the stage for computable categories, by giving all necessary definitions needed throughout this thesis. We introduce the notions of computability and computable abelian categories. We start by defining *computable functions* and *decidable sets*. Using these computable functions and decidable sets, we define the realization of a category, which will serve as data structure for a category. We then define when a certain category with a given realization is computable preadditive, additive, preabelian, and abelian, following the hierarchy of category types described in [**BLH11**, Appendix A].

We do not use the classical definition of a category, but an extended one called *category with Hom-setoids* to define the realization and a computable category.

This chapter contains no theory, and the running example is meant to illustrate the definitions. A more involved example for the definitions in this chapter will be given in Chapter III.

## 1. Computable functions and decidable sets

Before we can even talk about computability of a category we need to define what it means for functions to be computable and for mathematical data to be representable on a computer.

**Definition 1.1.** Let $\mathfrak{A}, \mathfrak{B} \subset \mathbb{N}$ and $\mathfrak{f}: \mathfrak{A} \to \mathfrak{B}$ a function. We call $\mathfrak{f}$ **computable** if there is a deterministic Turing machine $M$ which computes $\mathfrak{f}$.

The phrase "there is a deterministic Turing machine which computes $\mathfrak{f}$" can be rephrased by "there is an algorithm implementable on a computer which for every $\mathfrak{a} \in \mathfrak{A}$ computes $\mathfrak{f}(\mathfrak{a})$".

We now establish when a set is representable on a computer. A category can only be represented on a computer if objects and morphisms have finite data structures. Every data on a computer boils down to a finite sequence of natural numbers, and we can encode such a sequences as a single natural numbers in a computable way. So any computer representation of mathematical objects is just a natural number.

**Definition 1.2** (Decidable sets)**.** A subset $\mathfrak{A} \subset \mathbb{N}$ is **decidable** if there is a computable function

$$\texttt{IsContained}_{\mathfrak{A}} : \mathbb{N} \to \{0, 1\}$$

such that

$$\texttt{IsContained}_{\mathfrak{A}}(\mathfrak{a}) = 1 \Leftrightarrow \mathfrak{a} \in \mathfrak{A}.$$

More elaborately we say that $\mathfrak{A}$ is **decidable by** $\texttt{IsContained}_{\mathfrak{A}}$.

We will use decidable sets as data structures for objects and morphisms in computable categories. Note that not every subset of $\mathbb{N}$ is decidable. Also decidability of a set does not mean subsets are decidable as well: By Rice's Theorem, the set of computable functions is decidable, but every nontrivial subset thereof is undecidable.

**Definition 1.3** (Decidable equivalence relation)**.** Let $\mathfrak{A} \subset \mathbb{N}$ and $\approx \subset \mathfrak{A} \times \mathfrak{A}$ an equivalence relation. We say $\approx$ is **decidable** if there is a computable function

$$\texttt{IsEqual}_{\approx} : \mathfrak{A} \times \mathfrak{A} \to \{0, 1\}$$

such that

$$\texttt{IsEqual}_{\approx}(\mathfrak{a}, \mathfrak{b}) = 1 \Leftrightarrow \mathfrak{a} \approx \mathfrak{b}.$$

**Definition 1.4** (Realization of a set)**.** Let $A$ be a set. We say that the decidable set $\mathfrak{A} \subset \mathbb{N}$ is a **realization** of $A$ if there is a surjective map

$$\mathrm{Interpret}_A : \mathfrak{A} \twoheadrightarrow A,$$

such that the induced equivalence relation $\approx_{\mathrm{Interpret}_A}$ on $\mathfrak{A}$ is decidable. We call $\mathrm{Interpret}_A$ the **realization map**.

A set $A$ is representable on a computer if it has a realization, and the realization defines a data structure for the elements of the set. Note that the set $A$ itself does not need to be finite.

Remark 1.5.

(1) We have $A \cong \mathfrak{A}/\approx_{\mathrm{Interpret}_A}$ as sets.
(2) The notion of computability does not apply to $\mathrm{Interpret}_A$, since $A \not\subseteq \mathbb{N}$. It merely describes the interpretation of the computer data $\mathfrak{A}$ in the mathematical context of $A$.

We will now identify the set $\{0, 1\}$ with the set $\{\texttt{false}, \texttt{true}\}$, with $0 = \texttt{false}$ and $1 = \texttt{true}$.

**Definition 1.6.** In the setting of Definition II.1.4 we define $\mathrm{Serialize}_A$ to be an arbitrary, but fixed section of $\mathrm{Interpret}_A$, i.e., $\mathrm{Interpret}_A \circ \mathrm{Serialize}_A = \mathrm{id}_A$.

While $\mathrm{Interpret}_A$ is seen as the interpretation of computer data in the mathematical context, $\mathrm{Serialize}_A$ can be seen as the "constructor". As for $\mathrm{Interpret}_A$, the notion of computability does not apply to $\mathrm{Serialize}_A$. Part of the application of $\mathrm{Serialize}_A$ to elements of $A$ is usually done by hand, e.g., rewriting matrices as a list of lists which form a valid input for the computer. Mapping such input data into memory, e.g., converting this list of lists to a single number, is then done by the computer itself.

After having defined data structures for sets, we will now define when a function between two realized sets is computable.

**Definition 1.7** (Computable function)**.** Let $A, B$ be sets with realizations $\mathfrak{A}, \mathfrak{B} \subseteq \mathbb{N}$ and corresponding interpretations $\mathrm{Interpret}_A$ and $\mathrm{Interpret}_B$, $A' \subset A$, $B' \subset B$, $\mathfrak{A}' := \mathrm{Interpret}_A^{-1}(A')$, and $\mathfrak{B}' := \mathrm{Interpret}_B^{-1}(B')$.

We call a map $f : A' \to B'$ **computable by the realizations $\mathfrak{A}$ and $\mathfrak{B}$** if there is a computable function

$$\mathfrak{f} : \mathfrak{A}' \to \mathfrak{B}'$$

such that the following diagram commutes:

$$\begin{tikzcd}
\mathfrak{A}' \arrow[r, two heads, "\mathrm{Interpret}_A"] \arrow[d, "\mathfrak{f}"'] & A' \arrow[d, "f"] \\
\mathfrak{B}' \arrow[r, two heads, "\mathrm{Interpret}_B"'] & B'
\end{tikzcd}$$

The choice of the computable counterpart $\mathfrak{f}$ of $f$ is not unique in general. Furthermore, the subsets $A'$ and $B'$ are not realized sets by themselves, since the membership in the realizations of realized sets is by definition decidable, but we do not require decidability for $\mathfrak{A}'$ and $\mathfrak{B}'$.

## 2. Categories with Hom-setoids

We now define the type of category we are going to work with: *categories with Hom-setoids*. As a reminder we first give the classical definition of a category.

**Definition 2.1** (Category, classical definition)**.** A **(locally small) category $\mathcal{A}$** consists of a class of objects $\mathrm{Obj}_{\mathcal{A}}$ and for each pair $A, B \in \mathrm{Obj}_{\mathcal{A}}$ a set of morphisms $\mathrm{Hom}_{\mathcal{A}}(A, B)$ such that for $A, B, C \in \mathrm{Obj}_{\mathcal{A}}$ there is a map

$$\mathrm{PreCompose} : \mathrm{Hom}_{\mathcal{A}}(A, B) \times \mathrm{Hom}_{\mathcal{A}}(B, C) \to \mathrm{Hom}_{\mathcal{A}}(A, C), \ (\varphi, \psi) \mapsto \varphi\psi$$

for which the following holds:

(1) For $A, B, C, D \in \mathrm{Obj}_{\mathcal{A}}$, $\varphi \in \mathrm{Hom}_{\mathcal{A}}(A, B)$, $\psi \in \mathrm{Hom}_{\mathcal{A}}(B, C)$, and $\omega \in \mathrm{Hom}_{\mathcal{A}}(C, D)$

$$(\varphi\psi)\,\omega = \varphi\,(\psi\omega).$$

(2) For every $A \in \mathrm{Obj}_{\mathcal{A}}$ there is an $\mathrm{id}_A \in \mathrm{Hom}_{\mathcal{A}}(A, A)$ such that for every $B \in \mathrm{Obj}_{\mathcal{A}}$ and every $\varphi \in \mathrm{Hom}_{\mathcal{A}}(A, B)$ and $\psi \in \mathrm{Hom}_{\mathcal{A}}(B, A)$ the equalities $\mathrm{id}_A\varphi = \varphi$ and $\psi\mathrm{id}_A = \psi$ hold. We call $\mathrm{id}_A$ the **identity morphism** of $A$.

For a $\varphi \in \mathrm{Hom}_{\mathcal{A}}(A, B)$ we define

$$A =: \mathrm{Source}(\varphi),$$
$$B =: \mathrm{Range}(\varphi).$$

Furthermore, we denote by

$$\mathrm{Mor}_{\mathcal{A}} := \bigcup_{A, B \in \mathrm{Obj}_{\mathcal{A}}} \mathrm{Hom}_{\mathcal{A}}(A, B)$$

the disjoint union of all morphisms.

For computable categories, we are not going to use the classical definition of categories above, but a generalized one: categories with Hom-setoids[1].

**Definition 2.2** (Category with Hom-setoids)**.** A **(locally small) category (with Hom-setoids)** $\mathcal{A}$ consists of the following data:

(1) A class of objects $\mathrm{Obj}_{\mathcal{A}}$.
(2) For two objects $A, B \in \mathrm{Obj}_{\mathcal{A}}$ there is a set $\mathrm{Hom}_{\mathcal{A}}(A, B)$ with an equivalence relation $\sim_{A,B}$, called **congruence of morphisms**.
(3) For every triple $A, B, C \in \mathrm{Obj}_{\mathcal{A}}$ there is a composition function

$$\mathrm{PreCompose} : \mathrm{Hom}_{\mathcal{A}}(A, B) \times \mathrm{Hom}_{\mathcal{A}}(B, C) \to \mathrm{Hom}_{\mathcal{A}}(A, C)\,,\ (\varphi, \psi) \mapsto \varphi\psi,$$

such that for $\varphi, \varphi' \in \mathrm{Hom}_{\mathcal{A}}(A, B)$ with $\varphi \sim_{A,B} \varphi'$ and $\psi, \psi' \in \mathrm{Hom}_{\mathcal{A}}(B, C)$ with $\psi \sim_{B,C} \psi'$ we have

$$\varphi\psi \sim_{A,C} \varphi'\psi',$$

and for all $A, B, C, D \in \mathrm{Obj}_{\mathcal{A}}$, $\alpha \in \mathrm{Hom}_{\mathcal{A}}(A, B)$, $\beta \in \mathrm{Hom}_{\mathcal{A}}(B, C)$, $\gamma \in \mathrm{Hom}_{\mathcal{A}}(C, D)$ we have

$$((\alpha\beta)\gamma) \sim_{A,D} (\alpha(\beta\gamma)).$$

(4) A function

$$\mathrm{IdentityMorphism} : \mathrm{Obj}_{\mathcal{A}} \to \mathrm{Mor}_{\mathcal{A}},\ A \mapsto \mathrm{id}_A,$$

such that for all $A, B \in \mathrm{Obj}_{\mathcal{A}}$ and $\varphi \in \mathrm{Hom}_{\mathcal{A}}(A, B)$ we have

$$\mathrm{id}_A\varphi \sim_{A,B} \varphi \text{ and } \varphi\mathrm{id}_B \sim_{A,B} \varphi.$$

**Notation.** From now on a category is a category with Hom-setoids as defined in Definition II.2.2. If we want to refer to the classical setting in Definition II.2.1, we denote this category by **classical category**.

## 3. Categories with Hom-setoids vs. classical categories

We explain how categories with Hom-setoids relate to classical categories. We will show that every classical category can naturally be interpreted as a category with Hom-setoids. Afterwards, we will define two ways how a classical category can be equivalent to a category with Hom-setoids: Either by using the natural way of a classical category to be interpreted as category with Hom-setoids, or by using the natural way of a category with Hom-setoids to be interpreted as classical category.

**Theorem 3.1.** *Let $\mathcal{A}$ be a category. Then $\mathcal{A}$ is a category with Hom-setoids by setting for two objects $A, B \in \mathrm{Obj}_{\mathcal{A}}$ and two morphisms $\alpha, \beta \in \mathrm{Hom}_{\mathcal{A}}(A, B)$*

$$\alpha \sim_{A,B} \beta :\Leftrightarrow \alpha = \beta.$$

*We call this the **setoid interpretation** of $\mathcal{A}$.*

**Definition 3.2** (Functor)**.** Let $\mathcal{A}$ and $\mathcal{B}$ be two categories (with Hom-setoids). A **functor** $F : \mathcal{A} \to \mathcal{B}$ consists of the following data:

[1]The necessity of this definition will be explained in Chapter III, in particular in Example III.2.13.

(1) A function
$$F_0 : \mathrm{Obj}_{\mathcal{A}} \to \mathrm{Obj}_{\mathcal{B}}, M \mapsto F_0\left(M\right);$$
(2) For each pair of objects $A, B \in \mathrm{Obj}_{\mathcal{A}}$ there is a function
$$F_{A,B} : \mathrm{Hom}_{\mathcal{A}}\left(A, B\right) \to \mathrm{Hom}_{\mathcal{B}}\left(F_0\left(A\right), F_0\left(B\right)\right), \varphi \mapsto F_{A.B}\left(\varphi\right)$$
such that
  (a) for each object $A \in \mathrm{Obj}_{\mathcal{A}}$,
$$F\left(\mathrm{id}_A\right) \sim \mathrm{id}_{F(A)};$$
  (b) for composable $\varphi, \psi \in \mathrm{Mor}_{\mathcal{A}}$,
$$F\left(\varphi\psi\right) \sim F\left(\varphi\right) F\left(\psi\right).$$

For $M \in \mathrm{Obj}_{\mathcal{A}}$ and $\varphi \in \mathrm{Hom}_{\mathcal{A}}\left(A, B\right)$ we define $F\left(M\right) := F_0\left(M\right)$ and $F\left(\varphi\right) := F_{A,B}\left(\varphi\right)$.

**Definition 3.3.** Let $\mathcal{A}$ and $\mathcal{B}$ be two categories (with Hom-setoids). $\mathcal{A}$ and $\mathcal{B}$ are **equivalent** if there is a functor $F : \mathcal{A} \to \mathcal{B}$ for which the following holds:

(1) For any two objects $A, B \in \mathrm{Obj}_{\mathcal{A}}$ the induced map
$$\mathrm{Hom}_{\mathcal{A}}\left(A, B\right) / \sim_{A,B} \to \mathrm{Hom}_{\mathcal{B}}\left(F\left(A\right), F\left(B\right)\right) / \sim_{F(A),F(B)}$$
is well-defined and bijective.
(2) For every $B \in \mathrm{Obj}_{\mathcal{B}}$ there is an $A \in \mathrm{Obj}_{\mathcal{A}}$ such that $B \cong F\left(A\right)$.

**Proposition 3.4.** *Let $\mathcal{A}$ and $\mathcal{B}$ be two classical categories. Then $\mathcal{A}$ and $\mathcal{B}$ are equivalent as classical categories if and only if they are equivalent as categories with Hom-setoids in the sense of Theorem II.3.1.*

Proof. If the equivalence relation on the Hom sets is just the equality, the Definition II.3.3 is the classical definition of equivalence. The claim follows. □

**Proposition 3.5.** *Let $\mathcal{A}$ be a category (with Hom-setoids). Then we can obtain a classical category $\mathcal{A}'$ with*

*(1)* $\mathrm{Obj}_{\mathcal{A}'} := \mathrm{Obj}_{\mathcal{A}}$ *and*
*(2)* $\mathrm{Hom}_{\mathcal{A}'}\left(A, B\right) := \mathrm{Hom}_{\mathcal{A}}\left(A, B\right) / \sim_{A,B}$.

Proof. Since two morphisms of $\mathcal{A}'$ are equal if and only if they are equivalent in $\mathcal{A}$, the axioms of a classical category are fulfilled. □

**Definition 3.6.** In the setting of Theorem II.3.5, we call $\mathcal{A}'$ the **reduction** of $\mathcal{A}$.

**Theorem 3.7.** *Let $\mathcal{A}$ be a category (with Hom-setoids). Then $\mathcal{A}$ is equivalent to its reduction $\mathcal{A}'$ as a category with Hom-setoids.*

Proof. Let $F : \mathcal{A} \to \mathcal{A}'$ the functor defined by $F\left(A\right) := A$ for all $A \in \mathrm{Obj}_{\mathcal{A}}$ and $F\left(\varphi\right) := \overline{\varphi}$ for all $\varphi \in \mathrm{Mor}_{\mathcal{A}}$. Then $F$ is an equivalence. □

Remark 3.8. If there is a classical category $\mathcal{A}$ and a category (with Hom-setoids) $\mathcal{B}$, such that the reduction $\mathcal{B}'$ of $\mathcal{B}$ is equivalent to $\mathcal{A}$ in the classical sense, the setoid interpretation of $\mathcal{A}$ is equivalent to $\mathcal{B}$.

## 4. Computable categories

We now define a computable category by stating the requirements to the realizations of the set of objects and the set of morphisms. We start by defining the requirements for the data structures of objects and morphisms, i.e., the realizations thereof, and then define which functions we require to be computable in those realizations to call a category computable. From now on, we use `typewriter font` for computable functions and normal font for categorical constructions.

**Definition 4.1** (Realization of a category)**.** Let $\mathcal{A}$ be a category where $\mathrm{Obj} := \mathrm{Obj}_{\mathcal{A}}$ and $\mathrm{Mor} := \mathrm{Mor}_{\mathcal{A}}$ are *sets.* A **realization** $\mathfrak{R}$ of $\mathcal{A}$ consists of

(1) a realization $\mathfrak{Obj}$ for Obj, where we denote the functions as follows:
  (a) the interpretation by InterpretObj,
  (b) the computable equivalence relation on $\mathfrak{Obj}$ induced by the interpretation InterpretObj by IsEqualForObjects,
  (c) and the decidability function $\texttt{IsContained}_{\mathfrak{Obj}}$ by IsWellDefinedForObjects.

(2) a realization $\mathfrak{Mor}$ for Mor, where we denote the functions as follows:
  (a) the interpretation by InterpretMor,
  (b) The computable equivalence relation on $\mathfrak{Mor}$ induced by the interpretation InterpretMor by IsEqualForMorphisms,
  (c) the decidability function $\texttt{IsContained}_{\mathfrak{Mor}}$ by IsWellDefinedForMorphisms.

(3) two computable functions

$$\mathrm{Source} : \mathfrak{Mor} \to \mathfrak{Obj},$$
$$\mathrm{Range} : \mathfrak{Mor} \to \mathfrak{Obj},$$

such that the following diagrams commute:

$$\begin{tikzcd}
\mathfrak{Mor} \arrow[r, "\mathrm{InterpretMor}"] \arrow[d, "\mathrm{Source}"'] & \mathrm{Mor} \arrow[d, "\mathrm{Source}"] \\
\mathfrak{Obj} \arrow[r, "\mathrm{InterpretObj}"'] & \mathrm{Obj}
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathfrak{Mor} \arrow[r, "\mathrm{InterpretMor}"] \arrow[d, "\mathrm{Range}"'] & \mathrm{Mor} \arrow[d, "\mathrm{Range}"] \\
\mathfrak{Obj} \arrow[r, "\mathrm{InterpretObj}"'] & \mathrm{Obj}
\end{tikzcd}$$

(4) a computable function

$$\mathrm{IsCongruentForMorphisms} : \ \mathfrak{Mor} \times \mathfrak{Mor} \to \{\texttt{true}, \ \texttt{false}\}$$

which models the equivalence relation on the homomorphism sets from Definition II.2.2, i.e., for two morphisms $\mathfrak{f}, \mathfrak{g} \in \mathfrak{Mor}$, one has

$$\begin{aligned} &\mathrm{IsCongruentForMorphisms}\,(\mathfrak{f}, \mathfrak{g}) = \texttt{true} \\ &\quad \Leftrightarrow \mathrm{InterpretMor}\,(\mathfrak{f}) \sim \mathrm{InterpretMor}\,(\mathfrak{g})\,. \end{aligned}$$

We say that $\mathcal{A}$ is **realized by** $\mathfrak{R} := (\mathfrak{Obj}, \mathfrak{Mor}, \mathrm{IsCongruentForMorphisms})$. If the realization is clear from the context we just say **realized**.

It is also possible to realize a category by undecidable sets, i.e., where the functions IsWellDefinedForObjects and IsWellDefinedForMorphisms are not computable. We will call such a realized category **undecidable**.

**Definition 4.2** (Computable category)**.** Let $\mathcal{A}$ be a category realized by $\mathfrak{R}$. We call **$\mathcal{A}$ computable (by the realization $\mathfrak{R}$)** if the two functions

$$\text{IdentityMorphism} : \mathrm{Obj}_{\mathcal{A}} \to \mathrm{Mor}_{\mathcal{A}},\ A \mapsto \mathrm{id}_A \text{ and}$$
$$\text{PreCompose} : \bigcup_{A,B,C \in \mathrm{Obj}_{\mathcal{A}}} \mathrm{Hom}_{\mathcal{A}}(A,B) \times \mathrm{Hom}_{\mathcal{A}}(B,C) \to \mathrm{Mor}_{\mathcal{A}}, (\varphi,\psi) \mapsto \varphi\psi$$

are computable by the realizations $\mathfrak{Obj}$ and $\mathfrak{Mor}$ by $\mathfrak{R}$.

So to render a category with a given realization computable, we must provide algorithms that make the functions PreCompose and IdentityMorphism computable.

**Definition 4.3.** Let $\mathcal{A}$ be a realized category. If only IdentityMorphism is computable, we call it a **category with computable identity morphism**. If only the function PreCompose is computable, we call it a **category with computable composition**.

We start the running example for this chapter and illustrate that the category of finite dimensional rational vector spaces is computable in the sense of Definition II.4.2.

**Example 4.4** (Vector spaces)**.** Let $\mathcal{V}$ be the category of isomorphism classes of finite dimensional vector spaces over $\mathbb{Q}$ with $\mathrm{Obj}_{\mathcal{V}} := \mathbb{N}$ and $\mathrm{Hom}_{\mathcal{V}}(m,n) = \mathbb{Q}^{m\times n}$, where the composition of morphisms is just matrix multiplication and the identity morphism of an object $m$ is the $m \times m$ identity matrix.[2] We define two matrices in $\mathbb{Q}^{m\times n}$ to be congruent if they are equal[3].

We give a realization for this category to see that it is indeed computable. Let

$$\text{ToInteger} : \bigcup_{i=0}^{\infty} \mathbb{Q}^i \to \mathbb{N}$$

be a computable, injective function with computable inverse.[4] Using ToInteger, we can establish data structures for the objects and the morphisms in the category. For $\mathrm{Obj}_{\mathcal{V}}$, we will use the integers $\mathbb{N} \subset \mathbb{Q}^1$, which we realize using ToInteger. For a morphism $A \in \mathbb{Q}^{m\times n}$,

[2]The category $\mathcal{V}$ is indeed equivalent to the category of finite vector spaces over $\mathbb{Q}$.

[3]In the computable category described in Chapter III IsCongruentForMorphisms will differ from IsEqualForMorphisms

[4] The function ToInteger can be defined by taking a computable injective function $\varphi : \mathbb{Q} \to \mathbb{N}$, for example

$$\frac{a}{b} \mapsto 2^{\mathrm{sgn}(a)\mathrm{sgn}(b)+1} 3^{|a|} 5^{|b|},\ \gcd(a,b) = 1.$$

Now, the function

$$\mathbb{N}^i \to \mathbb{N},\ (n_1,\dots,n_i) \mapsto \prod_{j=1}^{i} p_i^{n_i},$$

where $p_i$ is the $i$-th prime number together with $\varphi$ provides ToInteger.

we will use a list

$$(m, n, A_{1,1}, \dots, A_{m,n}) =: (m, n, A),$$

which we will model as an integer using ToInteger. The function for IsEqualForObjects is the equality of integers, and the decidability function IsWellDefinedForObjects is the check whether the preimage of an integer under ToInteger is a single integer. The functions for IsEqualForMorphisms and IsCongruentForMorphisms are also just comparison of integers, since the serialization function ToInteger is injective. So two integers in the realization of $\mathrm{Mor}_{\mathcal{V}}$ correspond to the same matrix if and only if there are equal. The function IsWellDefinedForMorphisms checks whether an integer corresponds to a list where the first two entries $a, b$ are non-negative integers and the rest of the list has length $ab$. The algorithm for IdentityMorphism is creating an identity matrix, i.e., for an integer $m$ it creates the list

$$(m, m, 1, \dots).$$

The algorithm for PreCompose is the multiplication of matrices, i.e., for two lists

$$(m, n, A) \text{ and } (n, p, B)$$

it creates the list

$$(m, p, AB).$$

From now on for the rest of this thesis, we will not go back to a single integer representation of computer data. Instead we are going to use data structures every modern programming language has, e.g., integers, floats, rationals, arrays/lists, etc. Giving these data structures an image in the memory of the computer is then done by compilers and interpreters, and not of any further interest for this thesis.

## 5. Decidable properties

Categorical properties of objects and morphisms in categories are often the desired result of a computation. So we define the decidability of a property in a category.

**Definition 5.1.** Let $\mathcal{A}$ be a computable category realized by $\mathfrak{R}$ and P a property, i.e., a mathematical attribute of objects or morphisms in $\mathcal{A}$ that can either be true or false. We say P is **decidable** if for the appropriate $\mathfrak{C} \in \{\mathfrak{Obj}, \mathfrak{Mor}\}$ there is a computable function

$$\mathrm{IsP} : \mathfrak{C} \to \{\texttt{true}, \texttt{false}\}$$

with the following property for all $x \in \mathfrak{C}$:

$$\mathrm{IsP}(x) = \texttt{true} \Leftrightarrow \mathrm{P}(\mathrm{Interpret}(x)).$$

The function IsP should return `true` on serialized objects or morphisms if and only if the property P is fulfilled for their interpreted counterpart. For all other serialized objects or morphisms, it should return false.

Important properties to mention here are whether a morphism is a mono-, epi-, or isomorphisms.

**Definition 5.2.** Let $\mathcal{A}$ be a category, $A, B \in \mathrm{Obj}_{\mathcal{A}}$, and $\varphi : A \to B$.

(1) The morphism $\varphi$ is a **monomorphism** if for any $C \in \mathrm{Obj}_{\mathcal{A}}$ and any two $\psi_1, \psi_2 \in \mathrm{Hom}_{\mathcal{A}}(C, A)$ we have
$$\psi_1\varphi \sim_{C,B} \psi_2\varphi \Rightarrow \psi_1 \sim_{C,A} \psi_2.$$
We then write $\varphi : A \hookrightarrow B$.

(2) The morphism $\varphi$ is an **epimorphism** if for any $C \in \mathrm{Obj}_{\mathcal{A}}$ and any two $\psi_1, \psi_2 \in \mathrm{Hom}_{\mathcal{A}}(B, C)$ we have
$$\varphi\psi_1 \sim_{A,C} \varphi\psi_2 \Rightarrow \psi_1 \sim_{B,C} \psi_2.$$
We then write $\varphi : A \twoheadrightarrow B$.

(3) The morphism $\varphi$ is an **isomorphisms** if there exists a morphism $\varphi^{-1} : B \to A$ such that
$$\mathrm{PreCompose}\left(\varphi, \varphi^{-1}\right) \sim_{A,A} \mathrm{IdentityMorphism}\,(A)$$
$$\mathrm{PreCompose}\left(\varphi^{-1}, \varphi\right) \sim_{B,B} \mathrm{IdentityMorphism}\,(B)\,.$$

**Definition 5.3.** Let $\mathcal{A}$ be a computable category.

(1) $\mathcal{A}$ has **decidable monomorphisms** if the monomorphism property is decidable for morphisms. We call the computable function that decides this property IsMonomorphism.

(2) $\mathcal{A}$ has **decidable epimorphisms** if the epimorphism property is decidable for morphisms. The name of the computable function in this case is IsEpimorphism.

(3) $\mathcal{A}$ has **decidable isomorphisms** if the isomorphism property is decidable for morphisms. The name of the computable function in this case is IsIsomorphism.

**Example 5.4** (II.4.4 cont.)**.** The category of vector spaces $\mathcal{V}$ has decidable monomorphisms, epimorphisms, and isomorphisms. The function `Rank` which returns the rank of a rational matrix is well-known to be computable and can be used to compute all of those properties. Let $\varphi \in \mathrm{Mor}_{\mathcal{V}}$ be the triple $(m, n, M)$.

(1) The function to decide monomorphisms is defined by
$$\mathrm{IsMonomorphism}\,(\varphi) := (\texttt{Rank}\,(M) = m)\,.$$

(2) The function to decide epimorphisms is defined by
$$\mathrm{IsEpimorphism}\,(\varphi) := (\texttt{Rank}\,(M) = n)\,.$$

(3) The function to decide isomorphisms is defined by
$$\mathrm{IsIsomorphism}\,(\varphi) := (m = n \ \wedge \ \texttt{Rank}\,(M) = n)\,.$$

## 6. Preadditive categories

We continue following the hierarchy (preadditive, additive, preabelian, abelian) from [**BLH11**, Appendix A] to define abelian categories. For the corresponding hierarchy of computability notions the disjunctions and existential quantifiers in the definitions of (preadditive, additive, preabelian, abelian) categories need to be turned into algorithms. So we emphasize existential quantifiers and disjunctions in the definitions in this chapter.

**Definition 6.1** (Preadditive category)**.** Let $\mathcal{A}$ be a category. $\mathcal{A}$ is **preadditive** if the following conditions hold:

(1) For $A, B \in \operatorname{Obj}_{\mathcal{A}}$ there *exists* a **zero morphism** $0_{A,B} \in \operatorname{Hom}_{\mathcal{A}}(A, B)$ such that for every $C \in \operatorname{Obj}_{\mathcal{A}}$ and every $\varphi \in \operatorname{Hom}_{\mathcal{A}}(C, A)$ and $\psi \in \operatorname{Hom}_{\mathcal{A}}(B, C)$, $\varphi 0_{A,B} \sim_{C,B} 0_{C,B}$ and $0_{A,B}\psi \sim_{A,C} 0_{A,C}$ holds.
(2) For every $A, B \in \operatorname{Obj}_{\mathcal{A}}$ and every $\varphi, \psi \in \operatorname{Hom}_{\mathcal{A}}(A, B)$ there *exists* a **sum** $\varphi + \psi \in \operatorname{Hom}_{\mathcal{A}}(A, B)$ which is associative, commutative, and distributive with the composition up to congruence.
(3) For every $A, B \in \operatorname{Obj}_{\mathcal{A}}$ and every $\varphi \in \operatorname{Hom}_{\mathcal{A}}(A, B)$ there *exists* the **additive inverse** $-\varphi \in \operatorname{Hom}_{\mathcal{A}}(A, B)$ such that $\varphi + (-\varphi) \sim_{A,B} 0_{A,B}$.

In other words, for every $A, B \in \operatorname{Obj}_{\mathcal{A}}$ the set $\operatorname{Hom}_{\mathcal{A}}(A, B)/\sim_{A,B}$ together with the addition, inversion and the zero morphism is an Abelian group.

**Definition 6.2** (Computable preadditive category)**.** Let $\mathcal{A}$ be a computable category by the realization $\mathfrak{R}$. $\mathcal{A}$ is **computable preadditive** (by $\mathfrak{R}$) if $\mathcal{A}$ is preadditive and

(1) There is a function
$$\text{ZeroMorphism} : \operatorname{Obj}_{\mathcal{A}} \times \operatorname{Obj}_{\mathcal{A}} \to \operatorname{Mor}_{\mathcal{A}},\ (A, B) \mapsto 0_{A,B}$$
which is computable by $\mathfrak{R}$.
(2) There is a function
$$\text{AdditionForMorphisms} : \bigcup_{A,B \in \operatorname{Obj}_{\mathcal{A}}} (\operatorname{Hom}_{\mathcal{A}}(A, B) \times \operatorname{Hom}_{\mathcal{A}}(A, B)) \to \operatorname{Mor}_{\mathcal{A}}, (\varphi, \psi) \mapsto \varphi + \psi$$
which is computable by $\mathfrak{R}$.
(3) There is a function
$$\text{AdditiveInverse} : \operatorname{Mor}_{\mathcal{A}} \to \operatorname{Mor}_{\mathcal{A}},\ \varphi \mapsto -\varphi$$
which is computable by $\mathfrak{R}$.

While the addition of morphisms in the classical sense is parameterized by the source and the range of the two morphisms in the sum, the definition of a computable preadditive category states a single function, having all pairs of summable morphisms as domain. When parameterizing the functions by objects, we would end up with a (possibility infinite) set of functions, which would not be implementable, even if every single function in this set is computable. Therefore we need a single function for the addition. The addition function is also not required to check whether their two input morphisms are summable. If the computable function for addition gets the realization of two morphisms as input which cannot be summed, the result is undefined.

Following the terminology Definition II.4.3 we get the following notation:

**Notation.** A computable preadditive category is a preadditive computable category with **computable zero morphisms**, **computable addition**, and **computable additive inverse**.

**Example 6.3** (II.4.4 cont.)**.** We describe the three functions ZeroMorphism, AdditionForMorphisms, and AdditiveInverse for $\mathcal{V}$. These functions can be sketched as follows: create a zero matrix, add two matrices, and additively invert a matrix. Up to applying the inverse of ToInteger to the arguments and again applying ToInteger to the result of the computation, we can model the functions as follows:

(1) ZeroMorphism takes two integers $m, n$ and returns

$$\left( m, n, \underbrace{0, \ldots, 0}_{mn \text{ times}} \right).$$

(2) AdditionForMorphisms takes two sequences $(m, n, A)$ and $(m', n', B)$, and returns $(m, n, A + B)$, where $A + B$ is the entrywise addition.

(3) AdditiveInverse takes a sequence $(m, n, A)$ and returns $(m, n, -A)$.

Since all of them are computable, $\mathcal{V}$ is computable preadditive.

## 7. Additive categories

Next in the hierarchy of category types from [**BLH11**, Appendix A] is the notion of an additive category. Again we emphasize the existential quantifiers.

**Definition 7.1** (Additive category)**.** Let $\mathcal{A}$ be a preadditive category. $\mathcal{A}$ is additive if the following conditions hold:

(1) There *exists* a unique **zero object** $0 \in \mathrm{Obj}_{\mathcal{A}}$ such that for every $A \in \mathrm{Obj}_{\mathcal{A}}$ the sets $\mathrm{Hom}_{\mathcal{A}}(A, 0)$ and $\mathrm{Hom}_{\mathcal{A}}(0, A)$ *consist* of exactly one element up to congruence.

(2) For every pair of objects $A_1, A_2 \in \mathrm{Obj}_{\mathcal{A}}$ there *exists* a **product object** $A_1 \times A_2$ together with **projections** $\pi_i : A_1 \times A_2 \to A_i$, $i = 1, 2$ such that for all $B \in \mathrm{Obj}_{\mathcal{A}}$ and all pairs of morphisms $\varphi_i \in \mathrm{Hom}_{\mathcal{A}}(B, A_i)$ there *exists* an up to congruence unique **universal morphism** $\{\varphi_1, \varphi_2\} \in \mathrm{Hom}_{\mathcal{A}}(B, A_1 \times A_2)$ which makes the following diagram commute up to congruence:

$$\begin{tikzcd}
 & B \arrow[d, "\{\varphi_1,\varphi_2\}"] \arrow[ddl, bend right, "\varphi_1"'] \arrow[ddr, bend left, "\varphi_2"] & \\
 & A_1 \times A_2 \arrow[dl, "\pi_1"] \arrow[dr, "\pi_2"'] & \\
A_1 & & A_2
\end{tikzcd}$$

(3) For every pair of objects $A_1, A_2 \in \mathrm{Obj}_{\mathcal{A}}$ there *exists* a **coproduct object** $A_1 \amalg A_2$ together with **injections** $\iota_i : A_i \to A_1 \amalg A_2$, $i = 1, 2$ such that for all $B \in \mathrm{Obj}_{\mathcal{A}}$ and all pairs of morphisms $\varphi_i \in \mathrm{Hom}_{\mathcal{A}}(A_i, B)$ there *exists* an up to congruence unique **universal morphism** $\langle \varphi_1, \varphi_2 \rangle \in \mathrm{Hom}_{\mathcal{A}}(A_1 \amalg A_2, B)$ which makes the following diagram commute up to congruence:

$$\begin{tikzcd}
 & B & \\
 & A_1 \amalg A_2 \arrow[u, "\langle \varphi_1, \varphi_2 \rangle"'] & \\
A_1 \arrow[ur, "\iota_1"'] \arrow[uur, bend left, "\varphi_1"] & & A_2 \arrow[ul, "\iota_2"] \arrow[uul, bend right, "\varphi_2"']
\end{tikzcd}$$

We will split up the definition of a computable additive category in several parts corresponding to the granulated notions in Definition II.4.3.

**Definition 7.2** (Computable zero object)**.** Let $\mathcal{A}$ be a computable preadditive category by the realization $\mathfrak{R}$. Then $\mathcal{A}$ has a **computable zero objects** if the two constructions

$$\begin{aligned}
\text{UniversalMorphismIntoZeroObject} : \mathrm{Obj}_{\mathcal{A}} &\to \mathrm{Mor}_{\mathcal{A}},\ A \mapsto 0_{A,0},\\
\text{UniversalMorphismFromZeroObject} : \mathrm{Obj}_{\mathcal{A}} &\to \mathrm{Mor}_{\mathcal{A}},\ A \mapsto 0_{0,A}
\end{aligned}$$

are computable by $\mathfrak{R}$.

**Definition 7.3** (Computable product)**.** Let $\mathcal{A}$ be a computable additive category by the realization $\mathfrak{R}$. Then $\mathcal{A}$ has **computable products** if the three constructions

$$\begin{aligned}
\text{DirectProduct} : &\ \mathrm{Obj}_{\mathcal{A}} \times \mathrm{Obj}_{\mathcal{A}} \to \mathrm{Obj}_{\mathcal{A}},\ (A_1, A_2) \mapsto A_1 \times A_2,\\
\text{ProjectionInFactorOfDirectProduct} : &\ \mathrm{Obj}_{\mathcal{A}} \times \mathrm{Obj}_{\mathcal{A}} \times \{1,2\} \to \mathrm{Mor}_{\mathcal{A}},\ (A_1, A_2, i) \mapsto \pi_i,\\
\text{UniversalMorphismIntoDirectProduct} : &\ \bigcup_{A_1, A_2, B \in \mathrm{Obj}_{\mathcal{A}}} \left(\mathrm{Hom}_{\mathcal{A}}(B, A_1) \times \mathrm{Hom}_{\mathcal{A}}(B, A_2)\right)\\
&\qquad \to \mathrm{Mor}_{\mathcal{A}},\ (\varphi_1, \varphi_2) \mapsto \{\varphi_1, \varphi_2\}
\end{aligned}$$

are computable by $\mathfrak{R}$.

**Definition 7.4** (Computable coproduct)**.** Let $\mathcal{A}$ be a computable additive category by the realization $\mathfrak{R}$. Then $\mathcal{A}$ has **computable coproducts** if the three constructions

$$\begin{aligned}
\text{Coproduct} : &\ \mathrm{Obj}_{\mathcal{A}} \times \mathrm{Obj}_{\mathcal{A}} \to \mathrm{Obj}_{\mathcal{A}},\ (A_1, A_2) \mapsto A_1 \amalg A_2,\\
\text{InjectionOfCofactorOfCoproduct} : &\ \mathrm{Obj}_{\mathcal{A}} \times \mathrm{Obj}_{\mathcal{A}} \times \{1,2\} \to \mathrm{Mor}_{\mathcal{A}}, (A_1, A_2, i) \mapsto \iota_i,\\
\text{UniversalMorphismFromCoproduct} : &\ \bigcup_{A_1, A_2, B \in \mathrm{Obj}_{\mathcal{A}}} \left(\mathrm{Hom}_{\mathcal{A}}(A_1, B) \times \mathrm{Hom}_{\mathcal{A}}(A_2, B)\right)\\
&\qquad \to \mathrm{Mor}_{\mathcal{A}}, (\varphi_1, \varphi_2) \mapsto \langle \varphi_1, \varphi_2 \rangle
\end{aligned}$$

are computable $\mathfrak{R}$.

**Proposition 7.5** ([**ML71**, p. 194])**.** *Let $\mathcal{A}$ be an additive category. Then finite direct products and coproducts are isomorphic.*

**Corollary 7.6.** *Let $\mathcal{A}$ be a computable preadditive category by the realization $\mathfrak{R}$ in which direct product and coproduct are computable. Then $\mathcal{A}$ has* ***computable isomorphisms between products and coproducts****, i.e., the following functions are computable by $\mathfrak{R}$:*

$$\begin{aligned}\text{IsomorphismFromCoproductToDirectProduct} : &\ \text{Obj}_{\mathcal{A}} \times \text{Obj}_{\mathcal{A}} \to \text{Mor}_{\mathcal{A}},\\ &(A_1, A_2) \mapsto (A_1 \amalg A_2 \to A_1 \times A_2)\,,\\ \text{IsomorphismFromDirectProductToCoproduct} : &\ \text{Obj}_{\mathcal{A}} \times \text{Obj}_{\mathcal{A}} \to \text{Mor}_{\mathcal{A}},\\ &(A_1, A_2) \mapsto (A_1 \times A_2 \to A_1 \amalg A_2)\,.\end{aligned}$$

Both functions can be computed using the universal properties of products and coproducts.

**Definition 7.7.** Objects which are simultaneously direct products and coproducts are called **biproducts** or **direct sums**.

**Definition 7.8.** Let $\mathcal{A}$ be a computable preadditive category by realization $\mathfrak{R}$ in which finite direct products and coproducts coincide, i.e., for two $A, B \in \text{Obj}_{\mathcal{A}}$ we have

$$A \oplus B := A \times B = A \amalg B.$$

The corresponding categorical notions for direct sums are:

$$\begin{aligned}\text{DirectProduct} : &\ \text{Obj}_{\mathcal{A}} \times \text{Obj}_{\mathcal{A}} \to \text{Obj}_{\mathcal{A}},\ (A_1, A_2) \mapsto A_1 \oplus A_2,\\ \text{ProjectionInFactorOfDirectSum} : &\ \text{Obj}_{\mathcal{A}} \oplus \text{Obj}_{\mathcal{A}} \oplus \{1,2\} \to \text{Mor}_{\mathcal{A}},\ (A_1, A_2, i) \mapsto \pi_i,\\ \text{UniversalMorphismIntoDirectSum} : &\ \bigcup_{A_1, A_2, B \in \text{Obj}_{\mathcal{A}}} (\text{Hom}_{\mathcal{A}}(B, A_1) \oplus \text{Hom}_{\mathcal{A}}(B, A_2))\\ &\to \text{Mor}_{\mathcal{A}},\ (\varphi_1, \varphi_2) \mapsto \{\varphi_1, \varphi_2\}\,,\\ \text{InjectionOfCofactorOfDirectSum} : &\ \text{Obj}_{\mathcal{A}} \times \text{Obj}_{\mathcal{A}} \times \{1,2\} \to \text{Mor}_{\mathcal{A}}, (A_1, A_2, i) \mapsto \iota_i,\\ \text{UniversalMorphismFromDirectSum} : &\ \bigcup_{A_1, A_2, B \in \text{Obj}_{\mathcal{A}}} (\text{Hom}_{\mathcal{A}}(A_1, B) \times \text{Hom}_{\mathcal{A}}(A_2, B))\\ &\to \text{Mor}_{\mathcal{A}}, (\varphi_1, \varphi_2) \mapsto \langle \varphi_1, \varphi_2 \rangle\,.\end{aligned}$$

We say $\mathcal{A}$ has **computable direct sums**.

A category with computable direct sums has computable direct products and coproducts.

**Definition 7.9** (Computable additive category)**.** Let $\mathcal{A}$ be an additive category which is computable preadditive by the realization $\mathfrak{R}$. Then $\mathcal{A}$ is **computable additive** if it has a computable zero object, computable products, and computable coproducts, all computable by the realization $\mathfrak{R}$.

For additive categories, one can decide whether an object is isomorphic to the zero object.

**Definition 7.10.** Let $\mathcal{A}$ be a additive category with realization $\mathfrak{R}$. An object $A \in \mathcal{A}$ is **zero** if it is isomorphic to $0 \in \mathcal{A}$. The category $\mathcal{A}$ has **decidable zeros** if this property is decidable for the realization $\mathfrak{R}$.

**Lemma 7.11.** *Let $\mathcal{A}$ be computable additive by realization $\mathfrak{R}$. Then $\mathcal{A}$ has decidable zeros.*

PROOF. For $A \in \mathrm{Obj}_{\mathcal{A}}$ we have $A \cong 0$ if and only if $\mathrm{id}_A \sim_{A,A} 0_{A,A}$. Since both $\mathrm{id}_A$ and $0_{A,A}$ are computable by $\mathfrak{R}$, and congruence of morphisms is decidable, $\mathcal{A}$ has decidable zeros. □

**Example 7.12** (II.6.3 cont.)**.** We give algorithms for the zero object and direct sum in $\mathcal{V}$ to show that $\mathcal{V}$ is computable additive. Again, we describe the algorithms for the list data structures of matrices, assuming the function ToInteger is applied a posteriori.

(1) UniversalMorphismFromZeroObject: For a number $n$ return $(0, n)$.
(2) UniversalMorphismIntoZeroObject: For a number $n$ return $(n, 0)$.
(3) DirectSum: For two integers $m$ and $n$ return $m + n$.
(4) ProjectionInFactorOfDirectSum: Let $1_m$ be the $m \times m$ identity matrix and $0_{n,m}$ the $n \times m$ zero matrix. For three parameters $m$, $n$, and $i$ return
$$\left(m+n, m, \begin{pmatrix} 1_m \\ 0_{n,m} \end{pmatrix}\right)$$
if $i = 1$ and
$$\left(m+n, n, \begin{pmatrix} 0_{m,n} \\ 1_n \end{pmatrix}\right)$$
if $i = 2$.
(5) InjectionOfCofactorOfDirectSum: The same function as ProjectionInFactorOfDirectSum, but with the output matrices transposed.
(6) UniversalMorphismIntoDirectSum: Takes two matrices $(m, n, A)$ and $(m, n', B)$ and returns the matrix $(m, n + n', (A, B))$.
(7) UniversalMorphismFromDirectSum: The same function as UniversalMorphismIntoDirectSum, but with the output matrices transposed.

Since all of these functions are computable, $\mathcal{V}$ is computable additive.

REMARK 7.13. In definition of a computable additive category, there is already a redundancy: The function ZeroMorphism can be computed using UniversalMorphismFromZeroObject and UniversalMorphismIntoZeroObject. So when implementing a computable additive category it suffices to give algorithms for UniversalMorphismFromZeroObject and UniversalMorphismIntoZeroObject, and the algorithm for ZeroMorphism can be constructed from these two algorithms. This process is called *derivation of operations* and will be explained in Section VII.7.

## 8. Preabelian categories

We now give the definition of a preabelian category and a computable notion thereof.

**Definition 8.1** (Preabelian category)**.** Let $\mathcal{A}$ be an additive category. $\mathcal{A}$ is **preabelian** if the following conditions hold:

(1) For every morphism $\varphi : A \to B$ there *exists* a **kernel** $\kappa : K \hookrightarrow A$ with $\kappa\varphi \sim_{K,B} 0_{K,B}$ such that for every morphism $\tau : T \to A$ with $\tau\varphi \sim_{T,B} 0_{T,B}$ there *exists* an up to congruence unique **kernel lift** $\tau/\kappa : T \to K$ which makes the following diagram commute up to congruence:

$$\begin{tikzcd}
K \arrow[ddrr, hook, "\kappa"] \arrow[ddrrrr, bend left, "0_{K,B}"] & & & & \\
& & & & \\
& & A \arrow[rr, "\varphi"] & & B \\
& & & & \\
T \arrow[uuuu, dashed, "\tau/\kappa"] \arrow[uurr, "\tau"'] \arrow[uurrrr, bend right, "0_{T,B}"'] & & & &
\end{tikzcd}$$

(2) For every morphism $\varphi : A \to B$ there *exists* a **cokernel** $\epsilon : B \twoheadrightarrow C$ with $\varphi\epsilon \sim_{A,C} 0_{A,C}$ such that for every morphism $\tau : B \to T$ with $\varphi\tau \sim_{A,C} 0_{A,C}$ there *exists* an up to congruence unique **cokernel colift** $\epsilon\backslash\tau : C \to T$ which makes the following diagram commute up to congruence:

$$\begin{tikzcd}
& & & & C \arrow[dddd, dashed, "\epsilon\backslash\tau"] \\
& & & & \\
A \arrow[rr, "\varphi"] \arrow[uurrrr, bend left, "0_{A,C}"] \arrow[ddrrrr, bend right, "0_{A,C}"'] & & B \arrow[uurr, "\epsilon"] \arrow[ddrr, "\tau"'] & & \\
& & & & \\
& & & & T
\end{tikzcd}$$

**Definition 8.2.** Let $\mathcal{A}$ be a computable category by the realization $\mathfrak{R}$. Then $\mathcal{A}$ has **computable kernels** if the three functions

$$\begin{aligned}
\mathrm{KernelObject} &: \mathrm{Mor}_{\mathcal{A}} \to \mathrm{Obj}_{\mathcal{A}},\ \varphi \mapsto K,\\
\mathrm{KernelEmbedding} &: \mathrm{Mor}_{\mathcal{A}} \to \mathrm{Mor}_{\mathcal{A}},\ \varphi \mapsto \kappa,\\
\mathrm{KernelLift} &: \mathcal{M} \to \mathrm{Mor}_{\mathcal{A}},\ (\varphi, \tau) \mapsto \tau/\kappa
\end{aligned}$$

are computable by the realization $\mathfrak{R}$, where

$$\mathcal{M} := \left\{ (\varphi, \tau) \in \bigcup_{T,A,B \in \mathrm{Obj}_{\mathcal{A}}} \left(\mathrm{Hom}_{\mathcal{A}}(A, B) \times \mathrm{Hom}_{\mathcal{A}}(T, A)\right) \;\middle|\; \tau\varphi \sim 0 \right\}.$$

Remark 8.3. To treat the kernel of a morphism in a computable category algorithmically, it is not sufficient to have an algorithm to compute the kernel object, but also algorithms to compute the kernel embedding and the kernel lift are necessary. So for a proper algorithmical handling of the kernel not only a computable function KernelObject is needed, but also a computable functions for KernelEmbedding and KernelLift. On the other hand we have

$$\text{KernelObject} = \text{Source} \circ \text{KernelEmbedding},$$

so the algorithm for the kernel object can be "derived" from the algorithm for kernel embedding.

In Example II.8.6 we see that it can be more effective not to use this "derivation" (cf. Section VII.7) to implement the algorithm for KernelObject.

**Definition 8.4.** Let $\mathcal{A}$ be a computable category by the realization $\mathfrak{R}$. Then $\mathcal{A}$ has **computable cokernels** if the three functions

$$\begin{aligned} \text{CokernelObject} &: \text{Mor}_{\mathcal{A}} \to \text{Obj}_{\mathcal{A}},\ \varphi \mapsto C, \\ \text{CokernelProjection} &: \text{Mor}_{\mathcal{A}} \to \text{Mor}_{\mathcal{A}},\ \varphi \mapsto \epsilon, \\ \text{CokernelColift} &: \mathcal{M} \to \text{Mor}_{\mathcal{A}},\ (\varphi, \tau) \mapsto \epsilon \backslash \tau \end{aligned}$$

are computable by the realization $\mathfrak{R}$, where

$$\mathcal{M} := \left\{ (\varphi, \tau) \in \bigcup_{T, A, B \in \text{Obj}_{\mathcal{A}}} \left(\text{Hom}_{\mathcal{A}}(A, B) \times \text{Hom}_{\mathcal{A}}(B, T)\right) \;\middle|\; \varphi\tau \sim 0 \right\}.$$

**Definition 8.5** (Computable preabelian categories)**.** Let $\mathcal{A}$ be a computable additive category by the realization $\mathfrak{R}$, which is preabelian. Then $\mathcal{A}$ is **computable preabelian** if it has computable kernels and cokernels by the realization $\mathfrak{R}$.

**Example 8.6** (II.7.12 cont)**.** We give the algorithms to make $\mathcal{V}$ a computable preabelian category. All algorithms are based on Gaussian elimination, which is well-known to be computable.

(1) KernelObject: Gets a matrix $A \in \mathbb{Q}^{m \times n}$, and returns $k := m - \texttt{Rank}(A)$.
(2) KernelEmbedding: Let $A \in \mathbb{Q}^{m \times n}$ . There is a computable function

$$\texttt{BasisOfSyzygiesOfRows}(A)$$

which computes a matrix $T \in \mathbb{Q}^{k \times m}$ such that $T$ is in Gaussian normal form, i.e., the rows form a basis, $TA = 0$, and for every $T' \in \mathbb{Q}^{k' \times m}$ with $T'A = 0$ the equation $XT = T'$ is solvable. KernelEmbedding is then `BasisOfSyzygiesOfRows`[5].
(3) KernelLift: Let $A \in \mathbb{Q}^{m \times n}$ and $B \in \mathbb{Q}^{k \times n}$ be matrices. Define $B' \in \mathbb{Q}^{k \times n}$ such that $B - B' = XA$ is solvable and the $i$-th column $B'_{-,i} = 0$ if and only if $xA = B_{-,i}$ is solvable. In particular, $XA = B$ is solvable if and only if $B' = 0$. Then we define

$$\texttt{RightDivide}(B, A) := X.$$

[5] A more usual name for this function could be `Nullspace` or `LeftNullspace`, but we want to emphasize the relations of this functions to the functions defined in III.2.4.

We can then set

$$\text{KernelLift}\,(B, A) := \texttt{RightDivide}\,(\texttt{BasisOfSyzygiesOfRows}\,(B)\,, A)\,.$$

(4) CokernelObject: Gets a matrix $A \in \mathbb{Q}^{m \times n}$, and returns $\ell := n - \texttt{Rank}\,(A)$.
(5) CokernelProjection: Let $A \in \mathbb{Q}^{m \times n}$ . There is a computable function

$$\texttt{BasisOfSyzygiesOfColumns}\,(A)$$

which computes a matrix $T \in \mathbb{Q}^{n \times \ell}$ such that $T$ is in Gaussian normal form, i.e., the columns form a basis, $AT = 0$, and for every $T' \in \mathbb{Q}^{n \times \ell'}$ with $AT' = 0$ the equation $TX = T'$ is solvable. CokernelProjection is then `BasisOfSyzygiesOfColumns`.
(6) CokernelColift: Let $A \in \mathbb{Q}^{n \times m}$ and $B \in \mathbb{Q}^{n \times \ell}$ be matrices. Define $B' \in \mathbb{Q}^{n \times \ell}$ such that $B - B' = AX$ is solvable and the $i$-th row $B'_{i,-} = 0$ if and only if $Ax = B_{i,-}$ is solvable. In particular, $AX = B$ is solvable if and only if $B' = 0$. Then we define

$$\texttt{LeftDivide}\,(A, B) := X.$$

We can then set

$$\text{CokernelColift}\,(B, A) := \texttt{LeftDivide}\,(A, \texttt{BasisOfSyzygiesOfColumns}\,(B))\,.$$

All of those functions are computable, and therefore $\mathcal{V}$ is computable preabelian. We will reencounter the functions `BasisOfSyzygiesOfRows`, `BasisOfSyzygiesOfColumns`, `LeftDivide`, `RightDivide`, `DecideZeroRows`, and `DecideZeroColumns`. in Definition III.2.4.

**Proposition 8.7.** *Let $\mathcal{A}$ be a computable preabelian category. Then $\mathcal{A}$ has decidable monomorphisms and epimorphisms.*

Proof. Let $\varphi : A \to B$ be a morphism in $\mathcal{A}$. We will show that $\varphi$ is a monomorphism if and only if the kernel object of $\varphi$ is isomorphic to the zero object. Let

$$(\kappa : K \to A) := \text{KernelEmbedding}\,(\varphi)\,.$$

Suppose $\varphi$ is a monomorphism. Then we have

$$\kappa\varphi \sim 0_{K,A}\varphi$$

and therefore

$$\kappa \sim 0_{K,A}.$$

Since the kernel lift is unique up to congruence for every test morphism this means that $K \cong 0$.

Now suppose $K \cong 0$ and let $\psi_1, \psi_2 : T \to A$ with $\psi_1\varphi \sim \psi_2\varphi$. By the additivity this means that $(\psi_1 - \psi_2)\,\varphi \sim 0_{T,B}$ therefore

$$\psi := \psi_1 - \psi_2$$

is a test morphism for the kernel. But, since $K \cong 0$, we have $\psi \sim 0_{T,A}$, and therefore $\psi_1 \sim \psi_2$.

The dual fact that a morphism is epi if and only if its cokernel is zero is analog.

Indeed, we can state the constructions as follows: Let $\varphi \in \mathrm{Mor}_{\mathcal{A}}$. Then

$$\mathrm{IsMonomorphism}\,(\varphi) := \mathrm{IsZero}\,(\mathrm{KernelObject}\,(\varphi))$$
$$\mathrm{IsEpimorphism}\,(\varphi) := \mathrm{IsZero}\,(\mathrm{CokernelObject}\,(\varphi))\,.$$

□

## 9. Abelian categories

The final type of category in the hierarchy in [**BLH11**, Appendix A] are abelian categories. We start again by defining an abelian category, then we give a computable notion thereof.

**Definition 9.1.** Let $\mathcal{A}$ be a preabelian category. Then $\mathcal{A}$ is **abelian** if the following conditions hold:

(1) Any monomorphism is the kernel of its cokernel: Let $\kappa : K \hookrightarrow A$ be a mono, $\epsilon : A \to C$ its cokernel. Then for any $\tau : T \to A$ with $\tau\epsilon \sim_{T,C} 0_{T,C}$ there *exists* an up to congruence unique lift $\tau/\kappa : T \to K$ with $(\tau/\kappa)\,\kappa \sim_{T,A} \tau$.
(2) Any epimorphism is the cokernel of its kernel: Let $\epsilon : A \twoheadrightarrow C$ be an epi, $\kappa : K \to A$ its kernel. Then for any $\tau : A \to T$ with $\kappa\tau \sim_{K,T} 0_{K,T}$ there *exists* an up to congruence unique colift $\epsilon\backslash\tau : C \to T$ with $\epsilon\,(\epsilon\backslash\tau) \sim_{A,T} \tau$.

**Definition 9.2** (Computable abelian categories)**.** Let $\mathcal{A}$ be a computable preabelian category by realization $\mathfrak{R}$ which is abelian. Then $\mathcal{A}$ is **computable abelian** if the following functions

$$\mathrm{LiftAlongMonomorphism} : \mathcal{M} \to \mathrm{Mor}_{\mathcal{A}},\ (\tau,\kappa) \mapsto \tau/\kappa,$$
$$\mathrm{ColiftAlongEpimorphism} : \mathcal{N} \to \mathrm{Mor}_{\mathcal{A}},\ (\epsilon,\tau) \mapsto \epsilon\backslash\tau$$

are computable by $\mathfrak{R}$, where

$$\mathcal{M} := \Big\{ (\tau,\kappa) \in \bigcup_{T,A,K\in\mathrm{Obj}_{\mathcal{A}}} (\mathrm{Hom}_{\mathcal{A}}(T,A) \times \mathrm{Hom}_{\mathcal{A}}(K,A)) \ \Big| \ \kappa \text{ mono},\ \tau\,\mathrm{CokernelProjection}\,(\kappa) \sim 0 \Big\},$$

$$\mathcal{N} := \Big\{ (\epsilon,\tau) \in \bigcup_{T,A,C\in\mathrm{Obj}_{\mathcal{A}}} (\mathrm{Hom}_{\mathcal{A}}(A,C) \times \mathrm{Hom}_{\mathcal{A}}(A,T)) \ \Big| \ \epsilon \text{ epi},\ \mathrm{KernelEmbedding}\,(\epsilon)\,\tau \sim 0 \Big\}.$$

**Corollary 9.3.** *Let $\mathcal{A}$ be a computable abelian category. Then $\mathcal{A}$ has decidable isomorphisms.*

Proof. For abelian categories a morphism is an isomorphism if it is both a monomorphism and an epimorphism, and those properties are decidable by Proposition II.8.7. □

We finish the definitions of computable categories by showing that the category of rational finite dimensional vector spaces is computable abelian.

**Example 9.4** (II.8.6 cont)**.** We show that $\mathcal{V}$ is computable abelian. Again, all computations depend on the Gaussian elimination.

(1) LiftAlongMonomorphism: Takes two matrices $K$ and $T$, returns

$$\texttt{RightDivide}\,(K, T)\,.$$

(2) ColiftAlongEpimorphism: Gets two matrices $C$ and $T$, returns

$$\texttt{LeftDivide}\,(T, C)\,.$$

Since those two are computable, $\mathcal{V}$ is computable abelian by the serialization given in Example II.4.4.

## 10. Categorical notions

We will establish notions for sub- and quotient (or factor) objects in categories, as well as the notion of the image of a morphism. All the definitions in this section are only of minor importance to this thesis. They are used to make definitions and proofs in the following chapters look more natural.

**Definition 10.1.** Let $\mathcal{A}$ be a category, and $A, B, C \in \mathrm{Obj}_{\mathcal{A}}$.

(1) Let $\varphi \in \mathrm{Hom}_{\mathcal{A}}(B, A)$ and $\psi \in \mathrm{Hom}_{\mathcal{A}}(C, A)$. We say $\varphi$ **dominates** $\psi$ if there is a morphism $\tau \in \mathrm{Hom}_{\mathcal{A}}(C, B)$ such that

$$\tau\varphi \sim \psi.$$

(2) A **subobject** of $A$ is a class of mutually dominating monomorphisms with range $A$. If $B \hookrightarrow A$ is an element of this class, we write $B$ for the subobject and $A \subseteq B$.
(3) Let $\varphi \in \mathrm{Hom}_{\mathcal{A}}(A, B)$ and $\psi \in \mathrm{Hom}_{\mathcal{A}}(A, C)$. We say $\varphi$ **codominates** $\psi$ if there is a morphism $\tau \in \mathrm{Hom}_{\mathcal{A}}(B, C)$ such that

$$\varphi\tau = \psi.$$

(4) A **factor object** or **quotient object** of $A$ is a class of mutually codominating epimorphisms with source $A$. If $\pi : A \twoheadrightarrow B$ is an element of this class, we identify the factor object with $B$ and write

$$B =: A/\mathrm{KernelObject}\,(\pi)\,.$$

Every time we write a subobject or a factor object we use it as a *placeholder for its embedding or projection*.

**Definition 10.2.** Let $\mathcal{A}$ be a category and $\varphi : A \to B \in \mathrm{Mor}_{\mathcal{A}}$.

(1) An **image** of $A$ under $\varphi$ is a monomorphism $\iota : I \hookrightarrow B$ such that there is a (necessarily unique) epimorphism $\varphi' : A \twoheadrightarrow I$ with

$$\varphi'\iota \sim \varphi.$$

We define

$$\operatorname{ImageObject}(\varphi) := I \text{ and } \operatorname{ImageEmbedding}(\varphi) := \iota.$$

(2) The **restriction** of $\varphi$ to a subobject $A'$ with embedding $\iota : A' \hookrightarrow A$ is the morphism $\iota\varphi$. We write $\varphi|_{A'}$ for the restriction and $\varphi(A')$ for its image.

**Definition 10.3.** Let $\mathcal{A}$ be an abelian category, $\varphi : A \to B \in \operatorname{Mor}_{\mathcal{A}}$, and $\iota : B' \hookrightarrow B$.

(1) If $\iota$ dominates $\operatorname{ImageEmbedding}(\varphi)$ we call the lift

$$\varphi/\iota : A \to B'$$

the **coastriction** of $\varphi$ with $\iota$.

(2) Let $\gamma : A' \hookrightarrow A$ a monomorphism such that $\iota$ dominates $\gamma\varphi$. Then we call

$$(\gamma\varphi)/\iota : A' \to B'$$

the **restriction-coastriction** of $\varphi$ with $\gamma$ and $\iota$.

CHAPTER III

# Implementation of graded modules

As a first involved example of a computable category we present the category of graded module presentations over a graded ring. This category is equivalent to the category of finitely presented graded modules over a graded ring and is computable abelian if certain conditions on the ring a met.

We start by defining a graded ring, graded modules, and morphisms between graded modules. Then we define the category of graded module presentations over a graded ring and state their equivalence to the category of f.p. graded modules. Afterwards, we define the conditions on the graded ring to be computable, and show that the category of graded module presentations over such a computable ring is computable abelian.

The category of graded module presentations is also the motivation for working with categories with Hom-setoids instead of classical categories. In the graded module presentations category, the notions of equality and congruence of morphisms will be distinct, and we will emphasize the reason for this distinction.

## 1. The category of graded module presentations

We now give the definitions of a graded rings, graded modules, and their computable versions.

Our definition of the grading will be general, i.e., the rings will be graded by a finitely presented abelian group. As stated in the Introduction I, we want to model the graded modules over Cox rings of toric varieties, and for each finitely presented abelian group $G$ a toric variety can be constructed whose Cox Ring is graded by the group $G$.

**Definition 1.1** (Graded ring)**.** Let $G$ be a finitely presented additively written abelian group and $S$ a ring. We call $S$ **graded by $G$ or $G$-graded** if there is a subring $S_0 \subset S$ and for every $g \in G$ an $S_0$-submodule $S_g \subset S$ such that

$$S = \bigoplus_{g \in G} S_g$$

with $g, h \in G$ implies $S_g S_h \subseteq S_{g+h}$ and the set

$$\{g \in G \mid S_g \neq \{0\}\}$$

generates $G$ as an abelian group.

An element $s_g \in S_g$ is called **homogeneous of degree** $g$ and $G$ is called the **degree group** of $R$.

**Definition 1.2** (Graded module)**.** Let $S$ be a $G$-graded ring. An $S$-(left)-module $M$ is **graded by $G$ or $G$-graded** if for every $g \in G$ there is a $S_0$ submodule $M_g \subset M$ such that

$$M = \bigoplus_{g \in G} M_g$$

and for every $g, h \in G$ we have $S_g M_h \subseteq M_{g+h}$.

**Definition 1.3** (Graded morphisms)**.** Let $S$ be a $G$-graded ring and $M, N$ $G$-graded $S$-modules. An $S$-module homomorphism

$$\varphi : M \to N$$

is called **graded** if there is an $h \in G$ such that for each $g \in G$ the restriction-coastriction[1]

$$\varphi_g : M_g \to N_{g+h}, \; x \mapsto \varphi(x)$$

is an $S_0$-module homomorphism. We call $h$ the **degree** of $\varphi$.

For a $g \in G$ we define

$$\mathrm{Hom}_g(M, N) := \{\varphi : M \to N \mid \deg(\varphi) = g\},$$

and set

$$\mathcal{H}\mathrm{om}(M, N) := \bigoplus_{g \in G} \mathrm{Hom}_g(M, N).$$

A **morphism of $G$-graded $S$-modules** is a graded morphism of degree 0.

Remark 1.4. Let $S$ be a $G$ graded ring and $M, N$ $G$-graded $S$-modules. Then the set $\mathcal{H}\mathrm{om}(M, N)$ is a $G$-graded $S$-module.

**Definition 1.5.** Let $S$ be a $G$-graded ring, $M \in S^{m \times k}$, and $N \in S^{n \times k}$ matrices with homogeneous entries. We say $N$ **row dominates** $M$ and write $N \geqslant_{\mathrm{row}} M$ if there is an matrix with homogeneous entries $X \in S^{m \times n}$ such that $XN = M$.

We now define the model of f.p. graded modules we are going to work with: the category of graded presentations.

**Definition 1.6** (Category of finitely presented $G$-graded $S$-modules)**.** Let $S$ be a graded ring with degree group $G$. The category $S$-grpres of graded left presentations over $S$ is defined as follows:

(1) The class $\mathrm{Obj}_{S\text{-grpres}}$ is the set of all tuples $M := (M', \omega)$, where $M' \in S^{m \times g_M}$, $m, g_M \in \mathbb{Z}_{\geqslant 0}$, is an matrix with homogeneous entries and $\omega \in G^{g_M}$ such that for all $k = 1, \ldots, m$ we have

$$\deg\left(M'_{k,1}\right) - \omega_1 = \cdots = \deg\left(M'_{k,g_M}\right) - \omega_{g_M}$$

for $i = 1, \ldots, g_M$ with $M'_{k,i} \neq 0$.

[1]For a definition, see II.10.3.

We call $M'$ the **relation matrix** of $M$, $g_M$ the **number of generators** of $M$, and $\omega$ the **generator degrees** of $M$. We define:

$$\texttt{UnderlyingMatrix}(M) := M',$$
$$\texttt{NumberOfGenerators}(M) := g_M,$$
$$\texttt{GeneratorDegrees}(M) := \omega.$$

(2) For two objects $M, N \in \mathrm{Obj}_{S\text{-grpres}}$ the set $\mathrm{Hom}_{S\text{-grpres}}(M, N)$ is the set of all triples $(M, A, N)$ such that
  (a) $A$ is a homogeneous $g_M \times g_N$ matrix.
  (b) For $\omega_M := \texttt{GeneratorDegrees}(M)$ and $\omega_N := \texttt{GeneratorDegrees}(N)$ we have
  $$\omega_{M,k} + \deg(A_{k,j}) = \omega_{N,j}$$
  for all $k = 1, \dots, g_M$ and $j = 1, \dots, g_N$ with $A_{k,j} \neq 0$.
  (c) $N'$ row dominates $M'A$, i.e., $XN' = M'A$ is solvable for $X$.

Two morphisms $(M_1, A_1, N_1)$ and $(M_2, A_2, N_2)$ are **equal** if
  (a) $M_1 = M_2$,
  (b) $A_1 = A_2$,
  (c) $N_1 = N_2$.

They are **congruent** if
  (a) $M_1 = M_2$,
  (b) $N_1 = N_2$,
  (c) there exist a matrix with homogeneous entries $Y$ such that $YN_1' = A_1 - A_2$.

For $S$-grpres the identity morphism and composition are defined as follows:
(1) $\mathrm{IdentityMorphism}(M) := (M, 1_{g_M}, M)$,
(2) $\mathrm{PreCompose}((M, A, N), (N, B, L)) := (M, AB, L)$,

where $1_{g_M}$ denotes the $g_M \times g_M$ identity matrix over $S$.

Remark 1.7. As in Definition II.2.2, we have different notions for equality and congruence on the Hom-setoids.

**Proposition 1.8.** *Let $S$ be a $G$-graded ring. Then $S$-grpres is a category.*

Proof. We need to show that composed morphisms are again morphisms and the defined operations for composition and identity morphism respect the congruence:

(1) We show that the composition of two morphisms is well-defined. Let $\alpha := (M, A, N)$ and $\beta := (N, B, R)$ be morphisms. Then there is a matrix $X$ with $XN' = M'A$ and a matrix $Y$ with $YR' = N'B$. So we have
$$M'AB = XN'B = XYR'$$
and therefore the equation $M'AB = ZR'$ is solvable for $Z$.
(2) We show the well-definedness of the identity morphism. Let $M$ be an object and $\alpha := (M, A, N)$ a morphism. We have
$$\mathrm{id}_A = (A, 1_{g_A}, A) = \mathrm{IdentityMorphism}(A).$$

So $\alpha \mathrm{id}_A = \alpha$ and therefore $\alpha \mathrm{id}_A \sim \alpha$. The same is true for $\mathrm{id}_A \alpha$. opposite side.

(3) We show that the composition respects the congruence. Let $M, N, R$ be objects and $\alpha := (M, A, N)$, $\alpha' := (M, A', N)$, $\beta := (N, B, R)$, $\beta' := (N, B', R)$ morphisms with

$$\alpha \sim \alpha' \text{ and } \beta \sim \beta',$$

i.e., there are matrices $X_{A,A'}$ and $Y_{B,B'}$ with $A - A' = X_{A,A'}N'$ and $B - B' = Y_{B,B'}R'$. Then we have

$$\begin{aligned} AB - A'B' &= (A' + Y_{A,A'}N')(B' + Y_{B,B'}R') - A'B' \\ &= Y_{A,A'}N'B' + A'Y_{B,B'}R' + Y_{A,A'}N'Y_{B,B'}R' \\ &= (Y_{A,A'}X + A'Y_{B,B'} + Y_{A,A'}N'Y_{B,B'})\,R' \end{aligned}$$

with $XR' = N'B'$. So we get

$$\beta\alpha \sim \beta'\alpha'. \qquad \square$$

**Proposition 1.9.** *Let $S$ be a $G$-graded ring and $(M, A, N) \in \mathrm{Mor}_{S\text{-grpres}}$. Then the degrees of the non-zero entries of $A$ are determined by the generator degrees of $M$ and $N$.*

Proof. Let $\omega_M :=$ `GeneratorDegrees`$(M)$ and $\omega_N :=$ `GeneratorDegrees`$(N)$. If $A_{i,j} \neq 0$, we have

$$\deg(A_{i,j}) = \omega_{M,i} - \omega_{N,j}. \qquad \square$$

**Theorem 1.10** ([BLH11, 3.1] using II.3.8)**.** *The category $S$-grpres is equivalent to the category of finitely presented graded modules over $S$, $S$-grmod, where the congruence of morphisms is just the equality.*

## 2. Computability of graded module presentations

We are now going to investigate which properties of the ring $S$ and the grading group $G$ need to hold such that $S$-grpres is computable abelian.

**Definition 2.1** (Realized ring)**.** Let $S$ be a ring. A **realization of** $S$ is a realization $\mathfrak{R}$ of the underlying set of $S$ such that the following functions are computable by the realization $\mathfrak{R}$:

(1) The addition
$$+ : S \times S \to S,\ (s_1, s_2) \mapsto s_1 + s_2,$$
(2) the inversion
$$- : S \to S,\ s \mapsto -s,$$
(3) the multiplication
$$\cdot : S \times S \to S,\ (s_1, s_2) \mapsto s_1 s_2,$$
(4) the zero element
$$0 : \{*\} \to S, * \mapsto 0,$$
(5) the one element
$$1 : \{*\} \to S, * \mapsto 1.$$

**Definition 2.2** (Realized group)**.** Let $G$ be an additively written group. A **realization of** $G$ is a realization $\mathfrak{R}$ of the underlying set of $G$ such that the following functions are computable by the realization $\mathfrak{R}$:

(1) The addition
$$+ : G \times G \to G,\ (g_1, g_2) \to g_1 + g_2,$$
(2) the inversion
$$^{-1} : G \to G,\ g \mapsto g^{-1},$$
(3) the neutral element
$$0 : \{*\} \to G,\ * \mapsto 0.$$

**Definition 2.3** (Realized $G$-graded ring)**.** Let $S$ be a $G$-graded ring. A **realization of** $S$ **as** $G$**-graded ring** consists of realizations $\mathfrak{R}_S$ of $S$ as a ring and $\mathfrak{R}_G$ of $G$, such that the function
$$\deg : \left(\bigcup_{g\in G} S_g\right) - \{0\} \to G,\ s \mapsto h \text{ if } s \in S_h$$
is computable for the realizations $\mathfrak{R}_S$ and $\mathfrak{R}_G$.

If a realization exists, we call $S$ **realized**.

**Definition 2.4** (Computable ring)**.** Let $S$ be a $G$-graded ring with realization $\mathfrak{R} := (\mathfrak{R}_S, \mathfrak{R}_G)$. We call $S$ **(left) computable** if the following functions are computable using $\mathfrak{R}_S$.

(1) For two matrices with homogeneous entries $A \in S^{m\times n}$ and $B \in S^{k\times n}$ there is an algorithm
$$\texttt{DecideZeroRows}\,(B, A)$$
which returns a matrix $B' \in S^{k\times n}$ such that $B - B' = XA$ is solvable and the $i$-th column $B'_{-,i} = 0$ if and only if $xA = B_{-,i}$ is solvable. In particular, $B = XA$ is solvable if and only if $B' = 0$.

(2) For two matrices with homogeneous entries $A \in S^{m\times n}, B \in S^{m\times k}$ over $S$ there is an algorithm
$$\texttt{DecideZeroRowsEffectively}\,(B, A)$$
which computes a matrix $X$ such that $B - XA = B'$ with
$$B' := \texttt{DecideZeroRows}\,(B, A)\,.$$
Furthermore, if $B' = 0$, we define
$$X := \texttt{RightDivide}\,(B, A)\,.$$

(3) For a matrix with homogeneous entries $A \in S^{m\times n}$ there is an algorithm
$$\texttt{SyzygiesOfRows}\,(A)$$
which computes the homogeneous syzygies of the rows of $A$, i.e., a matrix (with homogeneous entries) $T \in S^{k\times n}$ such that $TA = 0$ and for every $T' \in S^{k'\times n}$ with $T'A = 0$ the equation $XT = T'$ is solvable.

A right computable ring can be defined by replacing left with right, rows with columns, and the side of the computed matrices in Definition III.2.4. For a commutative ring, the notions left computable and right computable coincide. So we call a left computable commutative ring **computable**.

**Proposition 2.5.** *Let $S := K[x_1, \ldots, x_n]$ be a $G$-graded polynomial ring with a term order $\leqslant$ and $I := \langle f_1, \ldots, f_n \rangle$ with $f_i$ homogeneous for all $i$. Then the reduced Gröbner basis $\{g_1, \ldots, g_m\}$ of $I$ with respect to $\leqslant$ is homogeneous.*

Proof. We show that the reduced Gröbner basis computed with Buchberger's algorithm is homogeneous. It suffices to show that the S-polynomial of two homogeneous polynomials is homogeneous. Let

$$f := \sum_{i \in \mathbb{N}^n} f_i x^i$$

and

$$g := \sum_{i \in \mathbb{N}^n} g_i x^i$$

be two homogeneous polynomials in $S$, and $a := \mathrm{Lt}_{\leqslant}(f)$ and $b := \mathrm{Lt}_{\leqslant}(g)$, where $\mathrm{Lt}_{\leqslant}$ denotes the leading term with respect to $\leqslant$. Define $m := \mathrm{lcm}(a, b)$ and $m_f := m/a$ and $m_g := m/b$. Then we have

$$\mathrm{Lt}_{\leqslant}(m_f f) = \mathrm{Lt}_{\leqslant}(m_g g) = m$$

and therefore $m_f f$ and $m_g g$ are homogeneous of degree $\deg(m)$, since $m_f$ and $m_g$ are terms and $f$ and $g$ are homogeneous, and $m_f f$ and $m_g g$ contain the term $m$. But then the S-polynomial of $f$ and $g$,

$$m_f f - m_g g$$

is a homogeneous polynomial. From the same argument it follows that the autoreduction preserves the homogeneity. □

**Definition 2.6.** Let $K$ be a field. A **realization of** $K$ is a realization $\mathfrak{R}$ of $K$ as a ring such that the inversion

$$^{-1} : K - \{0\} \to K - \{0\}, x \mapsto x^{-1}$$

is computable by $\mathfrak{R}$.

**Proposition 2.7.** *Let $n \in \mathbb{N}$, $G$ a group realized by $R_G$ and $K$ a field realized by $R_K$. Furthermore, let $S := K[x_1, \ldots, x_n]$. Then $S$ has a realization $R_S$ and if the grading function is computable for $R_S$ and $R_G$, $S$ is computable $G$-graded.*

Proof. If $K$ is realized, a polynomial in $S$ can be realized as list of terms, consisting of a factor in $K$ and exponents. Multiplication and addition are then carried out in $K$ and $\mathbb{N}$, so $S$ is realizable with a realization $R_S$. If the grading is computable, $S$ is then also realized as a $G$-graded ring. Since polynomials in $S$ are just lists of terms, one can decide leading terms, and therefore can have a computable term ordering on $S$. So $S$ admits a Buchbergers algorithm. [**BR08**, Sections 1 and 2] show how `DecideZeroRows`, `RightDivide`, and `SyzygiesOfRows` boil down to Gröbner basis computations for polynomial rings, and since by Proposition III.2.5 homogeneous ideal generators lead homogeneous Gröbner bases, the described algorithms can be applied to the $G$-graded case. □

**Definition 2.8.** Let $S$ be a left computable $G$-graded ring.

(1) Let $B \in S^{m \times k}, A \in S^{\ell \times n}, N \in S^{m-\ell \times n}$ have homogeneous entries. We can compute
$$(X\ Y) := \texttt{RightDivide}\left(B, \begin{pmatrix} A \\ N \end{pmatrix}\right)$$
and define
$$\texttt{RightDivide}\,(B, A, N) := X.$$

(2) For two matrices with homogeneous entries $A \in S^{m \times n}, N \in S^{k \times n}$ we can compute
$$(K\ L) := \texttt{SyzygiesOfRows}\left(\begin{pmatrix} A \\ N \end{pmatrix}\right)$$
and define
$$\texttt{SyzygiesOfRows}\,(A, N) := K.$$

**Definition 2.9.** Let $S$ be a computable $G$-graded ring.

(1) Let $A \in S^{m \times n}$ be a matrix and $\omega \in G^n$. Then we define
$$\omega' := \texttt{NonTrivialDegreePerRow}\,(A, \omega),$$
with
$$\omega'_i := \omega_1$$
if $A_{i,1} = \cdots = A_{i,n} = 0$, or
$$\omega'_i := \deg(A_{i,j}) + \omega_j$$
if $A_{i,1} = \cdots = A_{i,j-1} = 0$ and $A_{i,j} \neq 0$ for $i = 1, \ldots, n$.

(2) Let $A \in S^{m \times n}$ be a matrix and $\omega \in G^m$. Then we define
$$\omega' := \texttt{NonTrivialDegreePerColumn}\,(A, \omega),$$
with
$$\omega'_i := \omega_1$$
if $A_{1,i} = \cdots = A_{n,i} = 0$, or
$$\omega'_i := \deg(A_{j,i}) + \omega_j$$
if $A_{1,i} = \cdots = A_{j-1,i} = 0$ and $A_{j,i} \neq 0$ for $i = 1, \ldots, n$.

**Proposition 2.10.** *The functions defined in Definition III.2.9 are computable.*

**Theorem 2.11.** *Let $S$ be a $G$-graded computable ring. Then $S$-*grpres *is computable.*

PROOF.

(1) Objects are pairs of matrices over $S$ and lists of elements in $G$. Since both $S$ and $G$ have realizations, there is a realization for the category.

(2) The equality of objects and morphisms are computable, since equality in $S$ is decidable.
(3) Let $\alpha := (M, A, N), \beta := (M, A', N) \in \mathrm{Mor}_{S\text{-grpres}}$ and $N' :=$ `UnderlyingMatrix`. We have
$$\alpha \sim \beta$$
iff $XN' = A - A'$ is solvable. Since $S$ is computable, we compute $A - A'$ and use
$$\texttt{DecideZeroRows}\,(A - A', N')$$
to decide whether the system is solvable. So the congruence of morphisms is decidable.
(4) $0, 1 \in S$ can be constructed, as well as $0 \in G$. So the identity morphism is computable.
(5) Since products and sums of elements of $S$ are computable, and sums in $G$ are computable, the composition of morphisms is computable.
(6) The well-definedness of objects is computable since $G$ is computable.
(7) The well-definedness of a morphism is computable because `DecideZeroRows` and $G$ are computable. □

**Theorem 2.12.** *Let $S$ be a computable $G$-graded ring. Then $S$-grpres is computable abelian.*

Proof. We go though the list of constructions in Chapter II and give the necessary algorithms and prove their correctness. They are all based on the algorithms defined for the ring to be computable. If not stated otherwise, the compatibility of entry degrees of morphisms follows directly from Proposition III.1.9.

(1) We start by constructing the zero morphism between two objects. For two objects $M, N \in \mathrm{Obj}_{S\text{-grpres}}$ we define
$$\mathrm{ZeroMorphism}\,(M, N) := (M, 0, N)\,,$$
where $0$ is the $g_M \times g_N$ zero matrix. The triple $(M, 0, N)$ clearly defines a morphism, and is computable since $0 \in S$ is computable. Let $\alpha := (N, B, R)$ and $\beta := (T, C, N)$ be morphisms. Then we have
$$\mathrm{ZeroMorphism}\,(M, N)\,\alpha = (M, 0, R) = \mathrm{ZeroMorphism}\,(M, R)$$
and
$$\beta\ \mathrm{ZeroMorphism}\,(M, N) = (T, 0, M) = \mathrm{ZeroMorphism}\,(T, M)\,,$$
so $(M, 0, N)$ fulfills the universal properties of the zero morphism.
(2) We define
$$\mathrm{AdditionForMorphisms}\,((M, A, N), (M, A', N)) := (M, A + A', N)\,.$$
Since $(M, A, N)$ and $(M, A', N)$ are morphisms, there exist matrices $X$ and $X'$ with $XN' = AM'$ and $X'N' = A'M'$. Then we have
$$(A + A')\,M' = AM' + A'M' = XN' + X'N' = (X + X')\,N'$$

so $(M, A + A', N)$ is indeed a morphism, and since the sum of elements in $S$ is computable, $A + A'$ is computable. Since addition of matrices is associative and distributive with multiplication, the defined sum of morphisms fulfills the properties of the addition.

(3) We define

$$\text{AdditiveInverse}\left((M, A, N)\right) := (M, -A, N)\,.$$

We have

$$(M, A, N) + (M, -A, N) = (M, 0, N) = \text{ZeroMorphism}\,(M, N)$$

and therefore $(M, -A, N)$ fulfills the properties of the additive inverse $(M, A, N)$. The additive inverse is computable since negation in $S$ is computable.

(4) We define

$$\text{ZeroObject}\,(S\text{-grpres}) := (o, ())\,,$$

where $o$ is the $0 \times 0$ matrix, and $() \in G^0$ is the empty list. The object $(o, ())$ is obviously computable. Let $M \in \text{Obj}_{S\text{-grpres}}$. Indeed, there is exactly one morphism

$$\alpha : M \to \text{ZeroObject}\,(S\text{-grpres})\,,$$

namely $\alpha = (M, A, (o, ()))$ where $A$ is the $g_M \times 0$ matrix, and exactly one morphism

$$\beta : \text{ZeroObject}\,(S\text{-grpres}) \to M,$$

namely $\beta\left((o, ()), A, M\right)$ where $A$ is the $0 \times g_M$ matrix. Both morphisms are computable using ZeroMorphism.

(5) We now define the direct sum. For two objects $M := (M', \omega_M)$ and $N := (N', \omega_N)$ we will prove that the object

$$D := \text{DirectSum}\,(M, N) := \left(\begin{pmatrix} M' & 0 \\ 0 & N' \end{pmatrix}, (\omega_M, \omega_N)\right)$$

together with the projections

$$\text{ProjectionInFactorOfDirectSum}\left((M, N), 1\right) := \left(D, \begin{pmatrix} 1_M \\ 0_N \end{pmatrix}, M\right),$$
$$\text{ProjectionInFactorOfDirectSum}\left((M, N), 2\right) := \left(D, \begin{pmatrix} 0_M \\ 1_N \end{pmatrix}, N\right)$$

and the injections

$$\text{InjectionOfCofactorOfDirectSum}\left((M, N), 1\right) := \left(M, (1_M\ 0_N), D\right),$$
$$\text{InjectionOfCofactorOfDirectSum}\left((M, N), 2\right) := \left(N, (0_M\ 1_N), D\right)$$

is a direct sum, where $1_M$ is the $g_M \times g_M$ identity matrix, $1_N$ is the $g_N \times g_N$ identity matrix, and $0_M$ and $0_N$ are the $g_M \times g_M$ and $g_N \times g_N$ zero matrices, respectively. Since 0 and 1 in $S$ are computable, the direct sum object, the projections, and the injections are computable. Indeed, let $(M, A, L)$ and $(N, B, L)$ be two morphisms. We define the universal morphism from the direct sum to be

$$\begin{aligned}&\text{UniversalMorphismFromDirectSum}\left((M, A, L), (N, B, L)\right)\\ &\quad := \left(D, \left(\begin{array}{c} A \\ B \end{array}\right), L\right).\end{aligned}$$

Since there is nothing to compute, this morphism is computable. For two morphisms $(L, A, M)$ and $(L, B, N)$ we define the universal morphism into the direct sum by

$$\begin{aligned}&\text{UniversalMorphismIntoDirectSum}\left((L, A, M), (L, B, N)\right)\\ &\quad := \left(L, \left(A\ B\right), D\right).\end{aligned}$$

Since there is again nothing to compute, this morphism is computable.

We now show that these two universal morphisms fulfill the properties of the universal morphisms of product and coproduct. Let

$$\begin{aligned}\iota &:= \text{UniversalMorphismFromDirectSum}\left((M, A, L), (N, B, L)\right),\\ \pi &:= \text{UniversalMorphismIntoDirectSum}\left((L, A, M), (L, B, N)\right).\end{aligned}$$

By blockwise matrix multiplication it follows that

$$\begin{aligned}\text{InjectionOfCofactorOfDirectSum}\left((M, N), 1\right)\iota &= (M, A, L),\\ \text{InjectionOfCofactorOfDirectSum}\left((M, N), 2\right)\iota &= (N, B, L),\\ \pi\,\text{ProjectionInFactorOfDirectSum}\left((M, N), 1\right) &= (L, A, M),\\ \pi\,\text{ProjectionInFactorOfDirectSum}\left((M, N), 2\right) &= (L, B, N).\end{aligned}$$

We still need to show the uniqueness of the universal morphisms. We show this for the morphism from the direct sum, the dual case for the morphism into the direct sum is analog. Let $\iota$ be as above and

$$\begin{aligned}\iota_1 &:= \text{InjectionOfCofactorOfDirectSum}\left((M, N), 1\right),\\ \iota_2 &:= \text{InjectionOfCofactorOfDirectSum}\left((M, N), 2\right).\end{aligned}$$

Now, let $\varphi := \left(D, \left(\begin{array}{c} A' \\ B' \end{array}\right), L\right)$ be a morphism with

$$\varphi\pi_1 \sim (M, A, L) \text{ and } \varphi\pi_2 \sim (N, B, L).$$

Then we have matrices $X$ and $Y$ such that $XL' = A - A'$ and $YL' = B - B'$. Therefore we have

$$\left(\begin{array}{c} X \\ Y \end{array}\right) L = \left(\begin{array}{c} A \\ B \end{array}\right) - \left(\begin{array}{c} A' \\ B' \end{array}\right),$$

so we have $\varphi \sim \iota$.

(6) We give the constructions for the kernel. Let $(M, A, N)$ be a morphism and $J' := \texttt{SyzygiesOfRows}\,(A, N')$. Furthermore, let $K' := \texttt{SyzygiesOfRows}\,(J', M')$. We define

$$\text{KernelObject}\left((M, A, N)\right) := (K', \omega)$$

with

$$\omega := \texttt{NonTrivialDegreePerRow}\left(J', \omega_M\right).$$

By definition $K'$ has homogeneous entries, and since there is a matrix $Y$ with homogeneous entries such that $K'J' = YM'$, the tuple $\omega$ fulfills the degree properties. So the tuple $(K', \omega)$ defines an object in $S$-grpres. Furthermore, let $K := (K', \omega)$ be the kernel object. Then the kernel embedding is defined as

$$\text{KernelEmbedding}\left((M, A, N)\right) := (K, J', M).$$

We have $J'A + XN' = 0$, so the composition of kernel embedding and morphism is congruent to 0, and since there is a matrix $X'$ with $K'J' + X'M' = 0$ the triple $(K, J', M)$ is compatible. By the definition of `NonTrivialDegreePerRow` the grading for $(K, J', M)$ is also compatible, so $(K, J', M)$ is indeed a morphism. Since `SyzygiesOfRows` is computable in $S$ and `NonTrivialDegreePerRow` is computable, kernel embedding and kernel object are computable. Now, let $(T, H, M)$ be a test morphism for the kernel, i.e,

$$(T, HA, N) \sim (T, 0, N),$$

which means that there is a matrix $Z'$ with $Z'N' = HA$. We define

$$\begin{aligned}&\text{KernelLift}\left((M, A, N), (T, H, M)\right)\\ &\quad := \left(T, \texttt{RightDivide}\left(H, J'\right), K\right).\end{aligned}$$

By definition, we have

$$\begin{aligned}\text{KernelLift}\left((M, A, N), (T, H, M)\right) \text{KernelEmbedding}\left((M, A, N)\right)&\\ \sim (T, H, M).&\end{aligned}$$

The uniqueness of the kernel lift follows from the fact that $(K, J', M)$ is a monomorphism. Since `RightDivide` is computable in $S$, the KernelLift is computable.

(7) We give the constructions for the cokernel. Let $(M, A, N)$ be a morphism. The cokernel is defined by

$$\text{CokernelObject}\left((M, A, N)\right) := \left(\begin{pmatrix} A \\ N \end{pmatrix}, \omega_N\right),$$

so the generator degrees are the generator degrees of $N$. Therefore, by the definition of an object and morphism, the object is well-defined.

The cokernel projection is defined by

$$\text{CokernelProjection}\left((M, A, N)\right) := (N, 1, C),$$

where $C := \text{CokernelObject}\left((M, A, N)\right)$. Since we have

$$(M, A, N)\,\text{CokernelProjection}\left((M, A, N)\right) = (M, A, C),$$

and $A = XC$ is clearly solvable for $X$, we have $(M, A, C) \sim 0_{M,C}$. Let $(N, H, T)$ be a test morphism for the cokernel, i.e.,

$$(M, A, N)\,(N, H, T) \sim 0_{M,T}.$$

We define the cokernel colift as

$$\mathrm{CokernelColift}\left((M, A, N), (N, H, T)\right) := (C, H, T),$$

with $C := \mathrm{CokernelObject}\left((M, A, N)\right)$. Obviously, this morphism fulfills the universal property of the cokernel colift, and the morphism is well-defined. Again, the uniqueness follows from the fact that the cokernel projection is an epimorphism.

(8) To define LiftAlongMonomorphism and ColiftAlongEpimorphism, we first show that in $S$-grpres, every monomorphism is the kernel of its cokernel, and every epimorphism is the cokernel of its kernel. We then use KernelLift to define LiftAlongMonomorphism and CokernelColift to define ColiftAlongEpimorphism.

We start by showing that every monomorphism is the kernel of its cokernel. Let $\varphi := (M, A, N)$ be a mono. Then we have

$$\epsilon := \mathrm{CokernelProjection}\left(\varphi\right) = (N, 1_N, C)$$

with $C = \mathrm{CokernelObject}\left(\varphi\right)$. Let $\gamma := (M_2, G, N)$ be another morphism with $\gamma\epsilon \sim 0_{M_2,C}$, i.e.,

$$\begin{pmatrix} A \\ N \end{pmatrix} \geqslant_{\mathrm{row}} G,$$

which means that there exist matrices $X, Y$ with $G = XA + YN$. A candidate for a unique lift would be $\lambda := (M_2, X, M)$ with

$$X := \texttt{RightDivide}\left(G, A, N'\right)$$

once we have shown that the triple $(M_2, X, N)$ is compatible, i.e., $M' \geqslant_{\mathrm{row}} M_2' X$ and

$$\omega_{M_2,k} + \deg\left(X_{k,j}\right) = \omega_{M,j}$$

for all $k = 1, \ldots, g_{M_2}$ and $j = 1, \ldots, g_M$ with $X_{k,j} \neq 0$. Since $\gamma$ and $\varphi$ are morphisms and we have $X = \texttt{RightDivide}\left(G, A, N'\right)$, the degrees of $\lambda$ are compatible. Indeed, note that $N' \geqslant_{\mathrm{row}} M_2' G$ since $\gamma$ is a morphism. Furthermore $N' \geqslant_{\mathrm{row}} M_2' Y N'$. Hence

$$N' \geqslant_{\mathrm{row}} M_2' G - M_2' Y N' = M_2' X A = \left(M_2' X\right) A.$$

Since $\varphi$ is a mono, the kernel embedding of $\varphi$ vanishes, which means that

$$M' \geqslant_{\mathrm{row}} \texttt{SyzygiesOfRows}\left(A, N'\right) =: K.$$

But $K$ row-dominates by definition any $T$ with $N' \geqslant_{\mathrm{row}} TA$, in particular we have

$$M' \geqslant_{\mathrm{row}} K \geqslant_{\mathrm{row}} M_2' X.$$

So we set

$$\mathrm{LiftAlongMonomorphism}\left(\gamma, \varphi\right) := \lambda = (M_2, X, M).$$

We continue to show that any epi is the cokernel projection of its kernel embedding. Let $\varphi := (M, A, N)$ be an epi and

$$\kappa := \text{KernelEmbedding}\,(\varphi) = (K_2, K, M)\,.$$

It suffices to show that the unique colift

$$\lambda := \text{CokernelColift}\,(\kappa, \varphi) = (C, A, N)$$

of $\varphi$ along $\epsilon := \text{CokernelProjection}\,(\kappa)$ is an isomorphism, where $C$ denotes the cokernel object of $\kappa$. We have $K = \texttt{SyzygiesOfRows}\,(A, N')$. An inverse $\alpha$ of the colift $\lambda$ must satisfy $\alpha\lambda \sim \mathrm{id}_N$. This implies that

$$\alpha = (N, Y, C)$$

with

$$Y := \texttt{RightDivide}\,(1_N, A, N')\,,$$

where $1_N$ is the $g_N \times g_N$ identity matrix. The matrix $Y$ exists since $\varphi$ is an epi. Now we show that the triple $\alpha = (N, Y, C)$ is compatible and hence defines a morphism. By definition of $Y$ there exists a matrix $Z$ such that

$$YA + ZN' = 1_N.$$

Multiplying with $N'$ from the left we obtain $N'YA + N'ZN' = N'$, which is equivalent to

$$N'YA = (1_N - N'Z)\,N',$$

i.e., $N' \geqslant_{\text{row}} N'YA$. But $K$ row-dominates by definition any $T$ with $N' \geqslant_{\text{row}} TA$, hence $K \geqslant_{\text{row}} N'Y$ and therefore $\alpha$ is a compatible triple. Since $\alpha\lambda \sim \mathrm{id}_N$ by definition of $Y$, it remains to show that $\lambda\alpha \sim \mathrm{id}_C$. We multiply with $A$ from the left and obtain

$$AYA + AZN' = A,$$

or equivalently

$$(AY - 1_N)\,A = (AZ)\,N',$$

hence $N \geqslant_{\text{row}} (AY - 1_N)\,A$ and as above $K \geqslant_{\text{row}} (AY - 1_N)$, which concludes the proof. □

From the algorithms for the categorical constructions in $S$-grpres, we can see why the distinction of congruence and equality of morphisms is important.

**Example 2.13.** Let $S$ be the $\mathbb{Q}$ with a trivial grading, $M' := (1) \in \mathbb{Q}^{1\times 1}$, $M := (M', (0))$, and $\varphi := (M, M', M)$. Then the cokernel projection of $\varphi$ is

$$\epsilon := \text{CokernelProjection}\,(\varphi) = \left(M, M', \left(\left(\begin{array}{c} 1 \\ 1 \end{array}\right), (0)\right)\right).$$

So the composition $\varphi\epsilon$ of $\varphi$ and its cokernel projection is the same morphism as the cokernel projection $\epsilon$ of $\varphi$. The morphism $\varphi\epsilon$ is congruent to the zero morphism, but not equal.

Remark 2.14. In the actual Cap implementation the category $S$-grpres is implemented in two steps: There is a category of module presentation over a non-graded ring, and data structures and algorithms of this non-graded module presentations are just like the above, but without any grading conditions. The implementation in Cap can be found in Appendix F.2.

The implementation of the graded module presentation category then uses this category and equips the objects with gradings. It can also check whether the matrices in the underlying non-graded presentation category represent well-defined objects and morphisms in the graded module presentation category. The corresponding implementation can be found in Appendix F.4.

Also, the category defined in this chapter describes left modules. Since we are only going to work with commutative rings, this is general enough. Cap has implementations for both left and right modules. The implementations for right modules can be found in Appendix F.3 and Appendix F.5.

CHAPTER IV

# Generalized morphisms and Serre quotients

In this chapter we present an example of the flexibility a categorical organized setup offers for the implementation of computable categories: We create a computable category on top of another computable abelian category, only using the categorical constructions from that underlying category. We first define the category of generalized morphisms of an abelian category. Then we use a certain subcategory thereof, the category of so-called Gabriel morphisms, to establish the computability of Serre quotient categories.

For categorical operations, e.g., KernelEmbedding and UniversalMorphismFromDirectSum, we use the notation from Chapter II.

## 1. The category of generalized morphisms

Generalized morphisms will serve as a data structure for morphisms in Serre quotient categories. Generalized morphisms already are an interesting tool by themselves, for example for performing diagram chases. There are several data structures for generalized morphisms, and we will explicitly describe three of them. An extensive description of generalized morphisms, including their universal properties, can be found in [**Pos17**, Section II.1]. We will limit our exposition to the description of the data structures and operations of generalized morphism categories, and the proof of the necessary constructions to establish the computability of Serre quotients. We will also describe how to convert those three different generalized morphism data structures into each other, to state that they define equivalent categories.

The three described versions of the generalized morphisms category are currently implemented in Cap, and the implemented algorithms can be found in Appendices F.6, F.7, and F.8.

**1.a. Preliminaries.** To define generalized morphisms we need the notions of the fiber product and the pushout of two morphisms. Both constructions are computable in a computable preabelian category.

**Definition 1.1.** Let $\mathcal{A}$ be a category.

(1) Let $\varphi : A_1 \to M, \psi : A_2 \to M$ in $\mathrm{Mor}_{\mathcal{A}}$. The **fiber product** or **pullback** of $\varphi$ and $\psi$ is an object $P_f \in \mathrm{Obj}_{\mathcal{A}}$ together with two morphisms $\pi_1 : P_f \to A_1$ and $\pi_2 : P_f \to A_2$ such that $\pi_1\varphi \sim \pi_2\psi$ and for every pair of morphisms $\tau_1 : Q \to A_1$ and $\tau_1 : Q \to A_2$ with $\tau_1\varphi \sim \tau_2\psi$ there is an up to congruence unique morphism $\eta : Q \to P_f$ such that $\tau_1 \sim \eta\pi_1$ and $\tau_2 \sim \eta\pi_2$.[1]

[1]Recall, categories are categories with Hom-setoids

$$\begin{tikzcd}
& M & \\
A_1 \arrow[ur, "\varphi"] & & A_2 \arrow[ul, "\psi"'] \\
& P_f \arrow[ul, "\pi_1"] \arrow[ur, "\pi_2"'] & \\
& Q \arrow[uul, "\tau_1"] \arrow[uur, "\tau_2"'] \arrow[u, dashed, "\eta"'] &
\end{tikzcd}$$

(2) Let $\varphi : M \to A_1, \psi : M \to A_2$ in $\mathrm{Mor}_{\mathcal{A}}$. The **pushout** of $\varphi$ and $\psi$ is an object $P_p \in \mathrm{Obj}_{\mathcal{A}}$ together with two morphisms $\iota_1 : A_1 \to P_p$ and $\iota_2 : A_2 \to P_p$ such that $\varphi\iota_1 \sim \psi\iota_2$ and for every pair of morphisms $\tau_1 : A_1 \to Q$ and $\tau_2 : A_2 \to Q$ with $\varphi\tau_1 \sim \psi\tau_2$ there is an up to congruence unique morphism $\epsilon : P_p \to Q$ such that $\tau_1 \sim \iota_1\epsilon$ and $\tau_2 \sim \iota_2\epsilon$.

$$\begin{tikzcd}
& Q & \\
& P_p \arrow[u, dashed, "\epsilon"'] & \\
A_1 \arrow[uur, "\tau_1"] \arrow[ur, "\iota_1"'] & & A_2 \arrow[uul, "\tau_2"'] \arrow[ul, "\iota_2"] \\
& M \arrow[ul, "\varphi"] \arrow[ur, "\psi"'] &
\end{tikzcd}$$

**Definition 1.2.** Let $\mathcal{A}$ be a category computable by the realization $\mathfrak{R}$.

(1) We say $\mathcal{A}$ has **computable fiber products** if the functions

$$\begin{aligned}
\mathrm{FiberProduct} &: \mathcal{M} \to \mathrm{Obj}_{\mathcal{A}},\ (\varphi, \psi) \mapsto P_f, \\
\mathrm{ProjectionInFactorOfFiberProduct} &: \mathcal{M} \times \{1, 2\} \to \mathrm{Mor}_{\mathcal{A}},\ (\varphi, \psi, i) \mapsto \pi_i, \\
\mathrm{UniversalMorphismIntoFiberProduct} &: \mathcal{N} \to \mathrm{Mor}_{\mathcal{A}},\ (\varphi, \psi, \tau_1, \tau_2) \mapsto \eta,
\end{aligned}$$

with

$$\mathcal{M} := \bigcup_{A_1, A_2, M \in \mathrm{Obj}_{\mathcal{A}}} \mathrm{Hom}_{\mathcal{A}}(A_1, M) \times \mathrm{Hom}_{\mathcal{A}}(A_2, M),$$

$$\mathcal{N} := \bigcup_{A_1, A_2, M, Q \in \mathrm{Obj}_{\mathcal{A}}} \mathrm{Hom}_{\mathcal{A}}(A_1, M) \times \mathrm{Hom}_{\mathcal{A}}(A_2, M) \times \mathrm{Hom}_{\mathcal{A}}(Q, A_1) \times \mathrm{Hom}_{\mathcal{A}}(Q, A_2)$$

are computable by $\mathfrak{R}$.

(2) We say $\mathcal{A}$ has **computable pushouts** if the functions

$$\mathrm{Pushout} : \mathcal{M} \to \mathrm{Obj}_{\mathcal{A}},\ (\varphi, \psi) \mapsto P_p,$$
$$\mathrm{InjectionOfCofactorOfPushout} : \mathcal{M} \times \{1, 2\} \to \mathrm{Mor}_{\mathcal{A}},\ (\varphi, \psi, i) \mapsto \iota_i,$$
$$\mathrm{UniversalMorphismFromPushout} : \mathcal{N} \to \mathrm{Mor}_{\mathcal{A}},\ (\varphi, \psi, \tau_1, \tau_2) \mapsto \epsilon,$$

with

$$\mathcal{M} := \bigcup_{A_1, A_2, M \in \mathrm{Obj}_{\mathcal{A}}} \mathrm{Hom}_{\mathcal{A}}(M, A_1) \times \mathrm{Hom}_{\mathcal{A}}(M, A_2),$$

$$\mathcal{N} := \bigcup_{A_1, A_2, M, Q \in \mathrm{Obj}_{\mathcal{A}}} \mathrm{Hom}_{\mathcal{A}}(M, A_1) \times \mathrm{Hom}_{\mathcal{A}}(M, A_2) \times \mathrm{Hom}_{\mathcal{A}}(A_1, Q) \times \mathrm{Hom}_{\mathcal{A}}(A_2, Q)$$

are computable by $\mathfrak{R}$.

In a computable preabelian category the fiber product and the pushout can be derived from other categorical operations.

**Theorem 1.3.** *Let $\mathcal{A}$ be a computable preabelian category. Then $\mathcal{A}$ has computable fiber products.*

Proof. Let $\varphi : A_1 \to M$ and $\psi : A_2 \to M$ be morphisms in $\mathcal{A}$. We first give constructions for FiberProduct and ProjectionInFactorOfFiberProduct. Let

$$D := \mathrm{DirectSum}(A_1, A_2)$$

with projections

$$\pi_i := \mathrm{ProjectionInFactorOfDirectSum}((A_1, A_2), i)$$

for $i = 1, 2$. To compute the projections in the factors of the fiber product of $\varphi$ and $\psi$ and the fiber product object itself we compute the diagonal difference

$$\delta := \pi_1\varphi - \pi_2\psi$$

and set

$$P := \mathrm{FiberProduct}(\varphi, \psi) := \mathrm{KernelObject}(\delta).$$

With

$$\chi := \mathrm{KernelEmbedding}(\delta)$$

we set

$$\kappa_i := \mathrm{ProjectionInFactorOfFiberProduct}(\varphi, \psi, i) := \mathrm{PreCompose}(\chi, \pi_i)$$

for $i = 1, 2$.

$$\begin{tikzcd}
\mathrm{KernelObject}(\delta) \arrow[rrd, hook, "\chi"] \arrow[rrrd, bend left, "\kappa_1"] \arrow[rrdd, bend right, "\kappa_2"'] & & & \\
& & D \arrow[r, "\pi_1"] \arrow[d, "\pi_2"'] \arrow[rd, "\delta"] & A_1 \arrow[d, hook, "\varphi"] \\
& & A_2 \arrow[r, hook, "\mu_2"'] & M
\end{tikzcd}$$

To construct the universal morphisms into the fiber product of $\tau_1 : Q \to A_1$ and $\tau_2 : Q \to A_2$ set

$$\tau := \mathrm{UniversalMorphismIntoDirectSum}\left(\tau_1, \tau_2\right).$$

Then $\tau \in \mathrm{Hom}_{\mathcal{A}}\left(Q, D\right)$, and we have

$$\tau\delta \sim 0_{P,M}.$$

So we can define the universal morphisms to be

$$\mathrm{UniversalMorphismIntoFiberProduct}\left(\varphi, \psi, \tau_1, \tau_2\right) := \mathrm{KernelLift}\left(\delta, \tau\right). \qquad \square$$

Since the pushout is dual to the fiber product, it is also computable in preabelian categories.

**Corollary 1.4.** *Let $\mathcal{A}$ be a computable preabelian category. Then $\mathcal{A}$ has computable pushouts.*

For the rest of this chapter, we use the following notation.

**Notation.** Since we often work with different categories at the same time in this chapter, we will extend the operators for categorical operations with the name of the category, e.g., the composition in the category $\mathcal{A}$ will be denoted by $\mathrm{PreCompose}_{\mathcal{A}}$.

**1.b. Generalized morphisms by cospans.** As first data structure for generalized morphisms we describe the generalized morphisms by cospans.

**Definition 1.5.** Let $\mathcal{A}$ be an abelian category. A pair $\varphi := (\alpha : A \to C, \beta : B \to C)$ with $\alpha, \beta \in \mathrm{Mor}_{\mathcal{A}}$ is called **cospan** with source $A$ and range $B$ in $\mathcal{A}$.

$$\begin{tikzcd}
& C & \\
A \arrow[ru, "\alpha"] \arrow[rr, dashed, "\varphi"'] & & B \arrow[lu, "\beta"']
\end{tikzcd}$$

We call $\alpha$ the **arrow** of $\varphi$ and $\beta$ the **reversed arrow** of $\varphi$.

**Definition 1.6.** A cospan $A \xrightarrow{\alpha} C \xleftarrow{\beta} B$ in an abelian category $\mathcal{A}$ is called **normalized** if the universal morphism from the direct sum $A \oplus B \xrightarrow{\langle\alpha,\beta\rangle} C$ is an epimorphism.

**Definition 1.7** (Generalized morphisms by cospans)**.** Let $\mathcal{A}$ be an abelian category. The **category of generalized morphisms by cospans** $\mathrm{G}^{\mathrm{C}}(\mathcal{A})$ has the following objects and morphisms:

(1) $\mathrm{Obj}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A})} := \mathrm{Obj}_{\mathcal{A}}$.

(2) For two objects $A, B \in \mathrm{Obj}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A})}$ we set

$$\mathrm{Hom}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A})}(A,B) := \left( \bigcup_{C \in \mathrm{Obj}_{\mathcal{A}}} \mathrm{Hom}_{\mathcal{A}}(A,C) \times \mathrm{Hom}_{\mathcal{A}}(B,C) \right) / \simeq,$$

where $\simeq$ is the following equivalence relation: Let $\varphi := (\alpha_1, \beta_1)$ and $\psi := (\alpha_2, \beta_2)$ be two cospans in $\mathrm{Hom}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A})}(A,B)$, and $F_i := \mathrm{FiberProduct}_{\mathcal{A}}(\alpha_i, \beta_i)$, $i = 1,2$, the fiber products of $\alpha_1$ and $\beta_1$, and $\alpha_2$ and $\beta_2$ respectively, together with injections from the fiber product $\iota_{i,1} : F_i \to A$ and $\iota_{i,2} : F_i \to B$.

$$\begin{tikzcd}
 & C_1 & \\
 & F_1 \arrow[dl, "\iota_{1,1}"'] \arrow[dr, "\iota_{1,2}"] \arrow[d, hook, "\kappa_1"] \arrow[dd, bend right=50, "\alpha"'] & \\
A \arrow[uur, "\alpha_1"] \arrow[ddr, "\alpha_2"'] & A \oplus B & B \arrow[uul, "\beta_1"'] \arrow[ddl, "\beta_2"] \\
 & F_2 \arrow[ul, "\iota_{2,1}"] \arrow[ur, "\iota_{2,2}"'] \arrow[u, hook, "\kappa_2"'] & \\
 & C_2 &
\end{tikzcd}$$

The cospans $\varphi$ and $\psi$ are equivalent if the monomorphisms $\kappa_1 := \{\iota_{1,1}, \iota_{1,2}\} : F_1 \hookrightarrow A \oplus B$ and $\kappa_2 := \{\iota_{2,1}, \iota_{2,2}\} : F_2 \hookrightarrow A \oplus B$ are equivalent as subobjects of $A \oplus B$, i.e., if there is an isomorphism $\alpha : F_1 \to F_2$ such that $\alpha\kappa_2 \sim \kappa_1$ holds. [2]

For a morphism $\varphi \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A})}$ represented by the normalized cospan $(\alpha, \beta)$ we call $\alpha$ the **arrow** of $\varphi$ and write $\mathrm{Arrow}(\varphi) := \alpha$, and $\beta$ the **reversed arrow** of $\varphi$ and write $\mathrm{ReversedArrow}(\varphi) := \beta$.[3] The identity morphism for an $A \in \mathrm{Obj}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A})}$ can be represented by the cospan consisting of two times the identity of $A$ (viewed as an object in $\mathcal{A}$), or,

[2] $\mathrm{Hom}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A})}(A,B)$ is a set, not a setoid, i.e., two morphisms are congruent if they are equal.

[3] The terms arrow and reversed arrow of a morphism $\varphi \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A})}$ are dependent on the choice of the representative of $\varphi$. Every time we use the terms arrow and reversed arrow, we assume a normalized representative is fixed.

written in operators:

$$\begin{aligned}\mathrm{Arrow}\left(\mathrm{IdentityMorphism}_{\mathrm{G^C}(\mathcal{A})}(A)\right) &:= \mathrm{IdentityMorphism}_{\mathcal{A}}(A)\,,\\ \mathrm{ReversedArrow}\left(\mathrm{IdentityMorphism}_{\mathrm{G^C}(\mathcal{A})}(A)\right) &:= \mathrm{IdentityMorphism}_{\mathcal{A}}(A)\,.\end{aligned}$$

The composition of two composable morphisms $\varphi, \psi \in \mathrm{Mor}_{\mathrm{G^C}(\mathcal{A})}$ is defined as follows: Let

$$\iota_i := \mathrm{InjectionOfCofactorOfPushout}_{\mathcal{A}}\left(\left(\mathrm{ReversedArrow}(\varphi), \mathrm{Arrow}(\psi)\right), i\right),$$

for $i = 1, 2$. Then $\varphi\psi$ is represented by

$$\begin{aligned}\mathrm{Arrow}(\varphi\psi) &:= \mathrm{PreCompose}_{\mathcal{A}}\left(\mathrm{Arrow}(\varphi), \iota_1\right)\\ \mathrm{ReversedArrow}(\varphi\psi) &:= \mathrm{PreCompose}_{\mathcal{A}}\left(\iota_2, \mathrm{ReversedArrow}(\psi)\right).\end{aligned}$$

$$\begin{tikzcd}
& & P & & \\
& X \arrow[ur, "\iota_1"] & & Y \arrow[ul, "\iota_2"'] & \\
A \arrow[ur, "\mathrm{Arrow}(\varphi)"] \arrow[rr, dashed, "\varphi"'] & & B \arrow[ul] \arrow[ur] \arrow[rr, dashed, "\psi"'] & & C \arrow[ul, "\mathrm{ReversedArrow}(\psi)"']
\end{tikzcd}$$

For a proof that this category is well-defined and the composition is compatible with the equivalence relation on the Hom-sets, see [**Pos17**, Subsection II.1.2].

**Proposition 1.8.** *Let $\varphi : A \to B$ in $Mor_{\mathrm{G^C}(\mathcal{A})}$ represented by the cospan $A \xrightarrow{\alpha} C \xleftarrow{\beta} B$. Then a normalized representative $A \xrightarrow{\alpha'} C' \xleftarrow{\beta'} B$ is computable.*

Proof. We compute

$$\iota_i := \mathrm{ProjectionInFactorOfFiberProduct}_{\mathcal{A}}\left((\alpha, \beta), i\right)$$

for $i = 1, 2$ and

$$\begin{aligned}\alpha' &:= \mathrm{InjectionOfCofactorOfPushout}_{\mathcal{A}}\left((\iota_1, \iota_2), 1\right),\\ \beta' &:= \mathrm{InjectionOfCofactorOfPushout}_{\mathcal{A}}\left((\iota_1\iota_2), 2\right).\end{aligned}$$

$$\begin{tikzcd}
 & & C & & \\
 & & C' & & \\
A \arrow[uurr, "\alpha"] \arrow[urr, "\alpha'"'] \arrow[rrrr, dashed, "\varphi"'] & & & & B \arrow[uull, "\beta"'] \arrow[ull, "\beta'"] \\
 & & P \arrow[ull, "\iota_1"] \arrow[urr, "\iota_2"'] & &
\end{tikzcd}$$

Since $\iota_1, \iota_2$ is a fiber product diagram of $\alpha'$ and $\beta'$, the cospan $(\alpha', \beta')$ is equivalent to the cospan $(\alpha, \beta)$. Furthermore, since $\alpha'$ and $\beta'$ are the pushout of a fiber product, the cospan $(\alpha', \beta')$ is normalized. $\square$

**Definition 1.9.** Let $\mathcal{A}$ be an abelian category and $\varphi \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A})}$. The generalized morphism $\varphi$ is called **honest** if ReversedArrow$(\varphi)$ is an isomorphism in $\mathcal{A}$. We define

$$\mathrm{HonestRepresentative}(\varphi) := \mathrm{Arrow}(\varphi)\, \mathrm{ReversedArrow}(\varphi)^{-1}$$

to be the **honest representative** of $\varphi$.

The category $\mathcal{A}$ embeds naturally into $\mathrm{G}^{\mathrm{C}}(\mathcal{A})$:

**Definition 1.10.** There is an embedding

$$\mathrm{F}^{\mathrm{C}} : \mathcal{A} \to \mathrm{G}^{\mathrm{C}}(\mathcal{A}),$$

defined as follows: For $A \in \mathrm{Obj}_{\mathcal{A}}$ set $\mathrm{F}^{\mathrm{C}}(A) := A$ and for a morphism $\varphi \in \mathrm{Mor}_{\mathcal{A}}$ the image $\mathrm{F}^{\mathrm{C}}(\varphi)$ is represented by

$$\begin{aligned} \mathrm{Arrow}\left(\mathrm{F}^{\mathrm{C}}(\varphi)\right) &:= \varphi, \\ \mathrm{ReversedArrow}\left(\mathrm{F}^{\mathrm{C}}(\varphi)\right) &:= \mathrm{IdentityMorphism}_{\mathcal{A}}(\mathrm{Range}(\varphi)). \end{aligned}$$

We call $\mathrm{F}^{\mathrm{C}}(\varphi)$ the **corresponding honest cospans** of $\varphi$.

**1.c. Generalized morphisms by spans.** The second data structure, generalized morphisms by spans, is dual to the generalized morphisms by cospans data structure.

**Definition 1.11.** Let $\mathcal{A}$ be an abelian category. A pair $\varphi := (\alpha : C \to A, \beta : C \to B)$ is called **span** with source $A$ and range $B$ in $\mathcal{A}$.

$$\begin{tikzcd}
A \arrow[rr, dashed, "\varphi"] & & B \\
& C \arrow[ul, "\beta"] \arrow[ur, "\alpha"'] &
\end{tikzcd}$$

We call $\alpha$ the **reversed arrow** of $\varphi$ and $\beta$ the **arrow** of $\varphi$.

**Definition 1.12.** A span $A \overset{\alpha}{\leftarrow} C \overset{\beta}{\rightarrow} B$ in an abelian category $\mathcal{A}$ is called **normalized** if the universal morphism into the direct sum $C \xrightarrow{\{\alpha,\beta\}} A \oplus B$ is a monomorphism.

**Definition 1.13** (Generalized morphisms by spans)**.** Let $\mathcal{A}$ be an abelian a category. The **category of generalized morphisms by spans** is the category $\mathrm{G}^{\mathrm{S}}(\mathcal{A})$ with the following objects and morphisms:

(1) We set $\mathrm{Obj}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})} := \mathrm{Obj}_{\mathcal{A}}$.
(2) For two objects $A, B \in \mathrm{Obj}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}$ we set

$$\mathrm{Hom}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}(A,B) := \left( \bigcup_{C \in \mathrm{Obj}_{\mathcal{A}}} \mathrm{Hom}_{\mathcal{A}}(C,A) \times \mathrm{Hom}_{\mathcal{A}}(C,B) \right) / \simeq,$$

where $\simeq$ describes the following equivalence relation: Let $\varphi := (\alpha_1, \beta_1)$ and $\psi := (\alpha_2, \beta_2)$ be two spans in $\mathrm{Hom}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}(A,B)$, and $P_1 := \mathrm{Pushout}_{\mathcal{A}}(\alpha_1, \beta_1)$ and $P_2 := \mathrm{Pushout}_{\mathcal{A}}(\alpha_2, \beta_2)$ the pushouts of $\alpha_1$ and $\beta_1$, and $\alpha_2$ and $\beta_2$ respectively, together with projections to the pushout $\pi_{i,1} : A \to P_i$ and $\pi_{i,2} : B \to P_i$.

$$\begin{tikzcd}
& & C_1 \arrow[lldd, "\alpha_1"'] \arrow[rrdd, "\beta_1"] & & \\
& & P_1 \arrow[dd, bend right, "\alpha"'] & & \\
A \arrow[urr, "\pi_{1,1}"'] \arrow[drr, "\pi_{2,1}"] & & A \oplus B \arrow[u, two heads, "\kappa_1"'] \arrow[d, two heads, "\kappa_2"] & & B \arrow[ull, "\pi_{1,2}"] \arrow[dll, "\pi_{2,2}"'] \\
& & P_2 & & \\
& & C_2 \arrow[lluu, "\alpha_2"] \arrow[rruu, "\beta_2"'] & &
\end{tikzcd}$$

The pairs $\varphi$ and $\psi$ are equivalent if the epimorphisms $\kappa_1 := \langle \pi_{1,1}, \pi_{1,2} \rangle : A \oplus B \twoheadrightarrow P_1$ and $\kappa_2 := \langle \pi_{2,1}, \pi_{2,2} \rangle : A \oplus B \twoheadrightarrow P_2$ are equivalent as factor objects of $A \oplus B$, i.e., if there is an isomorphism $\alpha : P_1 \to P_2$ such that $\kappa_1 \alpha \sim \kappa_2$ holds.[4]

For a morphism $\varphi \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}$ represented by the normalized cospan $(\alpha, \beta)$ we call $\beta$ the **arrow** of $\varphi$ and write $\mathrm{Arrow}(\varphi) := \beta$, and $\alpha$ the **reversed arrow** of $\varphi$ and write $\mathrm{ReversedArrow}(\varphi) := \alpha$.[5]

The identity morphism for an $A \in \mathrm{Obj}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}$ is represented by the span consisting of two times the identity of $A$ (viewed as an object in $\mathcal{A}$), or, written in operators:

$$\mathrm{Arrow}\left(\mathrm{IdentityMorphism}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}(A)\right) := \mathrm{IdentityMorphism}_{\mathcal{A}}(A),$$
$$\mathrm{ReversedArrow}\left(\mathrm{IdentityMorphism}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}(A)\right) := \mathrm{IdentityMorphism}_{\mathcal{A}}(A).$$

The composition of two composable morphisms $\varphi : A \to B, \psi : B \to C \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}$ is defined as follows: Let

$$\pi_i := \mathrm{ProjectionInFactorOfFiberProduct}_{\mathcal{A}}\left(\left(\mathrm{Arrow}(\varphi), \mathrm{ReversedArrow}(\psi)\right), i\right)$$

for $i = 1, 2$. Then $\varphi\psi$ is represented by

$$\mathrm{ReversedArrow}(\varphi\psi) := \mathrm{PreCompose}_{\mathcal{A}}\left(\pi_1, \mathrm{ReversedArrow}(\varphi)\right),$$
$$\mathrm{Arrow}(\varphi\psi) := \mathrm{PreCompose}_{\mathcal{A}}\left(\pi_2, \mathrm{Arrow}(\psi)\right).$$

$$\begin{tikzcd}
A \arrow[rr, dashed, "\varphi"] & & B \arrow[rr, dashed, "\psi"] & & C \\
& X \arrow[ul, "\mathrm{ReversedArrow}(\varphi)"] \arrow[ur] & & Y \arrow[ul] \arrow[ur, "\mathrm{Arrow}(\psi)"'] & \\
& & P \arrow[ul, "\pi_1"] \arrow[ur, "\pi_2"'] & &
\end{tikzcd}$$

For a proof that this category is well-defined and the composition is compatible with the equivalence relation on the Hom-sets, see [**Pos17**, Subsection II.1.2].

**Proposition 1.14.** *Let $\varphi : A \to B$ in $Mor_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}$, represented by the span $A \xleftarrow{\alpha} C \xrightarrow{\beta} B$. Then a normalized representative $A \xleftarrow{\alpha'} C' \xrightarrow{\beta'} B$ of $\varphi$ is computable.*

The proof is dual to the proof of Proposition IV.1.8.

[4] $\mathrm{Hom}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}(A, B)$ is a set, not a setoid, i.e., two morphisms are congruent if they are equal.

[5] The terms arrow and reversed arrow of a morphism $\varphi \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}$ are dependent on the choice of the representative of $\varphi$. Every time we the use terms arrow and reversed arrow, we assume a normalized representative is fixed.

**Definition 1.15.** Let $\mathcal{A}$ be an abelian category and $\varphi \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}$. The generalized morphism $\varphi$ is called **honest** if $\mathrm{ReversedArrow}(\varphi)$ is an isomorphism in $\mathcal{A}$. We define

$$\mathrm{HonestRepresentative}(\varphi) := \mathrm{ReversedArrow}(\varphi)^{-1}\,\mathrm{Arrow}(\varphi)$$

to be the **honest representative** of $\varphi$.

The category $\mathcal{A}$ embeds naturally into $\mathrm{G}^{\mathrm{S}}(\mathcal{A})$:

**Definition 1.16.** There is an embedding

$$\mathrm{F}^{\mathrm{S}} : \mathcal{A} \to \mathrm{G}^{\mathrm{S}}(\mathcal{A}).$$

defined as follows: For $A \in \mathrm{Obj}_{\mathcal{A}}$ set $\mathrm{F}^{\mathrm{S}}(A) := A$ and for a morphism $\varphi \in \mathrm{Mor}_{\mathcal{A}}$

$$\mathrm{Arrow}\left(\mathrm{F}^{\mathrm{S}}(\varphi)\right) := \varphi,$$
$$\mathrm{ReversedArrow}\left(\mathrm{F}^{\mathrm{S}}(\varphi)\right) := \mathrm{IdentityMorphism}_{\mathcal{A}}(\mathrm{Source}(\varphi)).$$

We call $\mathrm{F}^{\mathrm{S}}(\varphi)$ the **corresponding honest span** of $\varphi$.

**1.d. Generalized morphisms by 3-arrows.** The third data structure for generalized morphisms consists of three morphisms from the underlying category $\mathcal{A}$ instead of two.

**Definition 1.17** (Data structure for 3-arrow generalized morphisms)**.** Let $\mathcal{A}$ be an abelian category. A **generalized morphism by 3-arrows** $\varphi$ with source $A$ and range $B$ in $\mathcal{A}$ is an equivalence class of tuples of three morphisms

$$\iota : A' \to A,$$
$$\alpha : A' \to B'',$$
$$\pi : B \to B'',$$

where $\iota$ is called the **source aid**, $\pi$ is called the **range aid**, and $\alpha$ is called the **arrow**. We write

$$\mathrm{SourceAid}(\varphi) := \iota,$$
$$\mathrm{RangeAid}(\varphi) := \pi,$$
$$\mathrm{Arrow}(\varphi) := \alpha.$$

$$\begin{tikzcd}
A \arrow[r, dashed] & B \arrow[d, "\mathrm{RangeAid}(\varphi)"] \\
A' \arrow[u, "\mathrm{SourceAid}(\varphi)"] \arrow[r, "\mathrm{Arrow}(\varphi)"'] & B''
\end{tikzcd}$$

We now define the equivalence relation for generalized morphisms by 3-arrows. To do this, we define a normalized generalized morphism by 3-arrows, then give the normalization algorithm. At last we show how to compare two normalized generalized morphisms by 3-arrows.

**Definition 1.18** (Normalized 3-arrow generalized morphism)**.** Let $\varphi$ be a 3-arrow generalized morphism in $\mathcal{A}$. We call $\varphi$ **normalized** if $\mathrm{SourceAid}\,(\varphi)$ is an $\mathcal{A}$-monomorphism and $\mathrm{RangeAid}\,(\varphi)$ is an $\mathcal{A}$-epimorphism. The source aid can then be seen as an $\mathcal{A}$-subobject of the source of $\varphi$, and the range aid as an $\mathcal{A}$-quotient object of the range of $\varphi$.

Remark 1.19. A generalized morphism $\varphi : A \to B$ can be interpreted as a morphism from an $\mathcal{A}$-subobject of $A$ to an $\mathcal{A}$-quotient object of $B$.

**Algorithm 1.20** (Normalization)**.** Let $\varphi : A \to B$ be a generalized morphism by 3-arrows in $\mathcal{A}$.

$$\begin{tikzcd}
A \arrow[r, dashed, "\varphi"] & B \arrow[d, "\mathrm{RangeAid}\,(\varphi)"] \\
A' \arrow[u, "\mathrm{SourceAid}\,(\varphi)"] \arrow[r, "\mathrm{Arrow}\,(\varphi)"'] & B''
\end{tikzcd}$$

The following algorithm computes a normalized representative of $\varphi$, i.e., three morphisms of the form

$$\begin{tikzcd}
A \arrow[r, dashed] & B \arrow[d, two heads, "\beta"] \\
Y \arrow[u, hook, "\alpha"] \arrow[r] & X'
\end{tikzcd}$$

such that $\alpha$ is a monomorphism, $\beta$ an epimorphism, and the 3-arrow represents the generalized morphism $\varphi$.

We start by computing the pushout of the source aid and the arrow of $\varphi$:

$$\iota_1 := \mathrm{InjectionOfCofactorOfPushout}_{\mathcal{A}}\left(\left(\mathrm{SourceAid}\,(\varphi)\,, \mathrm{Arrow}\,(\varphi)\right), 1\right),$$
$$\iota_2 := \mathrm{InjectionOfCofactorOfPushout}_{\mathcal{A}}\left(\left(\mathrm{SourceAid}\,(\varphi)\,, \mathrm{Arrow}\,(\varphi)\right), 2\right).$$

$$\begin{tikzcd}
A \arrow[rr, dashed, "\varphi"] \arrow[dr, "\iota_1"] & & B \arrow[dd, "\mathrm{RangeAid}\left(\varphi\right)"] \\
& X & \\
A' \arrow[uu, "\mathrm{SourceAid}\left(\varphi\right)"] \arrow[rr, "\mathrm{Arrow}\left(\varphi\right)"'] & & B'' \arrow[ul, "\iota_2"']
\end{tikzcd}$$

Now we compose the range aid of $\varphi$ with the second cofactor injection $\iota$ of the pushout, i.e.,

$$\gamma := \mathrm{PreCompose}_{\mathcal{A}}\left(\mathrm{RangeAid}\left(\varphi\right), \iota\right).$$

and then compute

$$\begin{aligned}
\gamma_1 &:= \mathrm{CoastrictionToImage}_{\mathcal{A}}\left(\gamma\right), \\
\gamma_2 &:= \mathrm{ImageEmbedding}_{\mathcal{A}}\left(\gamma\right).
\end{aligned}$$

$$\begin{tikzcd}
A \arrow[rrr, dashed, "\varphi"] \arrow[dr, "\iota_1"] & & & B \arrow[dl, two heads, "\gamma_1"'] \arrow[dd, "\mathrm{RangeAid}\left(\varphi\right)"] \\
& X & X' \arrow[l, hook', "\gamma_2"'] & \\
A' \arrow[uu, "\mathrm{SourceAid}\left(\varphi\right)"] \arrow[rrr, "\mathrm{Arrow}\left(\varphi\right)"'] & & & B'' \arrow[ull, "\iota_2"']
\end{tikzcd}$$

Now we compute the pullback of the image embedding $\gamma_2$ and the first injection $\iota_1$, i.e.,

$$\begin{aligned}
\delta_1 &:= \mathrm{ProjectionInFactorOfFiberProduct}_{\mathcal{A}}\left(\left(\iota_1, \gamma_2\right), 1\right), \\
\delta_2 &:= \mathrm{ProjectionInFactorOfFiberProduct}_{\mathcal{A}}\left(\left(\iota_1, \gamma_2\right), 2\right).
\end{aligned}$$

$$\begin{tikzcd}
A \arrow[rrr, dashed, "\varphi"] \arrow[dr, "\iota_1"] & & & B \arrow[ddl, two heads, "\gamma_1"'] \arrow[ddd, "\operatorname{RangeAid}(\varphi)"] \\
& & Y \arrow[ull, hook, "\delta_1"'] \arrow[d, "\delta_2"'] & \\
& X & X' \arrow[l, hook', "\gamma_2"'] & \\
A' \arrow[uuu, "\operatorname{SourceAid}(\varphi)"] \arrow[rrr, "\operatorname{Arrow}(\varphi)"'] & & & B'' \arrow[ull, "\iota_2"']
\end{tikzcd}$$

The normalized generalized morphism $\widetilde{\varphi}$ is now given by

$$\begin{tikzcd}
A \arrow[r, dashed, "\widetilde{\varphi}"] & B \arrow[d, two heads, "\gamma_2"] \\
Y \arrow[u, hook, "\delta_1"] \arrow[r, "\delta_2"'] & X'
\end{tikzcd}$$

We can now define the equivalence relation for 3-arrow generalized morphisms.

**Definition 1.21.** Let

$$\begin{tikzcd}
A \arrow[r, dashed, "\varphi_1"] & B \arrow[d, "\pi_1"] & A \arrow[r, dashed, "\varphi_2"] & B \arrow[d, two heads, "\pi_2"] \\
A_1' \arrow[u, "\iota_1"] \arrow[r, "\alpha_1"'] & B_1'' & A_2' \arrow[u, "\iota_2"] \arrow[r, "\alpha_2"'] & B_2''
\end{tikzcd}$$

be two 3-tuples as in Definition IV.1.17. Then the tuples $\varphi_1$ and $\varphi_2$ represent the same 3-arrow generalized morphism if for their normalizations $\widetilde{\varphi}_1$ and $\widetilde{\varphi}_2$ the following holds:

(1) The source aids $\operatorname{SourceAid}(\widetilde{\varphi}_1)$ and $\operatorname{SourceAid}(\widetilde{\varphi}_2)$ are equivalent as subobjects of $A$;
(2) The range aids $\operatorname{RangeAid}(\widetilde{\varphi}_1)$ and $\operatorname{RangeAid}(\widetilde{\varphi}_2)$ are equivalent as factor objects of $B$.

$$\begin{tikzcd}
A'' \arrow[r] \arrow[dd, bend right=40, leftrightarrow, "\wr"'] & B'' \arrow[dd, bend left=40, leftrightarrow, "\wr"] \\
A \arrow[u, hook] \arrow[d, hook] \arrow[r, dashed, bend left, "\widetilde{\varphi}_1"] \arrow[r, dashed, bend right, "\widetilde{\varphi_2}"'] & B \arrow[u, two heads] \arrow[d, two heads] \\
A' \arrow[r] & B'
\end{tikzcd}$$

**Definition 1.22.** Let $\mathcal{A}$ be an abelian category. The category $\mathrm{G}^{\mathrm{T}}(\mathcal{A})$ of generalized morphisms by 3-arrows in $\mathcal{A}$ is the category with

(1) $\mathrm{Obj}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A})} := \mathrm{Obj}_{\mathcal{A}}$ and
(2) $A, B \in \mathrm{Obj}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A})}$ set[6]

$$\mathrm{Hom}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A})}(A, B) := \{\varphi :\ A \to B \mid \varphi \text{ is a 3-arrow generalized morphism in } \mathcal{A}\}$$

Let $A \in \mathrm{Obj}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A})}$. The identity morphism $\mathrm{IdentityMorphism}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A})}(A)$ is the generalized morphism represented by the triple where all three morphisms, i.e., SourceAid, RangeAid, and Arrow are the identity morphism of $A$ (viewed as object in $\mathcal{A}$).

Let $\varphi : A \to B$ and $\psi : B \to C$ be two generalized morphisms in $\mathcal{A}$. We define their composition $\varphi\psi$ as follows: Let $\varphi$ and $\psi$ represented by the following triples:

$$\begin{tikzcd}
A \arrow[r, dashed, "\varphi"] & B \arrow[r, equal] \arrow[d, "\mathrm{RangeAid}(\varphi)"] & B \arrow[r, dashed, "\psi"] & C \arrow[d, "\mathrm{RangeAid}(\psi)"] \\
A' \arrow[u, "\mathrm{SourceAid}(\varphi)"] \arrow[r, "\mathrm{Arrow}(\varphi)"'] & B'' & B' \arrow[u, "\mathrm{SourceAid}(\psi)"] \arrow[r, "\mathrm{Arrow}(\psi)"'] & C''
\end{tikzcd}$$

We compose $\mathrm{SourceAid}(\psi)$ and $\mathrm{RangeAid}(\varphi)$ and get a morphism $\alpha : B' \to B''$, i.e.,

$$\alpha := \mathrm{PreCompose}_{\mathcal{A}}(\mathrm{SourceAid}(\psi), \mathrm{RangeAid}(\varphi)).$$

$$\begin{tikzcd}
A \arrow[r, dashed, "\varphi"] & B \arrow[r, equal] \arrow[d] & B \arrow[r, dashed, "\psi"] & C \arrow[d] \\
A' \arrow[u] \arrow[r] & B'' & B' \arrow[u] \arrow[l, "\alpha"'] \arrow[r] & C'
\end{tikzcd}$$

[6] $\mathrm{Hom}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A})}(A, B)$ is a set, not a setoid, i.e., two morphisms are congruent if they are equal.

The next step is an epi-mono-factorization of $\alpha$, resulting in two morphisms $\pi : B' \twoheadrightarrow X$ and $\iota : X \hookrightarrow B''$, i.e.,

$$(\pi, \iota) := \mathrm{EpiMonoFactorization}_{\mathcal{A}}(\alpha).$$

$$\begin{tikzcd}
A \arrow[r, dashed, "\varphi"] & B \arrow[rr, equal] \arrow[d] & & B \arrow[r, dashed, "\psi"] & C \arrow[d] \\
A' \arrow[u] \arrow[r, "\mathrm{Arrow}(\varphi)"'] & B'' & & B' \arrow[ll, "\alpha"'] \arrow[u] \arrow[r, "\mathrm{Arrow}(\psi)"'] \arrow[dl, two heads, "\pi"] & C' \\
& & X \arrow[ul, hook, "\iota"] & &
\end{tikzcd}$$

At last we compute $\mathrm{FiberProduct}_{\mathcal{A}}(\mathrm{Arrow}(\varphi), \iota)$ and $\mathrm{Pushout}_{\mathcal{A}}(\pi, \mathrm{Arrow}(\psi))$ and their corresponding projections and injections and get a full rectangle. We define

$$\begin{aligned}
\gamma_i &:= \mathrm{ProjectionInFactorOfFiberProduct}_{\mathcal{A}}((\mathrm{Arrow}(\varphi), \pi), i), \\
\delta_i &:= \mathrm{InjectionOfCofactorOfPushout}_{\mathcal{A}}((\iota, \mathrm{Arrow}(\psi)), i),
\end{aligned}$$

for $i = 1, 2$.

$$\begin{tikzcd}
A \arrow[r, dashed, "\varphi"] & B \arrow[rr, equal] \arrow[d] & & B \arrow[r, dashed, "\psi"] & C \arrow[d, "\mathrm{RangeAid}(\psi)"] \\
A' \arrow[u, "\mathrm{SourceAid}(\varphi)"] \arrow[r] & B'' & & B' \arrow[ll, "\alpha"'] \arrow[u] \arrow[r] \arrow[dl, hook, "\iota"] & C' \arrow[d, "\delta_2"] \\
A'' \arrow[u, "\gamma_1"] \arrow[rr, "\gamma_2"'] & & X \arrow[ul, two heads, "\pi"] \arrow[rr, "\delta_1"'] & & C''
\end{tikzcd}$$

Now we compose the borders of the rectangle, i.e,

$$\begin{aligned}
\epsilon &:= \mathrm{PreCompose}_{\mathcal{A}}(\gamma_1, \mathrm{SourceAid}(\varphi)), \\
\eta &:= \mathrm{PreCompose}_{\mathcal{A}}(\gamma_2, \delta_1), \\
\kappa &:= \mathrm{PreCompose}_{\mathcal{A}}(\delta_2, \mathrm{RangeAid}(\psi)),
\end{aligned}$$

and get the composition of $\varphi$ and $\psi$ as

$$\begin{tikzcd}
A \arrow[r, dashed, "\varphi\psi"] & B \arrow[d, "\kappa"] \\
A'' \arrow[u, "\epsilon"] \arrow[r, "\eta"'] & C''
\end{tikzcd}$$

REMARK 1.23. We summarize the steps for the composition $\varphi\psi$ of two 3-arrow generalized morphisms $\varphi$ and $\psi$:

(1) $\alpha := \mathrm{PreCompose}_{\mathcal{A}}(\mathrm{SourceAid}(\psi), \mathrm{RangeAid}(\varphi))$
(2) $(\iota, \pi) := \mathrm{EpiMonoFactorization}_{\mathcal{A}}(\alpha)$
(3)

$$\mathrm{SourceAid}(\varphi\psi) := \mathrm{PreCompose}_{\mathcal{A}}(\mathrm{ProjectionInFactorOfFiberProduct}_{\mathcal{A}}((\mathrm{Arrow}(\varphi), \pi), 1), \mathrm{SourceAid}(\varphi))$$

(4)

$$\mathrm{RangeAid}(\varphi\psi) := \mathrm{PreCompose}_{\mathcal{A}}(\mathrm{RangeAid}(\psi), \mathrm{InjectionOfCofactorOfPushout}_{\mathcal{A}}((\iota, \mathrm{Arrow}(\psi)), 1))$$

(5)

$$\mathrm{Arrow}(\varphi\psi) := \mathrm{PreCompose}_{\mathcal{A}}(\mathrm{ProjectionInFactorOfFiberProduct}_{\mathcal{A}}((\mathrm{Arrow}(\varphi), \pi), 2), \mathrm{InjectionOfCofactorOfPushout}_{\mathcal{A}}((\iota, \mathrm{Arrow}(\psi)), 1)).$$

For a proof that this category is well-defined and the composition is compatible with the equivalence relation on the Hom-sets, see [**Pos17**, Subsection II.1.2].

**Definition 1.24.** Let $\mathcal{A}$ be a category and $\varphi \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A})}$. The generalized morphism $\varphi$ is called **honest** if $\mathrm{SourceAid}(\varphi)$ and $\mathrm{RangeAid}(\varphi)$ are isomorphisms. We define

$$\mathrm{HonestRepresentative}(\varphi) := \mathrm{SourceAid}(\varphi)^{-1}\,\mathrm{Arrow}(\varphi)\,\mathrm{RangeAid}(\varphi)^{-1}$$

to be the **honest representative** of $\varphi$.

The category $\mathcal{A}$ embeds naturally into $\mathrm{G}^{\mathrm{T}}(\mathcal{A})$.

**Definition 1.25.** There is an embedding

$$\mathrm{F}^{\mathrm{T}} : \mathcal{A} \to \mathrm{G}^{\mathrm{T}}(\mathcal{A})$$

defined as follows: For $A \in \mathrm{Obj}_{\mathcal{A}}$ set $\mathrm{F}^{\mathrm{T}}(A) := A$ and for a morphism $\varphi \in \mathrm{Mor}_{\mathcal{A}}$

$$\mathrm{Arrow}\left(\mathrm{F}^{\mathrm{T}}(\varphi)\right) := \varphi,$$
$$\mathrm{SourceAid}\left(\mathrm{F}^{\mathrm{T}}(\varphi)\right) := \mathrm{IdentityMorphism}_{\mathcal{A}}(\mathrm{Source}(\varphi)),$$
$$\mathrm{RangeAid}\left(\mathrm{F}^{\mathrm{T}}(\varphi)\right) := \mathrm{IdentityMorphism}_{\mathcal{A}}(\mathrm{Range}(\varphi)).$$

We call $\mathrm{F}^{\mathrm{T}}(\varphi)$ the **corresponding honest 3-arrow** of $\varphi$.

**1.e. Conversion between different types of generalized morphisms.** We now show how to relate the three different kinds of generalized morphism categories. We give the conversion functors between all three types of generalized morphisms, which are equivalences of categories. For the proof of correctness see [Pos17, Subsection II.1.4].

**Definition 1.26.** Let $\mathcal{A}$ be an abelian category. The **conversion functor from cospans to spans** $\mathrm{C}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A}),\mathrm{G}^{\mathrm{C}}(\mathcal{A})} : \mathrm{G}^{\mathrm{C}}(\mathcal{A}) \to \mathrm{G}^{\mathrm{S}}(\mathcal{A})$ is defined as follows:

(1) For an object $A \in \mathrm{Obj}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A})}$ set $\mathrm{C}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A}),\mathrm{G}^{\mathrm{C}}(\mathcal{A})}(A) := A$ (recall, the object classes are the same).
(2) Let $\varphi \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A})}$ and set

$$\pi_i := \mathrm{ProjectionInFactorOfFiberProduct}_{\mathcal{A}}\left(\left(\mathrm{Arrow}(\varphi), \mathrm{ReversedArrow}(\varphi)\right), i\right),$$

$i = 1, 2$.

$$\begin{tikzcd} & P & \\ A \arrow[ur, "\mathrm{Arrow}(\varphi)"] \arrow[rr, dashed, "\varphi"] & & B \arrow[ul, "\mathrm{ReversedArrow}(\varphi)"'] \\ & F \arrow[ul, "\pi_1"] \arrow[ur, "\pi_2"'] & \end{tikzcd}$$

Then the span below the dashed arrow represents $\mathrm{C}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A}),\mathrm{G}^{\mathrm{C}}(\mathcal{A})}(\varphi)$, i.e.,

$$\mathrm{Arrow}\left(\mathrm{C}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A}),\mathrm{G}^{\mathrm{C}}(\mathcal{A})}(\varphi)\right) := \pi_2,$$
$$\mathrm{ReversedArrow}\left(\mathrm{C}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A}),\mathrm{G}^{\mathrm{C}}(\mathcal{A})}(\varphi)\right) := \pi_1.$$

**Definition 1.27.** Let $\mathcal{A}$ be an abelian category. The **conversion functor from spans to cospans** $\mathrm{C}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A}),\mathrm{G}^{\mathrm{S}}(\mathcal{A})} : \mathrm{G}^{\mathrm{S}}(\mathcal{A}) \to \mathrm{G}^{\mathrm{C}}(\mathcal{A})$ is defined as follows:

(1) For an object $A \in \mathrm{Obj}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}$ set $\mathrm{C}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A}),\mathrm{G}^{\mathrm{S}}(\mathcal{A})}(A) := A$.
(2) Let $\varphi \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}$ and set

$$\iota_i := \mathrm{InjectionOfCofactorOfPushout}_{\mathcal{A}}\left(\left(\mathrm{ReversedArrow}(\varphi), \mathrm{Arrow}(\varphi)\right), i\right),$$

$i = 1, 2$.

$$\begin{tikzcd}
 & P & \\
A \arrow[ur, "\iota_1"] \arrow[rr, dashed, "\varphi"] & & B \arrow[ul, "\iota_2"'] \\
 & F \arrow[ul, "\mathrm{ReversedArrow}(\varphi)"] \arrow[ur, "\mathrm{Arrow}(\varphi)"'] &
\end{tikzcd}$$

Then the cospan above the dashed arrow represents $\mathrm{C}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A}),\mathrm{G}^{\mathrm{S}}(\mathcal{A})}(\varphi)$, i.e.,

$$\mathrm{Arrow}\left(\mathrm{C}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A}),\mathrm{G}^{\mathrm{S}}(\mathcal{A})}(\varphi)\right) := \iota_1,$$
$$\mathrm{ReversedArrow}\left(\mathrm{C}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A}),\mathrm{G}^{\mathrm{S}}(\mathcal{A})}(\varphi)\right) := \iota_2.$$

**Definition 1.28.** Let $\mathcal{A}$ be an abelian category. The **conversion functor from cospans to 3-arrows** $\mathrm{C}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A}),\mathrm{G}^{\mathrm{C}}(\mathcal{A})} : \mathrm{G}^{\mathrm{C}}(\mathcal{A}) \to \mathrm{G}^{\mathrm{T}}(\mathcal{A})$ is defined as follows:

(1) For an object $A \in \mathrm{Obj}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A})}$ set $\mathrm{C}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A}),\mathrm{G}^{\mathrm{C}}(\mathcal{A})}(A) := A$.

(2) For a morphism $\varphi \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A})}$ with

$$\varphi = (\varphi_1 : A \to B'', \varphi_2 : B \to B'')$$

the 3-arrow generalized morphism $\mathrm{C}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A}),\mathrm{G}^{\mathrm{C}}(\mathcal{A})}(\varphi)$ is represented by

$$\begin{tikzcd}
A \arrow[r, dashed, "\mathrm{C}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A}),\mathrm{G}^{\mathrm{C}}(\mathcal{A})}(\varphi)"] & B \arrow[d, "\varphi_2"] \\
A \arrow[u, "\mathrm{IdentityMorphism}_{\mathcal{A}}(A)"] \arrow[ur, dotted, "\varphi"] \arrow[r, "\varphi_1"'] & B''
\end{tikzcd}$$

i.e.,

$$\mathrm{SourceAid}\left(\mathrm{C}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A}),\mathrm{G}^{\mathrm{C}}(\mathcal{A})}(\varphi)\right) := \mathrm{IdentityMorphism}_{\mathcal{A}}(A),$$
$$\mathrm{RangeAid}\left(\mathrm{C}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A}),\mathrm{G}^{\mathrm{C}}(\mathcal{A})}(\varphi)\right) := \varphi_2,$$
$$\mathrm{Arrow}\left(\mathrm{C}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A}),\mathrm{G}^{\mathrm{C}}(\mathcal{A})}(\varphi)\right) := \varphi_1.$$

**Definition 1.29.** Let $\mathcal{A}$ be an abelian category. The **conversion functor from spans to 3-arrows** $\mathrm{C}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A}),\mathrm{G}^{\mathrm{S}}(\mathcal{A})} : \mathrm{G}^{\mathrm{S}}(\mathcal{A}) \to \mathrm{G}^{\mathrm{T}}(\mathcal{A})$ is defined as follows:

(1) For an object $A \in \mathrm{Obj}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}$ set $\mathrm{C}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A}),\mathrm{G}^{\mathrm{C}}(\mathcal{A})}(A) := A$.

(2) For a morphism $\varphi \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}$ with

$$\varphi = (\varphi_1 : A' \to A, \varphi_2 : A' \to B)$$

the 3-arrow generalized morphism $\mathrm{C}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A}),\mathrm{G}^{\mathrm{S}}(\mathcal{A})}(\varphi)$ is represented by

$$\begin{tikzcd}
A \arrow[r, dashed, "\mathrm{C}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A}),\mathrm{G}^{\mathrm{S}}(\mathcal{A})}(\varphi)"] \arrow[dr, dotted, "\varphi"'] & B \arrow[d, "\mathrm{IdentityMorphism}_{\mathcal{A}}(B)"] \\
A \arrow[u, "\varphi_1"] \arrow[r, "\varphi_2"'] & B''
\end{tikzcd}$$

i.e.,

$$\begin{aligned}
\mathrm{SourceAid}\left(\mathrm{C}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A}),\mathrm{G}^{\mathrm{S}}(\mathcal{A})}(\varphi)\right) &:= \varphi_1,\\
\mathrm{RangeAid}\left(\mathrm{C}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A}),\mathrm{G}^{\mathrm{S}}(\mathcal{A})}(\varphi)\right) &:= \mathrm{IdentityMorphism}_{\mathcal{A}}(B),\\
\mathrm{Arrow}\left(\mathrm{C}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A}),\mathrm{G}^{\mathrm{S}}(\mathcal{A})}(\varphi)\right) &:= \varphi_2.
\end{aligned}$$

**Definition 1.30.** Let $\mathcal{A}$ be an abelian category. The **conversion functor from 3-arrows to cospans** $\mathrm{C}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A}),\mathrm{G}^{\mathrm{T}}(\mathcal{A})} : \mathrm{G}^{\mathrm{T}}(\mathcal{A}) \to \mathrm{G}^{\mathrm{C}}(\mathcal{A})$ is defined as follows:

(1) For an object $A \in \mathrm{Obj}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A})}$ set $\mathrm{C}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A}),\mathrm{G}^{\mathrm{T}}(\mathcal{A})}(A) := A$.

(2) For a morphism $\varphi \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A})}$ which is represented by

$$\begin{tikzcd}
A \arrow[r, dashed, "\varphi"] & B \arrow[d, "\pi"] \\
A' \arrow[u, "\iota"] \arrow[r, "\alpha"'] & B''
\end{tikzcd}$$

we set $\mathrm{C}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A}),\mathrm{G}^{\mathrm{T}}(\mathcal{A})}(\varphi)$ to be the composition of the cospans

$$\begin{tikzcd}
 & A & & B'' & \\
A \arrow[ur, "\mathrm{IdentityMorphism}_{\mathcal{A}}(A)"] \arrow[rr, dashed] \arrow[rrrr, dashed, bend right, "\mathrm{C}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A}),\mathrm{G}^{\mathrm{T}}(\mathcal{A})}(\varphi)"'] & & A' \arrow[ul, "\iota"'] \arrow[ur, "\alpha"] \arrow[rr, dashed] & & B \arrow[ul, "\pi"']
\end{tikzcd}$$

i.e.,

$$\mathrm{Arrow}\left(\mathrm{C}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A}),\mathrm{G}^{\mathrm{T}}(\mathcal{A})}(\varphi)\right) := \mathrm{InjectionOfCofactorOfPushout}_{\mathcal{A}}\left((\iota,\alpha),1\right),$$

$$\begin{aligned}\mathrm{ReversedArrow}\left(\mathrm{C}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A}),\mathrm{G}^{\mathrm{T}}(\mathcal{A})}\left(\varphi\right)\right) := \mathrm{PreCompose}_{\mathcal{A}}\,(\pi,\\ \mathrm{InjectionOfCofactorOfPushout}_{\mathcal{A}}\left(\left(\iota,\alpha\right),2\right)).\end{aligned}$$

$$\begin{tikzcd}
 & & X & & \\
 & A \arrow[ur] & & B'' \arrow[ul] & \\
A \arrow[ur] & & A' \arrow[ul, thick] \arrow[ur, thick] & & B \arrow[ul, thick]
\end{tikzcd}$$

**Definition 1.31.** Let $\mathcal{A}$ be an abelian category. The **conversion functor from 3-arrows to spans** $\mathrm{C}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A}),\mathrm{G}^{\mathrm{T}}(\mathcal{A})} : \mathrm{G}^{\mathrm{T}}\left(\mathcal{A}\right) \to \mathrm{G}^{\mathrm{S}}\left(\mathcal{A}\right)$ is defined as follows:

(1) For an object $A \in \mathrm{Obj}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A})}$ set $\mathrm{C}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A}),\mathrm{G}^{\mathrm{T}}(\mathcal{A})}\left(A\right) := A$.
(2) For a morphism $\varphi \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A})}$ which is represented by

$$\begin{tikzcd}
A \arrow[r, dashed, "\varphi"] & B \arrow[d, "\pi"] \\
A' \arrow[u, "\iota"] \arrow[r, "\alpha"'] & B''
\end{tikzcd}$$

we set $\mathrm{C}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A}),\mathrm{G}^{\mathrm{T}}(\mathcal{A})}\left(\varphi\right)$ to be the composition of the spans

$$\begin{tikzcd}
A \arrow[rr, dashed] \arrow[rrrr, dashed, bend left, "\mathrm{C}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A}),\mathrm{G}^{\mathrm{T}}(\mathcal{A})}\left(\varphi\right)"] & & B'' \arrow[rr, dashed] & & B \\
 & A' \arrow[ul, "\iota"] \arrow[ur, "\alpha"'] & & B \arrow[ul, "\pi"] \arrow[ur, "\mathrm{IdentityMorphism}_{\mathcal{A}}\left(B\right)"'] &
\end{tikzcd}$$

i.e.,

$$\mathrm{Arrow}\left(\mathrm{C}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A}),\mathrm{G}^{\mathrm{T}}(\mathcal{A})}\left(\varphi\right)\right) := \mathrm{ProjectionInFactorOfFiberProduct}_{\mathcal{A}}\left(\left(\pi,\alpha\right),2\right),$$

$$\begin{aligned}\mathrm{ReversedArrow}\left(\mathrm{C}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A}),\mathrm{G}^{\mathrm{T}}(\mathcal{A})}(\varphi)\right) := \mathrm{PreCompose}_{\mathcal{A}}(\iota,&\\ \mathrm{ProjectionInFactorOfFiberProduct}_{\mathcal{A}}((\pi,\alpha),1)).&\end{aligned}$$

## 2. Structure of the category of generalized morphisms

As already seen from the constructions, all three types of generalized morphism categories are computable, as long as their underlying category $\mathcal{A}$ is computable abelian. We are going to formulate this as a theorem.

**Theorem 2.1.** *Let $\mathcal{A}$ be computable abelian. Then the categories $\mathrm{G}^{\mathrm{S}}(\mathcal{A})$, $\mathrm{G}^{\mathrm{C}}(\mathcal{A})$, and $\mathrm{G}^{\mathrm{T}}(\mathcal{A})$ are computable.*

Generalized morphisms are not preadditive, but they fulfill a property which is close.

**Definition 2.2.** Let $\mathcal{A}$ be a category. $\mathcal{A}$ is called **enriched over a commutative regular semigroup** if for any two objects

1. there is a commutative addition in $\mathrm{Hom}_{\mathcal{A}}(A,B)$;
2. there exists a morphism $0 \in \mathrm{Hom}_{\mathcal{A}}(A,B)$ such that for every $\varphi \in \mathrm{Hom}_{\mathcal{A}}(A,B)$ we have $0+\varphi = \varphi + 0 = \varphi$;
3. for every $\varphi \in \mathrm{Hom}_{\mathcal{A}}(A,B)$ there exists a morphism $-\varphi \in \mathrm{Hom}_{\mathcal{A}}(A,B)$ such that $\varphi + (-\varphi) + \varphi = \varphi$ and $(-\varphi)+\varphi+(-\varphi) = (-\varphi)$.

Remark 2.3. We use the same names ZeroMorphism and AdditiveInverse as for the preadditive case, since a preadditive category is a special case of a category enriched over a commutative regular semigroup.

**Proposition 2.4.** *Let $\mathcal{A}$ be computable abelian. Then the category of generalized morphisms by spans $\mathrm{G}^{\mathrm{S}}(\mathcal{A})$ is computable enriched over a commutative regular semigroup.*

The enrichment algorithms implemented in Cap for $\mathrm{G}^{\mathrm{S}}(\mathcal{A})$ can be found in Appendix F.7.

Proof. We are going to give the enrichment constructions which will turn out to be computable.

1. Let $A, B \in \mathrm{Obj}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}$ and $\varphi := \mathrm{ZeroMorphism}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}(A,B)$. Then $\varphi$ is defined via
$$\mathrm{Arrow}(\varphi) := \mathrm{ZeroMorphism}_{\mathcal{A}}(A,B),$$
$$\mathrm{ReversedArrow}(\varphi) := \mathrm{IdentityMorphism}_{\mathcal{A}}(A).$$
2. Let $\varphi,\psi \in \mathrm{Hom}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}(A,B)$ for two objects $A,B \in \mathrm{Obj}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}$. To compute the sum $\mathrm{AdditionForMorphisms}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}(\varphi,\psi) = \varphi+\psi$ one first computes the fiber product of the two reversed arrows:
$$\begin{aligned}\rho_1 := \mathrm{ProjectionInFactorOfFiberProduct}_{\mathcal{A}}((\mathrm{ReversedArrow}(\varphi),&\\ \mathrm{ReversedArrow}(\psi)),1),&\\ \rho_2 := \mathrm{ProjectionInFactorOfFiberProduct}_{\mathcal{A}}((\mathrm{ReversedArrow}(\varphi),&\\ \mathrm{ReversedArrow}(\psi)),2).&\end{aligned}$$

Then the sum is represented by

$$\begin{aligned}\operatorname{Arrow}(\varphi+\psi) := \operatorname{AdditionForMorphisms}_{\mathcal{A}}(&\operatorname{PreCompose}_{\mathcal{A}}(\rho_1, \operatorname{Arrow}(\varphi))\\ &\operatorname{PreCompose}_{\mathcal{A}}(\rho_2, \operatorname{Arrow}(\psi))),\end{aligned}$$

$$\operatorname{ReversedArrow}(\varphi+\psi) := \operatorname{PreCompose}_{\mathcal{A}}(\rho_1, \operatorname{ReversedArrow}(\varphi)).$$

The commutativity of the addition and the fact that $0+\varphi = \varphi+0 = \varphi$ follow from the construction.

(3) Let $\varphi \in \operatorname{Mor}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}$. Then the AdditiveInverse $(\varphi) = -\varphi$ is defined as follows:

$$\operatorname{Arrow}(-\varphi) := -\operatorname{Arrow}(\varphi),$$
$$\operatorname{ReversedArrow}(-\varphi) := \operatorname{ReversedArrow}(\varphi).$$

The algorithms to construct the zero morphism, the addition, and the additive inverse in $\mathrm{G}^{\mathrm{S}}(\mathcal{A})$ were completely based on constructions from the computable category $\mathcal{A}$. Therefore these three constructions are computable in $\mathrm{G}^{\mathrm{S}}(\mathcal{A})$. ☐

Remark 2.5. Let $A, B \in \operatorname{Obj}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}$. Then we have

$$\operatorname{ZeroMorphism}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}(A,B) = \mathrm{F}^{\mathrm{S}}(\operatorname{ZeroMorphism}_{\mathcal{A}}(A,B)).$$

**Proposition 2.6.** *Let $\mathcal{A}$ be computable abelian. Then the category of generalized morphisms by cospans $\mathrm{G}^{\mathrm{C}}(\mathcal{A})$ is computable enriched over a commutative regular semigroup.*

The enrichment algorithms implemented in Cap for $\mathrm{G}^{\mathrm{S}}(\mathcal{A})$ can be found in Appendix F.6.

Proof. We are going to give the enrichment constructions which will turn out to be computable.

(1) Let $A, B \in \operatorname{Obj}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A})}$ and $\varphi := \operatorname{ZeroMorphism}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}(A,B)$. Then $\varphi$ is defined via

$$\operatorname{Arrow}(\varphi) := \operatorname{ZeroMorphism}_{\mathcal{A}}(A,B),$$
$$\operatorname{ReversedArrow}(\varphi) := \operatorname{IdentityMorphism}_{\mathcal{A}}(B).$$

(2) Let $\varphi, \psi \in \operatorname{Hom}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A})}(A,B)$ for two objects $A, B \in \operatorname{Obj}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A})}$. To compute the sum $\operatorname{AdditionForMorphisms}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A})}(\varphi,\psi) = \varphi+\psi$ one first computes the pushout of the reversed arrows of $\varphi$ and $\psi$:

$$\begin{aligned}\rho_1 := \operatorname{InjectionOfCofactorOfPushout}_{\mathcal{A}}((&\operatorname{ReversedArrow}(\varphi),\\ &\operatorname{ReversedArrow}(\psi)),1),\\ \rho_2 := \operatorname{InjectionOfCofactorOfPushout}_{\mathcal{A}}((&\operatorname{ReversedArrow}(\varphi),\\ &\operatorname{ReversedArrow}(\psi)),2).\end{aligned}$$

Then the sum can be described as

$$\begin{aligned}\operatorname{Arrow}(\varphi+\psi) := \operatorname{AdditionForMorphisms}_{\mathcal{A}}(&\operatorname{PreCompose}_{\mathcal{A}}(\operatorname{Arrow}(\varphi), \rho_1)\\ &\operatorname{PreCompose}_{\mathcal{A}}(\operatorname{Arrow}(\psi), \rho_2)),\end{aligned}$$

$$\text{ReversedArrow}\,(\varphi+\psi) := \text{PreCompose}_{\mathcal{A}}\,(\text{ReversedArrow}\,(\varphi)\,,\rho_1)\,.$$

The commutativity of the addition and the fact that $0+\varphi=\varphi+0=\varphi$ follow from the construction.

(3) Let $\varphi \in \text{Mor}_{\text{G}^{\text{S}}(\mathcal{A})}$. Then the AdditiveInverse $(\varphi) = -\varphi$ is defined as follows:

$$\begin{aligned}\text{Arrow}\,(-\varphi) &:= -\,\text{Arrow}\,(\varphi)\,,\\ \text{ReversedArrow}\,(-\varphi) &:= \text{ReversedArrow}\,(\varphi)\,.\end{aligned}$$

The algorithms to construct the zero morphism, the addition, and the additive inverse in $\text{G}^{\text{C}}\,(\mathcal{A})$ were completely based on constructions from the computable category $\mathcal{A}$. Therefore these three constructions are computable in $\text{G}^{\text{C}}\,(\mathcal{A})$. □

**Proposition 2.7.** *Let $\mathcal{A}$ be computable abelian. Then the category of generalized morphisms by 3-arrows $\text{G}^{\text{T}}\,(\mathcal{A})$ is computable enriched over a commutative regular semigroup.*

The enrichment algorithms implemented in CAP for $\text{G}^{\text{T}}\,(\mathcal{A})$ can be found in Appendix F.8.

PROOF. We give the enrichment constructions. The proofs can be found in [**BLH14b**, Thm. 2.7]. Let $A, B \in \text{Obj}_{\text{G}^{\text{T}}(\mathcal{A})}$. Then we define

$$\text{ZeroMorphism}_{\text{G}^{\text{T}}(\mathcal{A})}\,(A,B) = \text{F}^{\text{T}}\,(\text{ZeroMorphism}_{\mathcal{A}}\,(A,B))\,.$$

To compute the sum of $\varphi, \psi \in \text{Hom}_{\text{G}^{\text{T}}(\mathcal{A})}\,(A,B)$ we first define

$$\begin{aligned}\iota_i &:= \text{InjectionOfCofactorOfPushout}_{\mathcal{A}}\,((\text{RangeAid}\,(\varphi)\,,\text{RangeAid}\,(\psi))\,,i)\,,\\ \pi_i &:= \text{ProjectionInFactorOfFiberProduct}_{\mathcal{A}}\,((\text{SourceAid}\,(\varphi)\,,\text{SourceAid}\,(\psi))\,i,)\,.\end{aligned}$$

Then we represent the sum $\varphi+\psi$ by

$$\begin{aligned}\text{SourceAid}\,(\varphi+\psi) &:= \text{PreCompose}_{\mathcal{A}}\,(\pi_i, \text{SourceAid}\,(\varphi))\,,\\ \text{RangeAid}\,(\varphi+\psi) &:= \text{PreCompose}_{\mathcal{A}}\,(\text{RangeAid}\,(\varphi)\,,\iota_i)\,,\\ \text{Arrow}\,(\varphi+\psi) &:= \pi_1\,\text{Arrow}\,(\varphi)\,\iota_1 + \pi_2\,\text{Arrow}\,(\psi)\,\iota_2.\end{aligned}$$

The additive inverse of a morphism $\varphi \in \text{Mor}_{\text{G}^{\text{T}}(\mathcal{A})}$ is defined as

$$\begin{aligned}\text{SourceAid}\,(-\varphi) &:= \text{SourceAid}\,(\varphi)\,,\\ \text{RangeAid}\,(-\varphi) &:= \text{RangeAid}\,(\varphi)\,,\\ \text{Arrow}\,(-\varphi) &:= -\,\text{Arrow}\,(\varphi)\,.\end{aligned}$$

□

## 3. Generalized and pseudo-inverse

In this section we emphasize why generalized morphisms are useful for computations in abelian categories. Generalized morphisms offer the possibility to compute a one-sided inverse of a non-split mono- or epimorphism in any abelian category. The result of a composition with such a split can then be recovered from the resulting generalized morphism. In this sense generalized morphisms provide a computational tool for performing diagram chases.

**Notation.** If we do not specify the data structure of the generalized morphism category of a category $\mathcal{A}$, we write $\mathrm{G}(\mathcal{A})$. The corresponding embedding functor is $\mathrm{F} : \mathcal{A} \to \mathrm{G}(\mathcal{A})$.

**Definition 3.1** (Pseudo inverse)**.** Let $\mathcal{A}$ be an abelian category.

(1) Let $\varphi \in \mathrm{Mor}_{\mathrm{G^S}(\mathcal{A})}$. The **pseudo-inverse** of $\varphi$ is the morphism $\psi \in \mathrm{Mor}_{\mathrm{G^S}(\mathcal{A})}$ with

$$\begin{aligned} \mathrm{Arrow}(\psi) &:= \mathrm{ReversedArrow}(\varphi), \\ \mathrm{ReversedArrow}(\psi) &:= \mathrm{Arrow}(\varphi). \end{aligned}$$

$$\begin{tikzcd} A \arrow[rr, dashed, "\varphi"] & & B & & B \arrow[rr, dashed, "\psi"] & & A \\ & & & \overset{\text{pseudo-inverse}}{\Rightarrow} & & & \\ & X \arrow[uul, "\varphi_1"] \arrow[uur, "\varphi_2"'] & & & & X \arrow[uul, "\varphi_2"] \arrow[uur, "\varphi_1"'] & \end{tikzcd}$$

(2) Let $\varphi \in \mathrm{Mor}_{\mathrm{G^C}(\mathcal{A})}$. The **pseudo-inverse** of $\varphi$ is the morphism $\psi \in \mathrm{Mor}_{\mathrm{G^C}(\mathcal{A})}$ with

$$\begin{aligned} \mathrm{Arrow}(\psi) &:= \mathrm{ReversedArrow}(\varphi), \\ \mathrm{ReversedArrow}(\psi) &:= \mathrm{Arrow}(\varphi). \end{aligned}$$

$$\begin{tikzcd} & X & & & & X & \\ & & & \overset{\text{pseudo-inverse}}{\Rightarrow} & & & \\ A \arrow[uur, "\varphi_1"] \arrow[rr, dashed, "\varphi"] & & B \arrow[uul, "\varphi_2"'] & & B \arrow[uur, "\varphi_2"] \arrow[rr, dashed, "\psi"] & & A \arrow[uul, "\varphi_1"'] \end{tikzcd}$$

(3) Let $\varphi \in \mathrm{Mor}_{\mathrm{G^T}(\mathcal{A})}$. The **pseudo-inverse** of $\varphi$ is the morphism $\psi \in \mathrm{Mor}_{\mathrm{G^T}(\mathcal{A})}$ which can be computed as follows: Set $(\alpha, \beta) := \mathrm{EpiMonoFactorization}_{\mathcal{A}}(\mathrm{Arrow}(\varphi))$ and

$$\begin{aligned} \pi_i &:= \mathrm{ProjectionInFactorOfFiberProduct}_{\mathcal{A}}((\mathrm{RangeAid}(\varphi), \beta), i), \\ \iota_i &:= \mathrm{InjectionOfCofactorOfPushout}_{\mathcal{A}}((\alpha, \mathrm{SourceAid}(\varphi)), i). \end{aligned}$$

Then

$$\begin{aligned} \mathrm{SourceAid}(\psi) &:= \pi_1, \\ \mathrm{RangeAid}(\psi) &:= \iota_2, \\ \mathrm{Arrow}(\psi) &:= \mathrm{PreCompose}_{\mathcal{A}}(\pi_2, \iota_1). \end{aligned}$$

We write $\mathrm{PseudoInverse}(\varphi) := \psi$. Furthermore, for a morphism $\varphi \in \mathrm{Mor}_{\mathcal{A}}$ we write

$$\mathrm{GeneralizedInverse}(\varphi) := \mathrm{PseudoInverse}(F(\varphi)).$$

**Proposition 3.2** ([Pos17, Prop. II.1.35])**.** *Let $\varphi \in \mathrm{Mor}_{\mathrm{G}(\mathcal{A})}$ and $\varphi^{-1}$ the pseudo-inverse of $\varphi$. Then*

$$\varphi\varphi^{-1}\varphi = \varphi \text{ and } \varphi^{-1}\varphi\varphi^{-1} = \varphi^{-1}.$$

**Proposition 3.3.** *Let $\mathcal{A}$ be an abelian category and $\varphi \in \operatorname{Mor}_{\mathcal{A}}$ a mono- or epimorphism. Then $\mathrm{F}(\varphi)$ is split, and the corresponding pre- or post-inverse is the generalized inverse of $\varphi$.*

Proof. We give a proof for each type of generalized morphisms. Let $\widetilde{\varphi} \in \operatorname{Mor}_{\mathcal{A}}$ be a morphism, $\varphi := F(\widetilde{\varphi})$, and $\psi := \operatorname{GeneralizedInverse}(\varphi)$.

(1) Suppose we are working with spans and $\widetilde{\varphi}$ is a mono. Then the composition $\varphi\psi$ looks as follows:

$$\begin{tikzcd}
A \arrow[rr, dashed, "\varphi"] & & B \arrow[rr, dashed, "\psi"] & & A \\
& A \arrow[ul, "\mathrm{id}_A"] \arrow[ur, "\widetilde{\varphi}"'] & & A \arrow[ul, "\widetilde{\varphi}"] \arrow[ur, "\mathrm{id}_A"'] & \\
& & Y \arrow[ul, "\alpha"] \arrow[ur, "\beta"'] & &
\end{tikzcd}$$

Setting $\alpha, \beta := \operatorname{IdentityMorphism}_{\mathcal{A}}(A)$ fulfills the properties of a fiber product of $\widetilde{\varphi}$ with itself, so we have

$$\varphi\psi \text{ represented by } A \overset{\mathrm{id}_A}{\leftarrow} A \overset{\mathrm{id}_A}{\rightarrow} A.$$

Now suppose $\widetilde{\varphi}$ is an epi. Then the composition $\psi\varphi$ is represented by the two epis $B \overset{\widetilde{\varphi}}{\twoheadleftarrow} A \overset{\widetilde{\varphi}}{\twoheadrightarrow} B$ and $B$ together with the identities is a valid pushout of the two morphisms in $\psi\varphi$. So the result is equivalent to the identity.

(2) Suppose we are working with cospans and $\widetilde{\varphi}$ is an epi. Then the composed cospan $\psi\varphi$ looks as follows:

$$\begin{tikzcd}
& & Y & & \\
& B \arrow[ur, "\alpha"] & & B \arrow[ul, "\beta"'] & \\
B \arrow[ur, "\mathrm{id}_B"] \arrow[rr, dashed, "\psi"] & & A \arrow[ul, "\widetilde{\varphi}"'] \arrow[ur, "\widetilde{\varphi}"] \arrow[rr, dashed, "\varphi"] & & B \arrow[ul, "\mathrm{id}_B"']
\end{tikzcd}$$

Setting $\alpha, \beta := \mathrm{IdentityMorphism}_{\mathcal{A}}(B)$ fulfills the properties of a pushout of $\widetilde{\varphi}$ with itself, so we have

$$\psi\varphi \text{ represented by } B \overset{\mathrm{id}_B}{\leftarrow} B \overset{\mathrm{id}_B}{\rightarrow} B.$$

Now Suppose $\widetilde{\varphi}$ is a mono. Then the composition $\varphi\psi$ is represented by the two monos $A \overset{\widetilde{\varphi}}{\hookrightarrow} B \overset{\widetilde{\varphi}}{\hookleftarrow} A$ and $A$ together with two identities of $A$ is a valid pullback of the arrow and reversed arrow in $\varphi\psi$. So the composition $\varphi\psi$ is equivalent to the identity.

(3) The corresponding honest 3-arrow of a morphism can either be viewed as a span or a cospan, so the claim follows. □

We can use the pseudo-inverse to compute lifts and colifts in $\mathcal{A}$.

**Proposition 3.4** ([Pos17, Cor. II.1.54])**.** *Let $\mathcal{A}$ be an abelian category.*

*(1) Let $\kappa : A \hookrightarrow B$ a monomorphism and $\tau : T \to B$ a morphism such that a morphism $\lambda : T \to A$ with $\lambda\kappa \sim \tau$ exists. Then*

$$\widetilde{\lambda} := F(\tau)\,\mathrm{GeneralizedInverse}(\kappa)$$

*is an honest morphism and we have*

$$\lambda \sim \mathrm{HonestRepresentative}\left(\widetilde{\lambda}\right).$$

*(2) Let $\gamma : A \hookrightarrow B$ a monomorphism and $\tau : A \to T$ a morphism such that a morphism $\lambda : B \to Z$ with $\lambda\kappa \sim \tau$ exists. Then*

$$\widetilde{\lambda} := \mathrm{GeneralizedInverse}(\gamma)\,F(\tau)$$

*is an honest morphism and we have*

$$\lambda \sim \mathrm{HonestRepresentative}\left(\widetilde{\lambda}\right).$$

We give an example how pseudo-inverses allow to perform diagram chases in a purely categorical setting.

**Example 3.5** (Snake lemma)**.** We want to compute the snake in the following diagram with exact rows and commutative squares:

$$\begin{tikzcd}
 & & & \ker(\gamma_3) \arrow[d, "\mathrm{KernelEmbedding}(\gamma_3)"] & \\
 & A \arrow[r, "\alpha_1"] \arrow[d, "\gamma_1"'] & B \arrow[r, "\alpha_2"] \arrow[d, "\gamma_2"] & C \arrow[r] \arrow[d, "\gamma_3"] & 0 \\
0 \arrow[r] & A' \arrow[r, "\beta_1"] \arrow[d, "\mathrm{CokernelProjection}(\gamma_1)"'] & B' \arrow[r, "\beta_2"] & C & \\
 & \mathrm{coker}(\gamma_1) & & &
\end{tikzcd}$$

The snake can now be computed by the following composition:

$$\begin{aligned}\widetilde{\sigma} := \mathrm{KernelEmbedding}(\gamma_3)\, &\mathrm{GeneralizedInverse}(\alpha_2)\, \gamma_2 \\ &\mathrm{GeneralizedInverse}(\beta_1)\, \mathrm{CokernelProjection}(\gamma_1)\,.\end{aligned}$$

The resulting generalized morphism $\widetilde{\sigma}$ is honest, and its honest representative $\sigma$ makes the sequence

$$\ker(\gamma_1) \to \ker(\gamma_2) \to \ker(\gamma_3) \xrightarrow{\sigma} \mathrm{coker}(\gamma_1) \to \mathrm{coker}(\gamma_2) \to \mathrm{coker}(\gamma_3)$$

exact.

**Example 3.6** (IV.3.5, CAP version)**.** Let $\mathcal{A}$ be the category of presented $\mathbb{Z}$-modules, described in Chapter III. Consider the following diagram:

$$\begin{tikzcd}
0 \arrow[r] & \mathbb{Z}^1 \arrow[r, "\alpha_1"] \arrow[d, "\gamma_1"'] & \mathbb{Z}^3 \arrow[r, "\alpha_2"] \arrow[d, "\gamma_2"] & \mathbb{Z}^2 \arrow[r] \arrow[d, "\gamma_3"] & 0 \\
0 \arrow[r] & \mathbb{Z}^2 \arrow[r, "\beta_1"] & \mathbb{Z}^4 \arrow[r, "\beta_2"] & \mathbb{Z}^2 \arrow[r] & 0
\end{tikzcd}$$

with the following matrices:

$$\alpha_1 := \begin{pmatrix} 1 & \cdot & \cdot \end{pmatrix}, \qquad \alpha_2 := \begin{pmatrix} \cdot & \cdot \\ 1 & \cdot \\ \cdot & 1 \end{pmatrix},$$

$$\beta_1 := \begin{pmatrix} 2 & 2 & \cdot & \cdot \\ \cdot & \cdot & 2 & 2 \end{pmatrix}, \ \beta_2 := \begin{pmatrix} 1 & \cdot \\ -1 & \cdot \\ \cdot & 1 \\ \cdot & -1 \end{pmatrix},$$

$$\gamma_1 := \begin{pmatrix} 1 & \cdot \end{pmatrix}, \qquad \gamma_3 := \begin{pmatrix} \cdot & 2 \\ \cdot & -2 \end{pmatrix},$$

$$\gamma_2 := \begin{pmatrix} 2 & 2 & \cdot & \cdot \\ \cdot & \cdot & 2 & \cdot \\ \cdot & \cdot & \cdot & 2 \end{pmatrix}.$$

Note that while $\beta_1$ is a monomorphism in $\mathcal{A}$, it has no split. Using the CAP implementation of $\mathcal{A}$, we compute the snake as follows:

```
gap> LoadPackage( "ModulePresentationsForCAP" );
true
gap> LoadPackage( "GeneralizedMorphismsForCAP" );
true
gap> ZZ := HomalgRingOfIntegers();
Z
gap> ZZ1 := FreeLeftPresentation( 1, ZZ );
<An object in Category of left presentations of Z>
gap> ZZ2 := FreeLeftPresentation( 2, ZZ );
<An object in Category of left presentations of Z>
gap> ZZ3 := FreeLeftPresentation( 3, ZZ );
<An object in Category of left presentations of Z>
gap> ZZ4 := FreeLeftPresentation( 4, ZZ );
<An object in Category of left presentations of Z>
gap> alpha2 := HomalgMatrix( [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ] ], ZZ );
<A 3 x 2 matrix over an internal ring>
gap> alpha2 := PresentationMorphism( ZZ3, alpha2, ZZ2 );
<A morphism in Category of left presentations of Z>
gap> beta1 := HomalgMatrix( [ [ 2, 2, 0, 0 ], [ 0, 0, 2, 2 ] ], ZZ );
<A 2 x 4 matrix over an internal ring>
gap> beta1 := PresentationMorphism( ZZ2, beta1, ZZ4 );
<A morphism in Category of left presentations of Z>
gap> gamma1 := HomalgMatrix( [ [ 1, 0 ] ], ZZ );
<A 1 x 2 matrix over an internal ring>
gap> gamma1 := PresentationMorphism( ZZ1, gamma1, ZZ2 );
```

```
<A morphism in Category of left presentations of Z>
gap> gamma2 := HomalgMatrix( [ [ 2, 2, 0, 0 ], [ 0, 0, 2, 0 ],
>    [ 0, 0, 0, 2 ] ], ZZ );
<A 3 x 4 matrix over an internal ring>
gap> gamma2 := PresentationMorphism( ZZ3, gamma2, ZZ4 );
<A morphism in Category of left presentations of Z>
gap> gamma3 := HomalgMatrix( [ [ 0, 2 ], [ 0, -2 ] ], ZZ );
<A 2 x 2 matrix over an internal ring>
gap> gamma3 := PresentationMorphism( ZZ2, gamma3, ZZ2 );
<A morphism in Category of left presentations of Z>
```

After we have set up the objects and morphisms in the category $\mathcal{A}$, we continue by constructing the necessary generalized morphisms:

```
gap> kernel_gamma3 := KernelEmbedding( gamma3 );
<A monomorphism in Category of left presentations of Z>
gap> coker_gamma1 := CokernelProjection( gamma1 );
<An epimorphism in Category of left presentations of Z>
gap> gen_kernel_gamma3 := AsGeneralizedMorphismBySpan( kernel_gamma3 );
<A morphism in Generalized morphism category of Category of
 left presentations of Z by span>
gap> gen_cokernel_gamma1 := AsGeneralizedMorphismBySpan( coker_gamma1 );
<A morphism in Generalized morphism category of Category of
 left presentations of Z by span>
gap> gen_gamma2 := AsGeneralizedMorphismBySpan( gamma2 );
<A morphism in Generalized morphism category of Category of
 left presentations of Z by span>
gap> gen_inv_alpha2 := GeneralizedInverseBySpan( alpha2 );
<A morphism in Generalized morphism category of Category of
  left presentations of Z by span>
gap> gen_inv_beta1 := GeneralizedInverseBySpan( beta1 );
<A morphism in Generalized morphism category of Category of
 left presentations of Z by span>
```

Now we can compute the snake:

```
gap> snake := PreCompose( gen_kernel_gamma3, gen_inv_alpha2 );
<A morphism in Generalized morphism category of Category of
 left presentations of Z by span>
gap> snake := PreCompose( snake, gen_gamma2 );
<A morphism in Generalized morphism category of Category of
 left presentations of Z by span>
gap> snake := PreCompose( snake, gen_inv_beta1 );
<A morphism in Generalized morphism category of Category of
```

```
 left presentations of Z by span>
gap> snake := PreCompose( snake, gen_cokernel_gamma1 );
<A morphism in Generalized morphism category of Category of
 left presentations of Z by span>
gap> IsHonest( snake );
true
gap> Display( HonestRepresentative( snake ) );
[ [  1,  1 ] ]

A morphism in Category of left presentations of Z
gap> Display( Range( snake ) );
[ [  1,  0 ] ]

An object in Category of left presentations of Z
```

So the snake morphism is

$$\mathbb{Z}^1 \xrightarrow{(1\ 1)} \mathbb{Z}^2/\langle(1\ 0)\rangle\,.$$

## 4. Serre quotients

One goal of this thesis mentioned in Chapter I is a computable description of coherent sheaves over toric varieties. The computational model for the category of coherent sheaves over a toric variety will be the so-called Serre quotients, which we will define using generalized morphisms. Serre quotients are described in [**BLH14b**, §1.1]. We recapitulate the main definitions.

**Definition 4.1** (Thick subcategory)**.** Let $\mathcal{A}$ be an abelian category. A subcategory $\mathcal{C}$ is called **thick** if it is closed under extensions and for any object $A \in \mathrm{Obj}_{\mathcal{C}}$ the subcategory $\mathcal{C}$ contains all subfactors of $A$.

The objects in $\mathcal{C}$ will all become zero objects in the Serre quotient, and, hence, a morphism $\varphi$ in $\mathcal{A}$ which kernel and cokernel object lie in $\mathcal{C}$ will be an isomorphism in $\mathcal{A}$ modulo $\mathcal{C}$.

**Definition 4.2** (Serre quotient)**.** Let $\mathcal{A}$ be an abelian category and $\mathcal{C} \subseteq \mathcal{A}$ a thick subcategory. Then the **Serre quotient category** $\mathcal{A}/\mathcal{C}$ is defined as follows:

(1) The object class is the same as that of $\mathcal{A}$;
(2) For two objects $A, B \in \mathrm{Obj}_{\mathcal{A}}$ we set

$$\mathrm{Hom}_{\mathcal{A}/\mathcal{C}}(A,B) := \varinjlim_{\substack{M' \hookrightarrow M, N' \hookrightarrow N \\ M/M', N' \in \mathcal{C}}} (M', N/N')\,.$$

Our model for Serre quotients will be a certain subcategory of the generalized morphism category, which we now define.

**Definition 4.3** (Gabriel morphisms)**.** Let $\mathcal{A}$ be an abelian category and $\mathcal{C}$ a thick subcategory.

(1) A *normalized* generalized morphism by 3-arrows $\varphi \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A})}$ is called a **Gabriel morphism (of $\mathcal{A}$ with respect to $\mathcal{C}$)** if

$$\begin{gathered}\mathrm{CokernelObject}_{\mathcal{A}}(\mathrm{SourceAid}(\varphi)),\\ \mathrm{KernelObject}_{\mathcal{A}}(\mathrm{RangeAid}(\varphi))\end{gathered}$$

are objects in $\mathcal{C}$.

$$\begin{tikzcd}
\mathrm{CokernelObject}_{\mathcal{A}}(\iota) \in \mathcal{C} & & \mathrm{KernelObject}_{\mathcal{A}}(\pi) \in \mathcal{C} \arrow[d, hook, "\mathrm{KernelEmbedding}_{\mathcal{A}}(\pi)"'] \\
A \arrow[u, two heads, "\mathrm{CokernelProjection}_{\mathcal{A}}(\iota)"] \arrow[rr, dashed, "\varphi"] & & B \arrow[d, two heads, "\pi"] \\
A' \arrow[u, hook, "\iota"] \arrow[rr, "\alpha"'] & & B''
\end{tikzcd}$$

Informally we say that both $\mathrm{SourceAid}(\varphi)$ and $\mathrm{RangeAid}(\varphi)$ are isomorphisms up to objects in $\mathcal{C}$.

(2) A generalized morphism by spans $\varphi \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}$ is called a **Gabriel morphism (of $\mathcal{A}$ with respect to $\mathcal{C}$)** if its conversion to a 3-arrow morphism $\mathrm{C}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A}),\mathrm{G}^{\mathrm{S}}(\mathcal{A})}(\varphi)$ is a Gabriel morphism.

Equivalently, $\varphi$ is a Gabriel morphism if both the cokernel object of the reversed arrow of $\varphi$ and the image object of the kernel object of the reversed arrow under the arrow of $\varphi$ are objects in $\mathcal{C}$.

$$\begin{tikzcd}
\mathrm{CokernelObject}_{\mathcal{A}}(\alpha) \in \mathcal{C} & & & & \\
A \arrow[u, two heads, "\mathrm{CokernelProjection}_{\mathcal{A}}(\alpha)"'] \arrow[rrrr, dashed, "\varphi"] & & & & B \\
& & X \arrow[llu, "\alpha"] \arrow[rru, "\beta"'] & & \mathrm{ImageObject}_{\mathcal{A}}(\kappa\beta) \in \mathcal{C} \arrow[u, hook] \\
& & \mathrm{KernelObject}_{\mathcal{A}}(\alpha) \arrow[u, hook, "\kappa := \mathrm{KernelEmbedding}_{\mathcal{A}}(\alpha)"] \arrow[rru, two heads] & &
\end{tikzcd}$$

(3) A generalized morphism by cospans $\varphi \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A})}$ is called a **Gabriel morphism (of $\mathcal{A}$ with respect to $\mathcal{C}$)** if its conversion to a 3-arrow morphism $\mathrm{C}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A}),\mathrm{G}^{\mathrm{C}}(\mathcal{A})}(\varphi)$ is a Gabriel morphism.

Equivalently, $\varphi$ is a Gabriel morphism if both the kernel object of the reversed arrow of $\varphi$ and the image object of the composition of the arrow and the cokernel projection of the reversed arrow of $\varphi$ are objects in $\mathcal{C}$.

$$\begin{tikzcd}
& & \mathrm{CokernelObject}_{\mathcal{A}}(\beta) & & \\
\mathrm{ImageObject}_{\mathcal{A}}(\alpha\epsilon) \in \mathcal{C} \arrow[rru, two heads] & & X \arrow[u, two heads, "\epsilon := \mathrm{CokernelProjection}_{\mathcal{A}}(\beta)"'] & & \\
A \arrow[u, hook] \arrow[rru, "\alpha"] \arrow[rrrr, dashed, "\varphi"] & & & & B \arrow[llu, "\beta"'] \\
& & & & \mathrm{KernelObject}_{\mathcal{A}}(\beta) \in \mathcal{C} \arrow[u, hook, "\mathrm{KernelEmbedding}_{\mathcal{A}}(\beta)"]
\end{tikzcd}$$

Since Gabriel morphisms are generalized morphisms, the term honest and all operations (SourceAid, RangeAid, Arrow, ReversedArrow, HonestRepresentative, PseudoInverse) defined for generalized morphisms apply.

**Definition 4.4.** Let $\mathcal{A}$ be an abelian category and $\mathcal{C}$ a thick subcategory. We denote by $\mathrm{G}_{\mathcal{C}}(\mathcal{A})$ the **subcategory of Gabriel morphisms** of $\mathrm{G}(\mathcal{A})$ and the respective presentations with $\mathrm{G}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$, $\mathrm{G}^{\mathrm{C}}_{\mathcal{C}}(\mathcal{A})$, and $\mathrm{G}^{\mathrm{T}}_{\mathcal{C}}(\mathcal{A})$.

The generalized morphisms are Gabriel morphisms if and only if the "helper" morphisms ReversedArrow, SourceAid, and RangeAid of the normalized representatives are isomorphisms in $\mathcal{A}/\mathcal{C}$, i.e., isomorphisms up to objects in $\mathcal{C}$.

For the proof that the category of Gabriel morphisms is indeed a category see [**BLH14b**, §2.5].

**Definition 4.5** (Zeroid of $\mathrm{G}_{\mathcal{C}}(\mathcal{A})$)**.** Let $\mathcal{A}$ be an abelian category and $\mathcal{C}$ a thick subcategory. For two objects $A, B \in \mathrm{G}_{\mathcal{C}}(\mathcal{A})$ we define the **zeroid** as

$$\mathrm{Z}_{A,B} := \left\{\varphi \in \mathrm{Hom}_{\mathrm{G}_{\mathcal{C}}(\mathcal{A})}(A,B) \;\middle|\; \mathrm{ImageObject}_{\mathcal{A}}(\mathrm{Arrow}(\varphi)) \in \mathrm{Obj}_{\mathcal{C}}\right\}.$$

**Definition 4.6.** Let $\mathcal{A}$ be an abelian category and $\mathcal{C}$ a thick subcategory. The **Serre morphism category** $\overline{\mathrm{G}}_{\mathcal{C}}(\mathcal{A})$ (of $\mathcal{A}$ with respect to $\mathcal{C}$) has the same object class as $\mathrm{G}_{\mathcal{C}}(\mathcal{A})$, and the Hom-sets are the quotients by the zeroid, i.e., for two objects $A, B \in \mathrm{G}_{\mathcal{C}}(\mathcal{A})$ we set

$$\mathrm{Hom}_{\overline{\mathrm{G}}_{\mathcal{C}}(\mathcal{A})}(A,B) := \mathrm{Hom}_{\mathrm{G}_{\mathcal{C}}(\mathcal{A})}(A,B)\,/\,\mathrm{Z}_{A,B}\,.$$

The respective presentations of $\overline{\mathrm{G}}_{\mathcal{C}}(\mathcal{A})$ using spans, cospans, and 3-arrow generalized morphisms are denoted with $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$, $\overline{\mathrm{G}}^{\mathrm{C}}_{\mathcal{C}}(\mathcal{A})$, and $\overline{\mathrm{G}}^{\mathrm{T}}_{\mathcal{C}}(\mathcal{A})$.

**Theorem 4.7.** *Let $\mathcal{A}$ be an abelian category and $\mathcal{C}$ a thick subcategory. The Serre quotient category $\mathcal{A}/\mathcal{C}$ is equivalent to the Serre morphism category $\mathrm{G}_{\mathcal{C}}(\mathcal{A})$.*

A proof of this theorem and further motivation can be found in [**BLH14b**, Thm. 3.1].

## 5. Computability of Serre quotients

We are now going to describe the algorithms necessary for the category of Serre morphisms to be computable and therefore a model for the Serre quotient category. The algorithms for the normalized 3-arrow morphism category can be found in the proof of [**BLH14b**, Thm. 1.1], but for completeness we include the necessary constructions in Subsection IV.5.d.

**Theorem 5.1** ([**BLH14b**, Thm. 1.1])**.** *Let $\mathcal{A}$ be computable abelian. Then the category $\overline{\mathrm{G}}^{\mathrm{T}}_{\mathcal{C}}(\mathcal{A})$ is computable abelian.*

### 5.a. Computability of Serre morphisms by spans.

We show that the category of Serre morphisms by spans is computable abelian by going through the constructions from Chapter II, and therefore provides a suitable data structure for $\mathcal{A}/\mathcal{C}$. Throughout the hole section $\mathcal{A}$ will denote a computable abelian category and $\mathcal{C}$ a thick subcategory with decidable membership.

**Theorem 5.2.** *The category $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ is equivalent to $\overline{\mathrm{G}}^{\mathrm{T}}_{\mathcal{C}}(\mathcal{A})$ and therefore abelian.*

Proof. Since $\mathrm{G}^{\mathrm{S}}(\mathcal{A})$ and $\mathrm{G}^{\mathrm{T}}(\mathcal{A})$ are equivalent, and the definitions of Gabriel morphisms and the zeroid in both categories $\mathrm{G}^{\mathrm{S}}(\mathcal{A})$ and $\mathrm{G}^{\mathrm{T}}(\mathcal{A})$ correspond to each other under the conversion functor $\mathrm{C}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A}),\mathrm{G}^{\mathrm{S}}(\mathcal{A})}$ the claim follows. □

**Proposition 5.3.** *The category $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ is computable.*

Proof. Since $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ is a subcategory of the computable category $\mathrm{G}^{\mathrm{S}}(\mathcal{A})$ and composition and identity morphisms are inherited from $\mathrm{G}^{\mathrm{S}}(\mathcal{A})$, the operations IdentityMorphism and PreCompose are computable. We still need to provide the proper constructions for the equalities:

(1) The operation $\mathrm{IsEqualForObjects}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}$ is inherited from the corresponding equality $\mathrm{IsEqualForObjects}_{\mathcal{A}}$ of objects in $\mathcal{A}$ and therefore computable.[7]

(2) The equality notion $\mathrm{IsEqualForMorphisms}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}$ is computed via the following algorithm: Let $\varphi, \psi : A \to B$ in $\mathrm{Mor}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}$:
   (a) Compute $-\varphi := \mathrm{AdditiveInverse}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}(\varphi)$.
   (b) Compute $\pi := \mathrm{AdditionForMorphisms}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}(-\varphi, \psi)$.
   (c) Compute $I := \mathrm{ImageObject}_{\mathcal{A}}(\mathrm{Arrow}(\pi))$.
   (d) Use the membership function to decide if $I \in \mathcal{C}$.

   If $I \in \mathcal{C}$, the morphisms $\varphi$ and $\psi$ are equal.

(3) Since the membership for $\mathcal{C}$ is decidable and we can compute images as kernels of cokernels of morphisms in $\mathcal{A}$, the morphism set $\mathrm{Mor}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}$ is decidable . □

The decidability of the membership function of $\mathcal{C}$ has a big part in the realization of the Serre quotient category. Without the decidability of $\mathcal{C}$, the category would not have decidable equalities, and therefore no realization.

**Proposition 5.4.** *The category $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ is computable preadditive.*

Proof. Let $\varphi, \psi : A \to B$ in $\mathrm{Mor}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}$. The operations for zero morphism, addition, and the additive inverse are defined by the zero morphism, addition, and additive inverse (for the enrichment structure) in $\mathrm{G}^{\mathrm{S}}(\mathcal{A})$. We have $\varphi = \psi$ if for any of their Gabriel morphism representatives $\varphi', \psi' \in \mathrm{G}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ the morphism $\varphi' + (-\psi')$ lies in $\mathrm{Z}_{A,B}$. So addition and additive inverse are independent of the choice of representative in the set $\mathrm{Hom}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}(A, B)$.

We still need to show that the additive inverse in the sense of commutative semigroups is an additive inverse in the sense of abelian groups. So we sum a morphism $\varphi : A \to B$ in $\mathrm{G}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ and its additive inverse. We get the following representative for the sum:

$$\begin{aligned}\mathrm{ReversedArrow}(\varphi + (-\varphi)) &= \mathrm{ReversedArrow}(\varphi)\,,\\ \mathrm{Arrow}(\varphi + (-\varphi)) &= \mathrm{Arrow}(\varphi) - \mathrm{Arrow}(\varphi)\,.\end{aligned}$$

[7]Remember, the object classes of $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ and $\mathcal{A}$ coincide.

The arrow of the resulting morphism is zero, and so its image is in $\mathcal{C}$, which means that $\varphi + (-\varphi) \in \mathrm{Z}_{A,B}$. So $\varphi + (-\varphi)$ is equivalent to the zero morphism from $A$ to $B$. $\square$

**Proposition 5.5.** *The category* $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ *is computable additive.*

To describe the operations needed for the proof, we need another definition.

**Definition 5.6.** Let $\varphi_1, \varphi_2 \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}$ with $\mathrm{Source}(\varphi_1) = \mathrm{Source}(\varphi_2)$ and

$\pi_i := \mathrm{ProjectionInFactorOfFiberProduct}_{\mathcal{A}}\left(\left(\mathrm{ReversedArrow}(\varphi_1), \mathrm{ReversedArrow}(\varphi_2)\right), i\right).$

$$\begin{tikzcd}
& B & \\
& X \arrow[u, "\beta_1"] \arrow[dl, "\alpha_1"'] & \\
A \arrow[uur, dashed, "\varphi_1"] \arrow[ddr, dashed, "\varphi_2"'] & & Z \arrow[ul, "\pi_1"'] \arrow[uul, "\pi_1\beta_1"'] \arrow[dl, "\pi_2"] \arrow[ddl, "\pi_2\beta_2"] \\
& Y \arrow[ul, "\alpha_2"] \arrow[d, "\beta_2"] & \\
& C &
\end{tikzcd}$$

The **common restriction** $\mathrm{CommonRestriction}(\varphi_1, \varphi_2)$ of $\varphi_1$ and $\varphi_2$ is the pair of generalized morphisms $(\varphi_1', \varphi_2')$ represented by

$$\mathrm{Arrow}(\varphi_i') := \pi_i \beta_i$$

and

$$\mathrm{ReversedArrow}(\varphi_i') := \pi_i \alpha_i.$$

**Proposition 5.7.** *Let* $\varphi_1 : A \to B$, $\varphi_2 : A \to C$ *in* $\mathrm{Mor}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}$, *and* $(\varphi_1', \varphi_2') := \mathrm{CommonRestriction}(\varphi_1, \varphi_2)$. *Then*

$$\varphi_1 = \varphi_1' \textit{ and } \varphi_2 = \varphi_2' \textit{ in } \overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A}).$$

Proof. We show $\varphi_1 - \varphi_1' = 0$. Suppose $\varphi_1$ is represented by the span $A \overset{\alpha}{\leftarrow} D \overset{\beta}{\rightarrow} B$ and after the restriction $\varphi_1'$ is represented by $A \overset{\alpha}{\leftarrow} D \overset{\pi}{\leftarrow} X \overset{\pi}{\rightarrow} D \overset{\beta}{\pi} B$. To compute the sum, we first compute the fiber product of the reversed arrows of the representations of $\varphi_1$ and $\varphi_1'$:

$$\begin{tikzcd}
 & & A & & \\
 & & & D \arrow[ul, "\alpha"'] & \\
D \arrow[uurr, "\alpha"] & & & & X \arrow[ul, "\pi"'] \\
 & & & & \\
 & & Y \arrow[uull, "\gamma_1"] \arrow[uurr, "\gamma_2"'] & &
\end{tikzcd}$$

To compute $\varphi_1 - \varphi_2'$, we need to compute $\gamma_1\beta - \gamma_2\pi\beta = (\gamma_1 - \gamma_2\pi)\,\beta$. By the commutativity of the fiber product diagram above[8], we know that

$$\gamma_1\alpha \sim \gamma_2\pi\alpha,$$

and therefore

$$(\gamma_1 - \gamma_2\pi)\,\alpha \sim 0_{Y,A},$$

which means that the image of $(\gamma_1 - \gamma_2\pi)$ lies in the kernel of $\alpha$, and therefore in $\mathcal{C}$. It follows that the image $(\gamma_1 - \gamma_2\pi)\,\beta$ lies in $\mathcal{C}$ as well. By symmetry, we also have $\varphi_2 = \varphi_2'$. □

Note that for generalized morphisms Proposition IV.5.7 is not true in general.

Proof of Proposition IV.5.5. To show that the category $\overline{\mathrm{G}}_{\mathcal{C}}^{\mathrm{S}}(\mathcal{A})$ is computable additive, we need to prove that there is a computable zero object, and computable direct sums, which are products and coproducts at the same time.

Let 0 be a zero object in $\mathcal{A}$. Then 0, interpreted as object in $\overline{\mathrm{G}}_{\mathcal{C}}^{\mathrm{S}}(\mathcal{A})$, is a zero object in $\overline{\mathrm{G}}_{\mathcal{C}}^{\mathrm{S}}(\mathcal{A})$ (remember, the object classes of $\mathcal{A}$ and $\overline{\mathrm{G}}_{\mathcal{C}}^{\mathrm{S}}(\mathcal{A})$ coincide). Furthermore, for an object $A \in \mathrm{Obj}_{\overline{\mathrm{G}}_{\mathcal{C}}^{\mathrm{S}}(\mathcal{A})}$ we define

$$\mathrm{UniversalMorphismIntoZeroObject}_{\overline{\mathrm{G}}_{\mathcal{C}}^{\mathrm{S}}(\mathcal{A})}(A) := \mathrm{ZeroMorphism}_{\overline{\mathrm{G}}_{\mathcal{C}}^{\mathrm{S}}(\mathcal{A})}(A, 0)\,,$$
$$\mathrm{UniversalMorphismFromZeroObject}_{\overline{\mathrm{G}}_{\mathcal{C}}^{\mathrm{S}}(\mathcal{A})}(A) := \mathrm{ZeroMorphism}_{\overline{\mathrm{G}}_{\mathcal{C}}^{\mathrm{S}}(\mathcal{A})}(0, A)\,.$$

By construction, those morphisms are well-defined and computable.

We now prove the universal properties of the universal morphisms from and to the zero object. Let

$$\zeta_A := \mathrm{UniversalMorphismIntoZeroObject}_{\overline{\mathrm{G}}_{\mathcal{C}}^{\mathrm{S}}(\mathcal{A})}(A)$$

be the universal morphism into the zero object, i.e., $\zeta_A$ is represented by $A \xleftarrow{\mathrm{id}_A} A \xrightarrow{0_A} 0$, and $\varphi \in \mathrm{Hom}_{\overline{\mathrm{G}}_{\mathcal{C}}^{\mathrm{S}}(\mathcal{A})}(A, 0)$. We need to show that $\zeta_A = \varphi$, so we compute $\zeta_A - \varphi$. Let $\varphi$ be

[8]Note that the commutativity of the fiber product diagram is only up to congruence $\sim$ of morphisms in $\mathcal{A}$.

represented by the span $A \xleftarrow{\alpha} X \xrightarrow{0_X} 0$. Then, to compute the sum, we first compute the fiber product of $\mathrm{id}_A$ and $\alpha$, i.e.,

$$\pi_i := \mathrm{ProjectionInFactorOfFiberProduct}_{\mathcal{A}}\left(\left(\mathrm{id}_A, \alpha\right), i\right),$$

$i = 1, 2$.

$$\begin{tikzcd} & A & \\ A \arrow[ur, "\mathrm{id}_A"] & & X \arrow[ul, "\alpha"'] \\ & Y \arrow[ul, "\pi_1"] \arrow[ur, "\pi_2"'] & \end{tikzcd}$$

Since $\mathrm{id}_A$ is an isomorphism, $\pi_2$ is also an isomorphism, and we can assume $Y = X$, $\pi_1 = \alpha$, and $\pi_2 = \mathrm{id}_X$. So have

$$\mathrm{Arrow}\left(\zeta_A - \varphi\right) = \alpha 0_X - \mathrm{id}_X 0_X = 0_Y,$$

and therefore $\zeta_A = \varphi$. The proof for the universality of UniversalMorphismFromZeroObject is analogous.

We now construct the direct sum and show the universality of the construction. Let $A_1, A_2 \in \overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}\left(\mathcal{A}\right)$. Their direct sum

$$\mathrm{DirectSum}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}\left(A_1, A_2\right)$$

is defined to be the direct sum $\mathrm{DirectSum}_{\mathcal{A}}\left(A_1, A_2\right) \in \mathrm{Obj}_{\mathcal{A}}$, interpreted as an object in $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}\left(\mathcal{A}\right)$. We define the injection $\iota_i := \mathrm{InjectionOfCofactorOfDirectSum}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}\left(\left(A_1, A_2\right), i\right)$ of the $i$-th cofactor, $i = 1, 2$, to be represented by the span

$$\begin{tikzcd} A_i \arrow[rr, dashed, "\iota_i"] & & A_1 \oplus A_2 \\ & A_i \arrow[ul, "\mathrm{id}_{A_i}"] \arrow[ur, "\widetilde{\iota_i}"'] & \end{tikzcd}$$

with

$$\widetilde{\iota_i} := \mathrm{InjectionOfCofactorOfDirectSum}_{\mathcal{A}}\left(\left(A_1, A_2\right), i\right) \in \mathrm{Mor}_{\mathcal{A}},$$

$i = 1, 2$. We define the projection

$$\pi_i := \mathrm{ProjectionInFactorOfDirectSum}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}\left(\left(A_1, A_2\right), i\right)$$

to the $i$-th factor, $i = 1, 2$, to be represented by the span

$$\begin{tikzcd}
A_1 \oplus A_2 \arrow[rr, dashed, "\pi_i"] & & A_i \\
& A_1 \oplus A_2 \arrow[ul, "\mathrm{id}_{A_1 \oplus A_2}"] \arrow[ur, "\widetilde{\pi}_i"'] &
\end{tikzcd}$$

with

$$\widetilde{\pi}_i := \mathrm{ProjectionInFactorOfDirectSum}_{\mathcal{A}}\left(\left(A_1, A_2\right), i\right) \in \mathrm{Mor}_{\mathcal{A}},$$

$i = 1, 2$.

To define the universal morphism into the direct sum, let $\varphi : A \to B, \psi : A \to C$ in $\mathrm{Mor}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}$ and set

$$\left(\varphi', \psi'\right) := \mathrm{CommonRestriction}\left(\varphi, \psi\right).$$

Suppose $\varphi'$ and $\psi'$ are represented by the following spans:

$$\begin{tikzcd}
A \arrow[rr, dashed, "\varphi'"] & & B & A \arrow[rr, dashed, "\psi'"] & & C \\
& Z \arrow[ul, "\alpha"] \arrow[ur, "\beta_1"'] & & & Z \arrow[ul, "\alpha"] \arrow[ur, "\beta_2"'] &
\end{tikzcd}$$

With

$$\beta := \mathrm{UniversalMorphismIntoDirectSum}_{\mathcal{A}}\left(\beta_1, \beta_2\right)$$

we define

$$\gamma := \mathrm{UniversalMorphismIntoDirectSum}_{\mathcal{A}}\left(\varphi, \psi\right)$$

to be represented by the span

$$\begin{tikzcd}
A \arrow[rr, dashed, "\gamma"] & & B \oplus C \\
& Z \arrow[ul, "\alpha"] \arrow[ur, "\beta"'] &
\end{tikzcd}$$

In operator language we have

$$\begin{aligned}
&\mathrm{ReversedArrow}\left(\mathrm{UniversalMorphismIntoDirectSum}\left(\varphi, \psi\right)\right) \\
&\quad := \mathrm{ReversedArrow}\left(\varphi'\right)
\end{aligned}$$

and

$$\begin{aligned}&\mathrm{Arrow}\left(\mathrm{UniversalMorphismIntoDirectSum}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}(\varphi,\psi)\right)\\ &\quad := \mathrm{UniversalMorphismIntoDirectSum}_{\mathcal{A}}\left(\mathrm{Arrow}\left(\varphi'\right),\mathrm{Arrow}\left(\psi'\right)\right).\end{aligned}$$

To show that the defined universal morphism into the direct fulfills the universal property, let

$$\pi_B := \mathrm{ProjectionInFactorOfDirectSum}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}\left(\left(B,C\right),1\right).$$

The diagram for the composition $\gamma\pi_B$ is

$$\begin{tikzcd}
A \arrow[rr, dashed, "\gamma"] & & B \oplus C \arrow[rr, dashed, "\pi_B"] & & B \\
& Z \arrow[ul, "\alpha"] \arrow[ur, "\beta"'] & \mathrm{id}_{B\oplus C} & B \oplus C \arrow[ul] \arrow[ur, "\widetilde{\pi}_B"'] & \\
& & K \arrow[ul, "\epsilon_1"] \arrow[ur, "\epsilon_2"'] & &
\end{tikzcd}$$

with

$$\widetilde{\pi}_B := \mathrm{ProjectionInFactorOfDirectSum}_{\mathcal{A}}\left(\left(B,C\right),1\right) \in \mathrm{Mor}_{\mathcal{A}}.$$

We can assume that $K = Z$, $\epsilon_1 = \mathrm{id}_Z$, and $\epsilon_2 = \beta$ since this setting leads to a valid fiber product. Therefore, by the universal property of $\widetilde{\pi}_B$ we have $\beta\widetilde{\pi}_B \sim \beta_1$ and $\gamma\pi_B = \varphi' = \varphi$.

To define the universal morphism from the direct sum, let $\varphi := B \to A$ and $\psi : C \to A$ in $\mathrm{Mor}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}$, with $\varphi : B \overset{\alpha_1}{\leftarrow} X \overset{\beta_1}{\to} A$ and $\psi : C \overset{\alpha_2}{\leftarrow} Y \overset{\beta_2}{\to} A$. We define $\gamma := \mathrm{UniversalMorphismFromDirectSum}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}(\varphi,\psi)$ to be represented by the span

$$\begin{tikzcd}
B \oplus C \arrow[rr, dashed, "\gamma"] & & A \\
& X \oplus Y \arrow[ul, "\alpha_1 \oplus \alpha_2"] \arrow[ur, "\beta"'] &
\end{tikzcd}$$

with

$$\beta := \mathrm{UniversalMorphismFromDirectSum}_{\mathcal{A}}\left(\beta_1,\beta_2\right).$$

In operator language we have

$$\begin{aligned}&\mathrm{Arrow}\left(\mathrm{UniversalMorphismFromDirectSum}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}(\varphi,\psi)\right)\\ &\quad := \mathrm{UniversalMorphismFromDirectSum}_{\mathcal{A}}(\mathrm{Arrow}(\varphi),\mathrm{Arrow}(\psi))\end{aligned}$$

and

$$\begin{aligned}&\mathrm{ReversedArrow}\left(\mathrm{UniversalMorphismFromDirectSum}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}(\varphi,\psi)\right)\\ &\quad := \mathrm{DirectSumFunctorial}_{\mathcal{A}}(\mathrm{ReversedArrow}(\varphi),\mathrm{ReversedArrow}(\psi)).\end{aligned}$$

To show that the defined universal morphism from the direct sum fulfills the universal property, let $\iota_B := \mathrm{InjectionOfCofactorOfDirectSum}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}((B,C),1)$. The diagram for the composition $\iota_B\gamma$ is

$$\begin{tikzcd}
B \arrow[rr, dashed, "\iota_B"] & & B \oplus C \arrow[rr, dashed, "\gamma"] & & A \\
& B \arrow[ul, "\mathrm{id}_B"] \arrow[ur, "\widetilde{\iota}_B"'] & & X \oplus Y \arrow[ul, "\alpha_1 \oplus \alpha_2"] \arrow[ur, "\beta"'] & \\
& & Z \arrow[ul, "\epsilon_1"] \arrow[ur, "\epsilon_2"'] & &
\end{tikzcd}$$

with

$$\widetilde{\iota}_B := \mathrm{InjectionOfCofactorOfDirectSum}_{\mathcal{A}}((B,C),1) \in \mathrm{Mor}_{\mathcal{A}}.$$

We have $\epsilon_1\widetilde{\iota}_B = \epsilon_2(\alpha_1 \oplus \alpha_2)$, which means that the image of $\epsilon_2(\alpha_1 \oplus \alpha_2)$ lies in $B$. Hence the image of $\epsilon_2$ lies in $X$. To get a valid fiber product we can assume $Z = X$, $\epsilon_1 = \alpha_1$, and $\epsilon_2 = \{\mathrm{id}_Y, 0_{Y,X}\}$, and therefore we have $\iota_B\gamma = \varphi$. □

Before we prove that the category $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ is computable preabelian, we establish a computational trick.

**Proposition 5.8.** *Let $\mathcal{A}$ be an abelian category and $\mathcal{C}$ a thick subcategory. Let furthermore $\mathrm{F} : \mathcal{A} \to \overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ be the projection functor, i.e., the functor mapping a morphism $\gamma : A \to B$ in $\mathrm{Mor}_{\mathcal{A}}$ to the morphism $\overline{\gamma} \in \overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ represented by the span $A \overset{\mathrm{id}_A}{\leftarrow} A \overset{\gamma}{\rightarrow} B$ in $\mathcal{A}$. Then the induced functor*

$$\mathrm{GF} : \mathrm{G}^{\mathrm{S}}(\mathcal{A}) \to \mathrm{G}^{\mathrm{S}}\left(\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})\right)$$

*is full and the preimage of a morphism $\psi \in \mathrm{G}^{\mathrm{S}}\left(\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})\right)$ with*

$$\mathrm{Arrow}(\psi) \;\; \textit{represented by } \psi_1 := X \overset{\alpha}{\leftarrow} Y \overset{\beta}{\rightarrow} B \in \mathrm{G}^{\mathrm{S}}(\mathcal{A})$$

*and*

$$\mathrm{ReversedArrow}\,(\psi) \text{ represented by } \psi_2' := X \xleftarrow{\gamma} Z \xrightarrow{\epsilon} A$$

*is* $\psi_2^{-1}\psi_1$*, where* $\psi_2^{-1}$ *denotes the pseudo-inverse of* $\psi_2$*.*

Proof. Let $\varphi \in \mathrm{Hom}_{\mathrm{G}^{\mathrm{S}}\left(\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})\right)}(A,B)$ represented by the span $A \xleftarrow{\alpha} X \xrightarrow{\beta} B$ with $\alpha,\beta \in \mathrm{Mor}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}$ such that $\alpha$ and $\beta$ are represented by morphisms $\alpha',\beta' \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}$. Let $\alpha'^{-1} : A \to X := \mathrm{PseudoInverse}\,(\alpha') \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}$. Then we can compute

$$\gamma' := \alpha'^{-1}\beta' \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})},$$

and we have

$$\mathrm{GF}\,(\gamma') = \mathrm{GF}\,(\alpha')^{-1}\,\mathrm{GF}\,(\beta') = \varphi. \qquad \square$$

Remark 5.9. We can interpret Proposition IV.5.8 as follows: Whenever we need to compute with a generalized morphism by spans over the Serre morphisms category, i.e., with a morphism $\varphi \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{S}}\left(\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})\right)}$, we can treat the two Serre morphisms $\mathrm{Arrow}\,(\varphi)$ and $\mathrm{ReversedArrow}\,(\varphi)$ in a representing span of $\varphi$ like the composition of their representatives in $\mathrm{G}^{\mathrm{S}}(\mathcal{A})$. The result can then be mapped back to $\mathrm{Mor}_{\mathrm{G}^{\mathrm{S}}\left(\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})\right)}$ using the functor GF from Proposition IV.5.8.

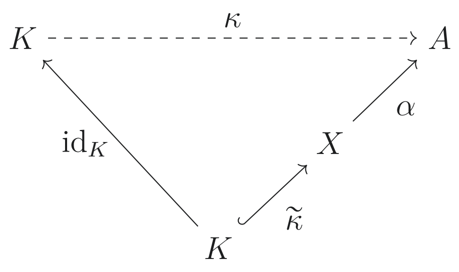
**Theorem 5.10.** *The category* $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ *is computable preabelian.*

Proof. We start by giving the constructions and proofs for the kernel. Let $\varphi \in \mathrm{Hom}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}(A,B)$, represented by the span $A \xleftarrow{\alpha} X \xrightarrow{\beta} B$ in $\mathcal{A}$. To construct the kernel embedding, let

$$\widetilde{\kappa} := \mathrm{KernelEmbedding}_{\mathcal{A}}\,(\beta).$$

Then we set $\kappa := \mathrm{KernelEmbedding}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}(\varphi)$ to be represented by the corresponding honest span of

$$\mathrm{PreCompose}_{\mathcal{A}}\,(\widetilde{\kappa},\alpha).$$


$$\begin{tikzcd} K \arrow[rr, dashed, "\kappa"] & & A \\ & & X \arrow[u, "\alpha"'] \\ & K \arrow[uul, "\mathrm{id}_K"] \arrow[ur, hook, "\widetilde{\kappa}"'] & \end{tikzcd}$$

We show that $\kappa\varphi = 0_{K,B}$. The composition can be represented by the following diagram:

$$\begin{tikzcd}
K \arrow[rrr, dashed, "\kappa"] & & & A \arrow[rrr, dashed, "\varphi"] & & & B \\
& & X \arrow[ur, "\alpha"] & & & & \\
& K \arrow[uul, "\mathrm{id}_K"] \arrow[ur, "\widetilde{\kappa}"] & & & & X \arrow[uull, "\alpha"'] \arrow[uur, "\beta"'] & \\
& & & Y \arrow[ull, "\epsilon_1"] \arrow[urr, "\epsilon_2"'] & & &
\end{tikzcd}$$

Setting $Y = K$ and $\epsilon_2 = \widetilde{\kappa}$ forms a valid fiber product diagram, so we have

$$\epsilon_2\beta = \widetilde{\kappa}\beta = 0_{K,B},$$

and therefore $\kappa\varphi$ is zero.

To construct the kernel lift, let $\tau : T \to A$ in $\mathrm{Mor}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}$ represented by the span $T \xleftarrow{\tau_1} Z \xrightarrow{\tau_2} A$ such that $\tau\varphi$ is zero. Since $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ is abelian, a lift $\lambda : T \to K$ with $\lambda\kappa = \tau$ exists. Since $\kappa$ is a mono, by Proposition IV.3.4 we can compute the lift $\lambda$ as honest representative of the honest morphism

$$\tau'\kappa^{-1},$$

where $\kappa^{-1}$ denotes the generalized inverse in $\mathrm{Mor}_{\mathrm{G}^{\mathrm{S}}\left(\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})\right)}$ of $\kappa$ and $\tau'$ the corresponding honest span of $\tau$ in $\mathrm{Mor}_{\mathrm{G}^{\mathrm{S}}\left(\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})\right)}$. By Proposition IV.5.8 we can also compose the following two spans to compute a representative of $\lambda$:

$$\begin{tikzcd}
T \arrow[rr, dashed, "\tau"] \arrow[rrrr, dashed, bend left, "\lambda"] & & A \arrow[rr, dashed, "\kappa^{-1}"] & & K \\
& Z \arrow[ul, "\tau_1"] \arrow[ur, "\tau_2"'] & & K \arrow[ul, "\widetilde{\kappa}"] \arrow[ur, "\mathrm{id}_K"'] &
\end{tikzcd}$$

We now give the constructions and proofs for the cokernel. Let $\varphi \in \mathrm{Mor}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}$ be represented by the span $A \xleftarrow{\alpha} X \xrightarrow{\beta} B$ in $\mathcal{A}$. For the cokernel projection let

$$\widetilde{\gamma} := \mathrm{CokernelProjection}_{\mathcal{A}}(\beta).$$

We set $\gamma := \mathrm{CokernelProjection}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}(\varphi)$ to be represented by the span

$$\begin{tikzcd}
B \arrow[rr, dashed, "\gamma"] & & C \\
& B \arrow[ul, "\mathrm{id}_B"] \arrow[ur, "\widetilde{\gamma}"'] &
\end{tikzcd}$$

The composition $\varphi\gamma$ can be displayed by the following diagram:

$$\begin{tikzcd}
A \arrow[rr, dashed, "\varphi"] & & B \arrow[rr, dashed, "\gamma"] & & C \\
& X \arrow[ul, "\alpha"] \arrow[ur, "\beta"'] & & B \arrow[ul, "\mathrm{id}_B"] \arrow[ur, "\widetilde{\gamma}"'] & \\
& & Y \arrow[ul, "\epsilon_1"] \arrow[ur, "\epsilon_2"'] & &
\end{tikzcd}$$

Setting $Y = X$ and $\epsilon_2 = \beta$ we get a valid fiber product diagram. It follows that the composition $\varphi\gamma$ is zero.

We now construct the cokernel colift. Let $\tau : B \to T$ in $\mathrm{Mor}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}$ represented by the span $B \xleftarrow{\tau_1} Z \xrightarrow{\tau_2} T$ in $\mathcal{A}$ such that $\varphi\tau$ is zero. Since $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ is abelian, a colift $\lambda : C \to T$ with $\gamma\lambda = \tau$ exists. Since $\gamma$ is an epimorphism, by Proposition IV.3.4 we can compute the colift $\lambda$ as an honest representative of the honest morphism

$$\gamma^{-1}\tau',$$

where $\gamma^{-1}$ denotes the generalized inverse in $\mathrm{Mor}_{\mathrm{G}^{\mathrm{S}}\left(\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})\right)}$ of $\gamma$ and $\tau'$ the corresponding honest span of $\tau$ in $\mathrm{Mor}_{\mathrm{G}^{\mathrm{S}}\left(\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})\right)}$. By Proposition IV.5.8 we can also compose the following two spans to compute $\lambda$:

$$\begin{tikzcd}
C \arrow[rr, dashed, "\gamma^{-1}"] \arrow[rrrr, dashed, bend left, "\lambda"] & & B \arrow[rr, dashed, "\tau'"] & & T \\
& B \arrow[ul, "\widetilde{\gamma}"] \arrow[ur, "\mathrm{id}_B"'] & & Z \arrow[ul, "\tau_1"] \arrow[ur, "\tau_2"'] &
\end{tikzcd}$$

□

**Theorem 5.11.** *The category $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ is computable abelian.*

Proof. The constructions of lifts along monos and colifts along epis are similar to the ones for the kernel lift and the cokernel colift.

We first provide a construction for the lift. Let $\varphi : A \hookrightarrow B$ in $\mathrm{Mor}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}$ a monomorphism, with cokernel projection $\psi : B \to C$ and $\tau : T \to B$ such that $\tau\psi$ is zero. By Theorem IV.5.2 $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ is abelian, which means that there is a morphism $\lambda : T \to A$ in $\mathrm{Mor}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}$ with $\lambda\varphi = \tau$. Let $\tau', \varphi' \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}$ be the generalized morphisms by spans that represent $\tau$ and $\varphi$, respectively. Then we can compute the lift $\gamma$ of $\tau$ along $\varphi$, i.e.,

$$\gamma := \mathrm{LiftAlongMonomorphism}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}(\tau, \varphi)$$

as the image under GF of

$$\tau'\varphi'^{-1},$$

where $\varphi'^{-1}$ denotes the pseudo-inverse of $\varphi'$. The well-definedness of this construction and the fact that $\gamma\varphi = \tau$ follows from Proposition IV.5.8 and Proposition IV.3.4.

To construct the colift, let $\varphi : A \twoheadrightarrow B$ in $\mathrm{Mor}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}$ be an epimorphism with kernel embedding $\psi : K \to A$ and $\tau : A \to T$ such that $\psi\tau$ is zero. By Theorem IV.5.2, $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ is abelian, which means that there is a morphism $\lambda : B \to T$ in $\mathrm{Mor}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}$ with $\varphi\lambda = \tau$. Let $\tau', \varphi' \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{S}}(\mathcal{A})}$ be the spans that represent $\tau$ and $\varphi$, respectively. Then we can compute the colift $\gamma$ of $\tau$ along $\varphi$, i.e.,

$$\gamma := \mathrm{ColiftAlongEpimorphism}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}(\varphi, \tau)$$

as the image under GF of

$$\varphi'^{-1}\tau',$$

where $\varphi'^{-1}$ denotes the pseudo-inverse of $\varphi'$. The well-definedness of this construction and the fact that $\varphi\gamma = \tau$ follows from Proposition IV.5.8 and Proposition IV.3.4. ☐

All categorical operations for $\mathrm{G}^{\mathrm{S}}(\mathcal{A})$ are implemented in Cap and can be found in Appendix F.10.

**5.b. Decidability.** For Serre morphisms, mono-, epi-, and isomorphisms are always decidable.

Remark 5.12. The category $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ has decidable zeros.

Proof. Let $A \in \mathrm{Obj}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}$. Then, by the construction of $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ we have $A \cong 0 \in \mathrm{Obj}_{\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})}$ if and only if $A \in \mathrm{Obj}_{\mathcal{C}}$. Since membership of $\mathrm{Obj}_{\mathcal{C}}$ is decidable, $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ has decidable zeros. ☐

We can now deduce from Proposition II.8.7 and Corollary II.9.3 that $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ has decidable monomorphisms, epimorphisms, and isomorphisms.

**Corollary 5.13.** *The category $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ has decidable monomorphisms, epimorphisms, and isomorphisms.*

**5.c. Computability of Serre morphisms by cospans.** In this section, we want to state the computability of the category of Serre morphisms by cospans and therefore provide the third data structure for $\mathcal{A}/\mathcal{C}$. We do not give proofs here, since they are all dual to the proofs for spans. Still we make all constructions explicit.

**Theorem 5.14.** *The category* $\overline{\mathrm{G}}^{\mathrm{C}}_{\mathcal{C}}(\mathcal{A})$ *is computable abelian.*

Before we give the explicit constructions, there is one construction left to mention: the dual of the common restriction.

**Definition 5.15.** Let $\varphi_1, \varphi_2 \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A})}$ with $\mathrm{Range}(\varphi_1) = \mathrm{Range}(\varphi_2)$ and $\varphi_i$ represented by the cospan $B_i \xrightarrow{\alpha_i} X_i \xleftarrow{\beta_i} A$ and

$$\iota_i := \mathrm{InjectionOfCofactorOfPushout}_{\mathcal{A}}((\beta_1, \beta_2), i).$$

The **common coarsening** $\mathrm{CommonCoarsening}(\varphi_1, \varphi_2)$ of $\varphi_1$ and $\varphi_2$ is the pair of generalized morphisms $(\varphi_1', \varphi_2')$ represented by $\mathrm{Arrow}(\varphi_i') := \alpha_i \iota_i$ and $\mathrm{ReversedArrow}(\varphi_i') := \beta_i \iota_i$.

$$\begin{tikzcd}
B_1 \arrow[rrd, "\alpha_1"'] \arrow[rrrdd, dashed, bend left, "\varphi_1"] & & & \\
& & X_1 \arrow[ld, "\iota_1"'] & \\
& Z & & A \arrow[lu, "\beta_1"'] \arrow[ld, "\beta_2"] \\
& & X_2 \arrow[lu, "\iota_2"] & \\
B_2 \arrow[rru, "\alpha_2"] \arrow[rrruu, dashed, bend right, "\varphi_2"'] & & &
\end{tikzcd}$$

**Proposition 5.16.** *Let* $\varphi_1 : A \to C$, $\varphi_2 : B \to C$ *in* $\mathrm{Mor}_{\mathrm{G}^{\mathrm{C}}(\mathcal{A})}$, *and* $(\varphi_1', \varphi_2') := \mathrm{CommonCoarsening}(\varphi_1, \varphi_2)$. *Then*

$$\varphi_1 = \varphi_1' \text{ and } \varphi_2 = \varphi_2' \text{ in } \overline{\mathrm{G}}^{\mathrm{C}}_{\mathcal{C}}(\mathcal{A}).$$

The proof is dual to that of Proposition IV.5.7 of the common restriction.

Sketch of proof of IV.5.14. We give sketches for the categorical constructions in $\overline{\mathrm{G}}^{\mathrm{C}}_{\mathcal{C}}(\mathcal{A})$.

(1) Some algorithms for categorical operations in $\overline{\mathrm{G}}^{\mathrm{C}}_{\mathcal{C}}(\mathcal{A})$ are, as for $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$, inherited from the underlying generalized morphism category $\mathrm{G}^{\mathrm{C}}(\mathcal{A})$. Those constructions

are:

$$\begin{gathered}\text{IsEqualForObjects}\\ \text{IdentityMorphism}\\ \text{PreCompose}\\ \text{AdditionForMorphisms}\\ \text{AdditiveInverse}\\ \text{ZeroMorphism.}\end{gathered}$$

They are carried out by applying the corresponding construction from $\mathrm{G}^{\mathrm{C}}(\mathcal{A})$ to an object or a representation of a morphism in $\overline{\mathrm{G}}^{\mathrm{C}}_{\mathcal{C}}(\mathcal{A})$.

(2) Some algorithms for categorical operations in $\overline{\mathrm{G}}^{\mathrm{C}}_{\mathcal{C}}(\mathcal{A})$ are, as for $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$, inherited from the underlying abelian category $\mathcal{A}$, by taking either the resulting object or the corresponding honest cospan of the result in $\mathcal{A}$ and interpreting those as objects or representatives of morphisms in $\overline{\mathrm{G}}^{\mathrm{C}}_{\mathcal{C}}(\mathcal{A})$. Those constructions are:

$$\begin{gathered}\text{ZeroObject}\\ \text{DirectSum}\\ \text{InjectionOfCofactorOfDirectSum}\\ \text{ProjectionInFactorOfDirectSum}\,.\end{gathered}$$

(3) Some constructions for categorical operations in $\overline{\mathrm{G}}^{\mathrm{C}}_{\mathcal{C}}(\mathcal{A})$ are the same constructions as the corresponding ones in $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ as they are carried out by computing with the representing generalized morphism in $\mathrm{G}^{\mathrm{C}}(\mathcal{A})$ of a morphism in $\overline{\mathrm{G}}^{\mathrm{C}}_{\mathcal{C}}(\mathcal{A})$ using only operations which are available for all generalized morphism categories. Those constructions are:

$$\begin{gathered}\text{IsEqualForMorphisms}\\ \text{KernelLift}\\ \text{CokernelColift}\\ \text{LiftAlongMonomorphism}\\ \text{ColiftAlongEpimorphism.}\end{gathered}$$

We make the remaining constructions explicit. To construct the kernel embedding and the cokernel projection, let $\varphi \in \mathrm{Mor}_{\overline{\mathrm{G}}^{\mathrm{C}}_{\mathcal{C}}(\mathcal{A})}$ represented by the cospan $A \xrightarrow{\alpha} X \xleftarrow{\beta} B$ in $\mathcal{A}$,

$$\kappa' := \mathrm{KernelEmbedding}_{\mathcal{A}}(\alpha)$$

the kernel embedding of $\alpha$, and

$$\pi' := \mathrm{CokernelProjection}_{\mathcal{A}}(\alpha)\,.$$

Then we define the kernel embedding $\kappa$ of $\varphi$ to be the Serre morphism in $\overline{\mathrm{G}}_{\mathcal{C}}^{\mathrm{C}}(\mathcal{A})$ represented by the cospan

$$\kappa : K \xrightarrow{\kappa'} A \xleftarrow{\mathrm{id}_A} A$$

and the cokernel projection $\pi$ of $\varphi$ to be the Serre morphism in $\overline{\mathrm{G}}_{\mathcal{C}}^{\mathrm{C}}(\mathcal{A})$ represented by the cospan

$$\pi : B \xrightarrow{\beta\pi'} C \xleftarrow{\mathrm{id}_C} C.$$

We now construct the universal morphism into the direct sum. Let $\varphi : A \to B$ and $\psi : A \to C$ be in $\mathrm{Mor}_{\overline{\mathrm{G}}_{\mathcal{C}}^{\mathrm{C}}(\mathcal{A})}$, $\varphi$ is represented by the cospan$A \overset{\alpha_1}{\to} X \overset{\beta_1}{\leftarrow} B$, and $\psi$ represented by the cospan $A \overset{\alpha_2}{\to} Y \overset{\beta_2}{\leftarrow} C$. Let furthermore

$$\alpha := \mathrm{UniversalMorphismIntoDirectSum}_{\mathcal{A}}(\alpha_1, \alpha_2).$$

Then the universal morphism into the direct sum $\gamma$ of $\varphi$ and $\psi$ is represented by the cospan

$$\begin{tikzcd} & X \oplus Y & \\ A \arrow[ur, "\alpha"] \arrow[rr, dashed, "\gamma"] & & B \oplus C \arrow[ul, "\beta_1 \oplus \beta_2"'] \end{tikzcd}$$

To construct the universal morphism from the direct sum, let $\varphi : A \to C$ and $\psi : B \to C$ in $\mathrm{Mor}_{\overline{\mathrm{G}}_{\mathcal{C}}^{\mathrm{C}}(\mathcal{A})}$ and

$$(\varphi', \psi') := \mathrm{CommonCoarsening}(\varphi, \psi)$$

their common coarsenings, represented by the spans $\varphi' : A \overset{\alpha_1}{\to} X \overset{\beta}{\leftarrow} C$ and $\psi : B \overset{\alpha_2}{\to} X \overset{\beta}{\leftarrow} C$. Let furthermore

$$\alpha := \mathrm{UniversalMorphismFromDirectSum}_{\mathcal{A}}(\alpha_1, \alpha_2).$$

Then the universal morphism from the direct sum $\gamma$ of $\varphi$ and $\psi$ is represented by the cospan

$$\begin{tikzcd} & X & \\ A \oplus B \arrow[ur, "\alpha"] \arrow[rr, dashed, "\gamma"] & & C \arrow[ul, "\beta"'] \end{tikzcd}$$

□

**5.d. Computability of Serre morphisms by 3-arrows.** For the sake of completeness, we want to give explicit constructions for the category of Serre morphisms modeled by 3-arrow generalized morphisms to be computable abelian, as stated in IV.5.1.

Before, we need to give the notion of common restrictions and coastrictions for 3-arrows.

**Definition 5.17.** Let $\varphi_1, \varphi_2 \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A})}$.

(1) Assume $\mathrm{Source}(\varphi_1) = \mathrm{Source}(\varphi_2)$. Then the **common restriction** $(\varphi_1', \varphi_2') := \mathrm{CommonRestriction}(\varphi_1, \varphi_2)$ of $\varphi_1$ and $\varphi_2$ is defined as follows: Let

$$\pi_i := \mathrm{ProjectionInFactorOfFiberProduct}_{\mathcal{A}}\left(\left(\mathrm{SourceAid}(\varphi_1), \mathrm{SourceAid}(\varphi_2)\right), i\right).$$

Then set

$$\begin{aligned} \mathrm{SourceAid}(\varphi_i') &:= \mathrm{PreCompose}(\pi_i, \mathrm{SourceAid}(\varphi_i)), \\ \mathrm{Arrow}(\varphi_i') &:= \mathrm{PreCompose}(\pi_i, \mathrm{Arrow}(\varphi_i)), \\ \mathrm{RangeAid}(\varphi_i') &:= \mathrm{RangeAid}(\varphi). \end{aligned}$$

(2) Assume $\mathrm{Range}(\varphi_1) = \mathrm{Range}(\varphi_2)$. Then the **common coarsening** $(\varphi_1', \varphi_2') := \mathrm{CommonCoarsening}(\varphi_1, \varphi_2)$ of $\varphi_1$ and $\varphi_2$ is defined as follows: Let

$$\iota_i := \mathrm{InjectionOfCofactorOfPushout}_{\mathcal{A}}\left(\left(\mathrm{RangeAid}(\varphi_1), \mathrm{RangeAid}(\varphi_2)\right), i\right).$$

Then set

$$\begin{aligned} \mathrm{SourceAid}(\varphi_i') &:= \mathrm{SourceAid}(\varphi_i), \\ \mathrm{Arrow}(\varphi_i') &:= \mathrm{PreCompose}_{\mathcal{A}}(\mathrm{Arrow}(\varphi_i), \iota_i), \\ \mathrm{RangeAid}(\varphi_i') &:= \mathrm{PreCompose}_{\mathcal{A}}(\mathrm{RangeAid}(\varphi_i), \iota_i). \end{aligned}$$

**Proposition 5.18.** *Let* $\varphi_1, \varphi_2 \in \mathrm{Mor}_{\mathrm{G}^{\mathrm{T}}(\mathcal{A})}$.

*(1) If* $\mathrm{Source}(\varphi_1) = \mathrm{Source}(\varphi_2)$ *and* $(\varphi_1', \varphi_2') = \mathrm{CommonRestriction}(\varphi_1, \varphi_2)$. *Then* $\varphi_1 = \varphi_1'$ *and* $\varphi_2 = \varphi_2'$ *in* $\overline{\mathrm{G}}_{\mathcal{C}}^{\mathrm{T}}(\mathcal{A})$.

*(2) If* $\mathrm{Range}(\varphi_1) = \mathrm{Range}(\varphi_2)$ *and* $(\varphi_1', \varphi_2') = \mathrm{CommonCoarsening}(\varphi_1, \varphi_2)$. *Then* $\varphi_1 = \varphi_1'$ *and* $\varphi_2 = \varphi_2'$ *in* $\overline{\mathrm{G}}_{\mathcal{C}}^{\mathrm{T}}(\mathcal{A})$.

The proof is again analogous to that of Proposition IV.5.7.

Sketch of proof of IV.5.1. We give sketches for the constructions in $\overline{\mathrm{G}}_{\mathcal{C}}^{\mathrm{T}}(\mathcal{A})$. The proofs and a longer description of the constructions can be found in [**BLH14b**, 1.1]. The implemented algorithms can be found in Appendix F.11.

(1) Some algorithms for categorical operations in $\overline{\mathrm{G}}_{\mathcal{C}}^{\mathrm{T}}(\mathcal{A})$ are, as for $\overline{\mathrm{G}}_{\mathcal{C}}^{\mathrm{S}}(\mathcal{A})$, inherited from the underlying generalized morphism category $\mathrm{G}^{\mathrm{T}}(\mathcal{A})$. Those constructions

are:

$$\begin{gathered}\text{IsEqualForObjects}\\ \text{IdentityMorphism}\\ \text{PreCompose}\\ \text{AdditionForMorphisms}\\ \text{AdditiveInverse}\\ \text{ZeroMorphism}.\end{gathered}$$

They are carried out by applying the corresponding construction from $\mathrm{G}^{\mathrm{T}}(\mathcal{A})$ to an object or a representation of a morphism in $\overline{\mathrm{G}}^{\mathrm{T}}_{\mathcal{C}}(\mathcal{A})$.

(2) Some algorithms for categorical operations in $\overline{\mathrm{G}}^{\mathrm{T}}_{\mathcal{C}}(\mathcal{A})$ are, as for $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$, inherited from the underlying abelian category $\mathcal{A}$, by taking either the resulting object or the corresponding honest cospan of the result in $\mathcal{A}$ and interpreting those as objects or representatives of morphisms in $\overline{\mathrm{G}}^{\mathrm{T}}_{\mathcal{C}}(\mathcal{A})$. Those constructions are:

$$\begin{gathered}\text{ZeroObject}\\ \text{DirectSum}\\ \text{InjectionOfCofactorOfDirectSum}\\ \text{ProjectionInFactorOfDirectSum}\,.\end{gathered}$$

(3) Some algorithms for categorical operations in $\overline{\mathrm{G}}^{\mathrm{T}}_{\mathcal{C}}(\mathcal{A})$ are the same constructions as the corresponding ones in $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$ as they are carried out by computing with the representing 3-arrow generalized morphism in $\mathrm{G}^{\mathrm{T}}(\mathcal{A})$ of a morphism in $\overline{\mathrm{G}}^{\mathrm{T}}_{\mathcal{C}}(\mathcal{A})$ using only operations which are available for all generalized morphism categories. Those constructions are:

$$\begin{gathered}\text{IsEqualForMorphisms}\\ \text{KernelLift}\\ \text{CokernelColift}\\ \text{LiftAlongMonomorphism}\\ \text{ColiftAlongEpimorphism}.\end{gathered}$$

We sketch the remaining four constructions:

To construct the kernel embedding and the cokernel projection let $\varphi : A \to B$ in $\mathrm{Mor}_{\overline{\mathrm{G}}^{\mathrm{T}}_{\mathcal{C}}(\mathcal{A})}$ represented by the normalized 3-arrow generalized morphism representative

$$\begin{tikzcd}
A \arrow[r, dashed, "\varphi"] & B \arrow[d, two heads, "\pi"] \\
A' \arrow[u, hook, "\iota"] \arrow[r, "\alpha"'] & B''
\end{tikzcd}$$

Let $\kappa := \mathrm{KernelEmbedding}_{\mathcal{A}}(\alpha)$ and $\gamma := \mathrm{CokernelProjection}_{\mathcal{A}}(\alpha)$. We define the kernel embedding of $\varphi$ to be the corresponding honest 3-arrow of $\kappa\iota$, and the cokernel projection of $\varphi$ to be the corresponding honest 3-arrow of $\pi\gamma$.

To construct the universal morphism from the direct sum let $\varphi : A \to C, \psi : B \to C$ in $\mathrm{Mor}_{\overline{\mathrm{G}}^{\mathrm{T}}_{\mathcal{C}}(\mathcal{A})}$. The common coarsening $(\varphi', \psi')$ of $\varphi$ and $\psi$ are represented by the normalized 3-arrow generalized morphism representatives

$$\begin{tikzcd}
A \arrow[r, dashed, "\varphi'"] & C \arrow[d, two heads, "\pi"] & B \arrow[r, dashed, "\psi'"] & C \arrow[d, two heads, "\pi"] \\
A' \arrow[u, hook, "\iota_1"] \arrow[r, "\alpha_1"'] & C'' & B' \arrow[u, hook, "\iota_2"] \arrow[r, "\alpha_2"'] & C''
\end{tikzcd}$$

The universal morphism from direct sum

$$\gamma := \mathrm{UniversalMorphismFromDirectSum}_{\overline{\mathrm{G}}^{\mathrm{T}}_{\mathcal{C}}(\mathcal{A})}(\varphi, \psi)$$

of $\varphi$ and $\psi$ is then represented by the 3-arrow generalized morphism

$$\begin{tikzcd}
A \oplus B \arrow[r, dashed, "\gamma"] & C \arrow[d, two heads, "\pi"] \\
A' \oplus B' \arrow[u, hook, "\iota_1 \oplus \iota_2"] \arrow[r, "\{\alpha_1, \alpha_2\}"'] & C''
\end{tikzcd}$$

To construct the universal morphism into the direct sum let $\varphi : A \to B, \psi : A \to C$ in $\mathrm{Mor}_{\overline{\mathrm{G}}^{\mathrm{T}}_{\mathcal{C}}(\mathcal{A})}$. The common restriction $(\varphi', \psi')$ of $\varphi$ and $\psi$ are represented by the normalized 3-arrow generalized morphism representatives

$$\begin{tikzcd} A \arrow[r, dashed, "\varphi'"] & B \arrow[d, two heads, "\pi_1"] & A \arrow[r, dashed, "\psi'"] & C \arrow[d, two heads, "\pi_2"] \\ A' \arrow[u, hook, "\iota"] \arrow[r, "\alpha_1"'] & B'' & A' \arrow[u, hook, "\iota"] \arrow[r, "\alpha_2"'] & C'' \end{tikzcd}$$

The universal morphism into direct sum

$$\gamma := \mathrm{UniversalMorphismIntoDirectSum}_{\overline{\mathrm{G}}^{\mathrm{T}}_{\mathcal{C}}(\mathcal{A})}(\varphi, \psi)$$

of $\varphi$ and $\psi$ is then represented by the 3-arrow generalized morphism

$$\begin{tikzcd} A \arrow[r, dashed, "\gamma"] & B \oplus C \arrow[d, two heads, "\pi_1 \oplus \pi_2"] \\ A' \arrow[u, hook, "\iota"] \arrow[r, "\langle \alpha_1, \alpha_2 \rangle"'] & B'' \oplus C'' \end{tikzcd}$$

□

CHAPTER V

# The category of coherent sheaves over a toric variety

In this chapter we show that the category of coherent sheaves over a toric variety is computable. The coherent sheaf category over a toric variety $X$ will be modeled as a Serre quotient of the category of finitely presented graded modules over the Cox ring of $X$. In Chapter III we already saw that the category of f.p. graded modules over a computable ring is computable abelian. We first state the equivalence of the mentioned Serre quotient category and the category of coherent sheaves over a toric variety. Then we give an algorithm to decide the membership in the thick subcategory of the f.p. graded modules to show that the category of coherent sheaves over a normal toric variety is indeed computable.

## 1. Preliminaries from toric geometry

We are going to recall the main definitions from toric geometry. We follow the definitions and notations of [**CLS11**]. For the whole chapter, $\mathbb{K}$ will denote an algebraically closed field of characteristic 0.

**Definition 1.1** (Toric variety)**.** An $n$-dimensional **toric variety** $X$ over $\mathbb{K}$ is an irreducible algebraic variety over $\mathbb{K}$ in which an algebraic torus $T \cong (\mathbb{K}^*)^n$ can be embedded such that the torus is a dense open subset and the algebraic action of $T$ on itself by multiplication can be extended to an algebraic action on the whole variety.

**Definition 1.2.** Let $T \cong (\mathbb{K}^*)^n$ be an algebraic torus.

(1) A **character** of $T$ is a group homomorphism

$$T \to \mathbb{K}^*.$$

(2) Let $m \in \mathbb{Z}^n$. Then $m$ defines a character of $T$ by the map

$$\chi^m : \; T \to \mathbb{K}^*, \; (t_1, \dots, t_n) \mapsto t_1^{m_1} \cdots t_n^{m_n}.$$

The characters of $T$ form a lattice isomorphic to $\mathbb{Z}^n$, called the **character lattice** of $T$, which we will denote by $M$.

(3) A **one-parametric subgroup** of $T$ is a group homomorphism

$$\mathbb{K}^* \to T.$$

(4) Let $\nu \in \mathbb{Z}^n$. Then $\nu$ defines a one-parametric subgroup of $T$ by the map

$$\mathbb{K}^* \to T : \; \lambda \mapsto (\lambda^{\nu_1}, \dots, \lambda^{\nu_n}).$$

The one-parametric subgroups of $T$ form a lattice isomorphic to $\mathbb{Z}^n$, called the **lattice of one-parametric subgroups**, which we will denote by $N$.

REMARK 1.3. The lattices $M$ and $N$ are dual to each other. For $m \in M$ and $\nu \in N$ there exists a $k \in \mathbb{Z}$ such that their composition can be written as

$$m \circ \nu : \mathbb{K}^* \to \mathbb{K}^*,\ x \mapsto x^k.$$

We define the duality

$$\langle , \rangle : M \times N \to \mathbb{Z},\ (m, \nu) \mapsto k.$$

We now state the basic combinatorial notions in toric geometry.

**Definition 1.4.** Let $T$ be a torus and $N$ its lattice of one-parametric subgroups. For a finite subset $G \subseteq N$ the set

$$\mathrm{Cone}\,(G) := \sum_{g \in G} \mathbb{R}_{\geqslant 0} g \subseteq N \otimes \mathbb{R}$$

is a **rational polyhedral cone** or simply **cone**.

**Definition 1.5.** Let $\sigma \subseteq N \otimes \mathbb{R}$ be a cone.

(1) For every $m \in M$ such that the minimum $\min \{\langle m, \mu\rangle \mid \mu \in \sigma\}$ exists, the set
$$\tau := \{\nu \in \sigma \mid \langle m, \nu\rangle = \min \{\langle m, k\rangle \mid k \in \sigma\}\}$$
is called a **face** of $\sigma$, denoted by $\tau \preceq \sigma$.

(2) For a cone $\sigma$ the **dual cone**
$$\{m \in M \otimes \mathbb{R} \mid \langle m, \nu\rangle \geqslant 0,\ \nu \in \sigma\}$$
is denoted by $\sigma^\vee$.

From now on every cone $\sigma$ is **pointed**, i.e., $\{0\}$ is a face of $\sigma$.

**Definition 1.6.** Let $\sigma \subseteq N \otimes \mathbb{R}$ be a cone.

(1) A face $\rho \preceq \sigma$ of dimension one is called **ray**. The unique $u_\rho \in N$ such that $\tau \cap N = \mathbb{Z}_{\geqslant 0} u_\rho$ is called **ray generator** of $\tau$.

(2) The generators of all rays in a cone $\sigma$ generate $\sigma$. They are called the **ray generators** of $\sigma$.

**Definition 1.7.** A **fan** $\Sigma$ is a finite collection of *pointed* cones $\Sigma = \{\sigma_1, \dots, \sigma_r\}$, $\sigma_i \subseteq N \otimes \mathbb{R}$ such that for every cone $\sigma_i \in \Sigma$ all faces of $\sigma_i$ are contained in $\Sigma$, and the intersection of two cones $\sigma_i, \sigma_j \in \Sigma$ is a face of both cones.

**Lemma 1.8** (Gordan). *Let $\sigma$ be a cone. Then $S_\sigma := \sigma^\vee \cap M$ is a finitely generated semigroup.*

**Proposition 1.9.** *Let $\sigma \subset N \otimes \mathbb{R}$ be a cone. Then*

$$U_\sigma := \mathrm{Spec}\,(\mathbb{K}\,[\sigma^\vee \cap M])$$

*is an affine toric variety with torus $M \otimes_{\mathbb{Z}} \mathbb{K}^*$.*

**Example 1.10.** Let $n := 2$. Consider the cone

$$\sigma := \mathrm{Cone}\,(e_1, e_2)\,.$$

The cone

$$\tau := \mathrm{Cone}\,(e_1)$$

is a face of $\sigma$, and

$$\begin{aligned} \mathbb{K}\,[S_\sigma] &\cong \mathbb{K}\,[x,y] \\ \mathbb{K}\,[S_\tau] &\cong \mathbb{K}\left[x,y^{\pm 1}\right]. \end{aligned}$$

In general, let $\tau$ be a face of $\sigma$. Then the semigroup embedding $S_\sigma \hookrightarrow S_\tau$ gives the inclusion

$$\mathbb{K}\,[S_\sigma] \hookrightarrow \mathbb{K}\,[S_\tau]$$

which leads to the embedding

$$U_\tau = \mathrm{Spec}\,(\mathbb{K}\,[S_\tau]) \to U_\sigma = \mathrm{Spec}\,(\mathbb{K}\,[S_\sigma])\,.$$

Given a fan $\Sigma$ and two maximal cones $\sigma_1, \sigma_2 \in \Sigma$, one can glue the affine varieties $U_{\sigma_1}$ and $U_{\sigma_2}$ along $U_{\sigma_1 \cap \sigma_2} \cong U_{\sigma_1} \cap U_{\sigma_2}$. If one does the gluing for all cones $\sigma \in \Sigma$ one gets a colimit variety

$$X_\Sigma := \varinjlim_{\sigma \in \Sigma} U_\sigma.$$

The variety $X_\Sigma$ is the **toric variety** of the fan $\Sigma$. It is indeed toric since $\{0\} \in \Sigma$ by definition of the fan and $U_{\{0\}} = (\mathbb{K}^*)^n$. The torus is a dense open subset of the variety $X_\Sigma$, and the action is extended naturally since the inclusions are compatible and every $U_\sigma$ is toric with that torus. Furthermore, the maximal cones $\Sigma_{\max}$ of the fan $\Sigma$ define a torus invariant affine open covering of the variety.

**Example 1.11.** Let $n = 1$ and

$$\begin{aligned} \sigma_1 &:= \mathrm{Cone}\,(e_1)\,, \\ \sigma_2 &:= \mathrm{Cone}\,(-e_1)\,. \end{aligned}$$

Then we have

$$\tau := \sigma_1 \cap \sigma_2 = \{0\}$$

and

$$\Sigma := \{\sigma_1, \sigma_2, \tau\}$$

is a fan. One gets $X_\Sigma$ by the maps

$$\mathbb{K}[x] \to \mathbb{K}\left[x^{\pm 1}\right] \leftarrow \mathbb{K}\left[x^{-1}\right]$$

which leads to the embeddings

$$\begin{tikzcd} \mathbb{K} & \mathbb{K}^* \arrow[l, hook', "x \mapsfrom x"'] \arrow[r, hook, "x \mapsto x^{-1}"] & \mathbb{K}. \end{tikzcd}$$

So we have $X_\Sigma \cong \mathbb{P}^1$.

We denote the set of $i$-dimensional cones in a fan $\Sigma$ by $\Sigma(i)$. Furthermore, we assume that $\Sigma(n)$ is not empty, with $n = \dim X_\Sigma$. This translates to the fact that the toric variety $X_\Sigma$ has no torus factors.

**Definition 1.12.** Let $X_\Sigma$ be a toric variety with torus $T$. For $\rho \in \Sigma(1)$ the closed subvariety $D_\rho := \overline{U_\rho} \subset X_\Sigma$ is a **torus invariant Weil divisor.**

**Definition 1.13.** The **group of torus invariant Weil divisors** on a variety $X_\Sigma$ with torus $T$ is denoted by

$$\operatorname{Div}_T(X_\Sigma) := \bigoplus_{\rho \in \Sigma(1)} \mathbb{Z} D_\rho.$$

Every torus invariant principal divisor on a toric variety $X_\Sigma$ with torus $T$ can be expressed as the divisor of a character, i.e., for every principal divisor $D$ on $X_\Sigma$ there is a $m \in M$ such that

$$D = \operatorname{div}(\chi^m) := \sum_{\rho \in \Sigma(1)} \langle m, u_\rho \rangle D_\rho,$$

where $u_\rho$ denotes the unique ray generator of the ray $\rho$.

So there is a $\mathbb{Z}$-homomorphism

$$M \to \operatorname{Div}_T(X_\Sigma)$$

with the torus invariant principal divisors as its image.

**Theorem 1.14** (Class group). *Let $X_\Sigma$ be an $n$-dimensional toric variety with torus $T$ and without torus factors. Then there exists an exact sequence*

$$\begin{array}{rcl} 0 \to M & \to & \operatorname{Div}_T(X_\Sigma) \xrightarrow{[\cdot]} \operatorname{Cl}(X_\Sigma) \to 0 \\ m & \mapsto & \operatorname{div}(\chi^m) \end{array}$$

*where $\operatorname{Cl}(X_\Sigma)$ is the **class group** of $X_\Sigma$.*

The exactness on the left is due to the assumption that no torus factor exists.

**Definition 1.15.** For a toric invariant divisor $D$ on a toric variety $X_\Sigma$ we denote by $[D]$ its **class** in $\operatorname{Cl}(X_\Sigma)$.

**Definition 1.16.** The **Cox ring** $S_\Sigma$ of a toric variety $X_\Sigma$ is defined as

$$S_\Sigma := \mathbb{K}[x_\rho \mid \rho \in \Sigma(1)].$$

$S_\Sigma$ is a $\operatorname{Cl}(X_\Sigma)$-graded ring and the **degree** of $x_\rho$ is the element $[D_\rho] \in \operatorname{Cl}(X_\Sigma)$.

$S_\Sigma$ is a computable graded ring by the definition of $\operatorname{Cl}(X_\Sigma)$ and $S_\Sigma$.

**Definition 1.17.** Let $S_\Sigma = \mathbb{K}[x_\rho \mid \rho \in \Sigma(1)]$ be the Cox ring of the variety $X_\Sigma$. For a cone $\sigma \in \Sigma$ define

$$x^{\hat{\sigma}} := \prod_{\substack{\rho \in \Sigma(1) \\ \rho \notin \sigma}} x_\rho.$$

The **irrelevant ideal** of $X_\Sigma$ is the ideal

$$B(\Sigma) := \left\langle x^{\hat{\sigma}} \mid \sigma \in \Sigma \right\rangle.$$

**Proposition 1.18.** *Let $X_\Sigma$ be a toric variety and $A$ a graded $S_\Sigma$-module. Then there is a quasi-coherent sheaf $\widetilde{A}$ on $X_\Sigma$ such that for every $\sigma \in \Sigma$ the sections of $\widetilde{A}$ over $U_\sigma \subset X_\Sigma$ are*

$$\Gamma\left(U_\sigma, \widetilde{A}\right) = \left(A_{x^{\hat{\sigma}}}\right)_0.$$

The sheaf $\widetilde{A}$ is coherent if $A$ is a finitely generated graded $S_\Sigma$-module. On the other hand, a sheaf $\mathcal{F}$ on $X_\Sigma$ is coherent if there exists a finitely presented graded $S_\Sigma$-module $A$ such that

$$\widetilde{A} \cong \mathcal{F}.$$

## 2. Equivalence of Serre quotient and coherent sheaves

We state how the category of graded module presentations over the Cox ring $S$ of a toric variety $X_\Sigma$ relates to the category of coherent sheaves over $X_\Sigma$. Recall that $\mathbb{K}$ is an algebraically closed field of characteristic 0 and all toric varieties are normal and with no torus factor, i.e., their fans contain the cone $\{0\}$ and a full-dimensional cone.

**Notation.** For the rest of this section $X_\Sigma$ will denote an $n$-dimensional toric variety with fan $\Sigma$, $S$ its Cox ring (homogeneous coordinate ring in [**CLS11**]) with irrelevant ideal $B$ and degree group $G := \mathrm{Cl}\left(X_\Sigma\right)$.

We denote by $S$-grmod the category of finitely presented $G$-graded modules over $S$ which is computable by III.2.11, and denote the sheafification functor by

$$\mathrm{Sh} : S\text{-grmod} \to \mathfrak{Coh} X_\Sigma,\ \ A \mapsto \widetilde{A}.$$

**Theorem 2.1** ([**BLH14a**, Cor. 4.5])**.** *The sheafification functor* Sh *induces an equivalence*

$$S\text{-grmod}/S\text{-grmod}^0 \cong \mathfrak{Coh} X_\Sigma,$$

*where $S$-grmod$^0$ is the kernel of* Sh *and the left side is a Serre quotient.*

There is a local description for $S$-grmod$^0$, since normal toric varieties have a natural torus invariant affine cover.

**Theorem 2.2.** *Let $X_\Sigma$ be a toric variety with fan $\Sigma$. Then a graded $S_\Sigma$-module $A$ is in the kernel of* Sh *if and only if*

$$\Gamma\left(U_\sigma, \widetilde{A}\right) = 0$$

*for all maximal cones $\sigma \in \Sigma$. Here $U_\sigma$ denotes the affine subvariety coming from the maximal cone $\sigma$ in the fan $\Sigma$.*

Proof. Indeed a sheaf is zero iff it is zero on every subvariety of an affine cover. Since the $U_\sigma$ belonging to the maximal cones $\sigma \in \Sigma$ form an affine open cover, the claim follows. $\square$

We can also formulate Theorem V.2.2 in a completely module theoretic setup. We denote by $S_{\widehat{\sigma}}$ the ring $S$ localized at the monomial $x^{\widehat{\sigma}}$, i.e., the product of all indeterminates of $S$ which correspond to the rays in $\Sigma$ which are not in the cone $\sigma$.

**Theorem 2.3.** *Let $X_\Sigma$ be a normal toric variety with fan $\Sigma$, Cox ring $S$, and $A$ a f.p. graded $S$-module. Then $A$ sheafifies to zero if and only if for every maximal cone $\sigma \in \Sigma$ the $(S_{\widehat{\sigma}})_0$-module $(A_{\widehat{\sigma}})_0$ is zero. Here $(S_{\widehat{\sigma}})_0$ resp. $(A_{\widehat{\sigma}})_0$ denotes the degree zero part of the ring $S$ resp. the module $A$ localized at $x^{\widehat{\sigma}}$.*

Proof. We have

$$\Gamma\left(U_\sigma, \widetilde{A}\right) = (A_{\widehat{\sigma}})_0$$

by [**CLS11**, Prop. 5.3.3], so the claim follows by Theorem V.2.2. □

For smooth toric varieties there is also a global criterion for deciding whether a finitely presented graded module over the Cox ring sheafifies to zero.

**Theorem 2.4** ( [**CLS11**, Prop. 5.3.10])**.** *Let $X_\Sigma$ be a smooth toric variety with Cox ring $S$, $B$ the irrelevant ideal, and $A$ in $S$-*grmod*. Then $A$ is in the kernel of the sheafification functor* Sh *if and only if there exists an $\ell \in \mathbb{N}$ such that $B^\ell A = 0$.*

## 3. Deciding membership of the kernel of the sheafification functor

We now give two distinct algorithms for deciding the membership of the kernel of the sheafification functor Sh. The first algorithm decided the kernel membership in the case where $X_\Sigma \cong \mathbb{P}^n$ and uses the global criterion for the kernel membership in Theorem V.2.4. The second algorithm decides the kernel membership for every normal toric variety $X_\Sigma$ with no torus factors. The algorithm uses the local criterion for the kernel membership from Theorem V.2.2.

### 3.a. Hilbert polynomial for projective spaces.

For projective spaces, the following holds:

**Proposition 3.1.** *Let $X_\Sigma := \mathbb{P}^n$ and $S := \mathbb{K}[x_0, \dots, x_n]$ a graded polynomial ring with $\deg(x_i) = 1$, $i = 0, \dots, n$. Then $S$ is the Cox ring of $X_\Sigma$ and the sheafifications $\widetilde{A}$ and $\widetilde{B}$ of two f.p. graded $S$-modules $A$ and $B$ are isomorphic if and only if $A_{\geqslant d} \cong B_{\geqslant d}$ as graded $S$-modules for some $d \geqslant 0$. In particular, $\widetilde{A} = 0$ if and only if $A_{\geqslant d} \cong 0$ for some $d \geqslant 0$.*

*Let $H_A(t) \in \mathbb{Z}[[t]]$ be the Hilbert series of $A$. Then $\widetilde{A} \cong 0$ if and only if $H_A \in \mathbb{Z}[t]$, and therefore the Hilbert polynomial $h_A \in \mathbb{Q}[t]$ of $A$ is $0$.*

For projective spaces it is therefore enough to compute the Hilbert polynomial of a f.p. graded module $A$ in order to to decide whether the module sheafifies to 0 or not.

This theorem cannot even be extended to weighted projective spaces.

**Example 3.2.** Let $X_\Sigma := \mathbb{P}(1, 1, 2)$ the weighted projective space. Then the Cox ring of $X_\Sigma$ is given by $S := \mathbb{C}[x_1, x_2, x_3]$ with

$$\deg(x_1) = \deg(x_2) = 1,\ \deg(x_3) = 2$$

and irrelevant ideal $B = \langle x_1, x_2, x_3 \rangle$. Let

$$M := S(1) / (x_1 S(1) + x_2 S(1)).$$

The Hilbert series of $M$ is

$$H_M := \sum_{i=0}^{\infty} t^{2i-1},$$

so it has no Hilbert polynomial, but instead a quasi polynomial which is not 0. But we have

$$(M_{x_1})_0 = (M_{x_2})_0 = (M_{x_3})_0 = 0,$$

where $(M_{x_1})_0$ and $(M_{x_2})_0$ are zero because $x_1$ and $x_2$ respectively are invertible in $S_{x_1}$ and $S_{x_2}$ and $(M_{x_3})_0$ is zero since $x_3$ has degree 2 and therefore $M_{x_3}$ is zero in all even degrees. So it follows that $\widetilde{M} = 0$.

This means that for the general case, we have to fall back to Theorem V.2.2 to decide if a module sheafifies to 0.

**3.b. Global sections.** To decide the kernel membership of the sheafification functor for toric varieties we need to compute the graded parts $M_\alpha$ of a f.p. graded module $M$ over a graded Laurent polynomial ring $S$ in which all monomials are homogeneous.

Remark 3.3. While it seems natural that monomials in a graded polynomial ring are homogeneous, this cannot be deduced from Definition III.1.1. Let $S := \mathbb{Q}[x]$, with degree group $\mathbb{Z}$ such that $(x-1)^n$ is homogeneous and

$$\deg((x-1)^n) = n$$

for all $n \in \mathbb{Z}_{\geqslant 0}$. Then the only homogeneous monomial is $1 = x^0$.

**Proposition 3.4.** *Let $S$ be a graded Laurent polynomial ring and $M \in \mathrm{Obj}_{S\text{-grmod}}$. Let $f \in S$ be a monomial. Furthermore, let*

$$0 \leftarrow M \leftarrow F^0 \leftarrow F^1$$

*be a graded free presentation of $M$. Then $(M_f)_0 \cong 0$ as an $(S_f)_0$-module if and only if the cokernel of the induced localized map*

$$\left(F^0_f \leftarrow F^1_f\right)_0 = \left(F^0_f\right)_0 \leftarrow \left(F^1_f\right)_0$$

*is 0.*

Proof. Both localizing at a monomial and taking the degree zero part of a module or morphism are exact functors. So the localized sequence remains exact, and we can restrict that sequence to its degree 0 part, and get an exact sequence of $(S_f)_0$-modules. □

In order to compute $(M_f)_0$ we will show how the map

$$\left(F^0_f \leftarrow F^1_f\right)_0$$

can be computed from the map

$$F^0 \leftarrow F^1.$$

For the rest of this chapter we use the following notation.

**Notation.** $S := \mathbb{K}\left[x_1, \dots, x_k, x_{k+1}^{\pm 1}, \dots, x_n^{\pm 1}\right]$ is a Laurent polynomial ring graded by a finitely presented abelian group $G$ such that all monomials in $S$ are homogeneous. For any subset $S' \subset S$ we define $\mathrm{Mon}\,(S')$ as the subset of monomials in $S'$.

**3.c. A generating set for $S_0$.** We first compute a finite generating set of $S_0 := \{f \in S \mid \deg(f) = 0\}$ as a $\mathbb{K}$-algebra.

**Notation.** Let $r > 0$. Then $\langle , \rangle : \mathbb{Z}^r \times \mathbb{Z}^r \to \mathbb{Z}$ denotes the standard scalar product on $\mathbb{Z}^r$.

**Definition 3.5.** We call the epimorphism of $\mathbb{Z}$-modules

$$\varphi : \mathbb{Z}^n \to G, \; e_i \mapsto \deg(x_i)$$

the **grading function** of $S$.
We call the isomorphism of semigroups

$$\chi : \{m \in \mathbb{Z}^n \mid \langle m, e_i\rangle \geqslant 0, i = 1, \dots, k\} \to \mathrm{Mon}\,(S), \; m \mapsto x^m.$$

the **character function** of $S$.

**Proposition 3.6** (First description of $S_0$). *Suppose $S := \mathbb{K}\left[x_1, \dots, x_k, x_{k+1}^{\pm 1}, \dots, x_n^{\pm 1}\right]$ with a grading function $\varphi$ as above. Then the monoid*

$$T := \{m \in \mathbb{Z}^n \mid \varphi(m) = 0, \langle m, e_i\rangle \geqslant 0, i = 1, \dots, k\}$$

*is isomorphic to the monoid $\mathrm{Mon}\,(S_0)$ via the character function $\chi$ of $S$.*

Proof. Let $m \in T$. Then $m \in \{m \in \mathbb{Z}^n \mid \langle m, e_i\rangle \geqslant 0, i = 1, \dots, k\}$ as well, so $\chi|_T$ is well-defined. We now compute $\deg \chi(m)$. Since $\chi(m) = x^m$, we have

$$\deg \chi(m) = \sum_{i=1}^{n} \deg\left(x_i^{\langle m, e_i\rangle}\right) = \sum_{i=1}^{n} \langle m, e_i\rangle \deg(x_i) = \varphi(m) = 0.$$

Hence $\chi(T) \subseteq \mathrm{Mon}\,(S_0)$. Now, let $x^m \in \mathrm{Mon}\,(S_0)$. Then $\varphi(m) = 0$ and $\langle m, e_i\rangle \geqslant 0$ for all $i = 1, \dots, n$. So $\mathrm{Mon}\,(S_0) \subseteq \chi(T)$. The injectiveness follows since $\chi$ is injective. □

Since $\varphi(m) = 0$ means that $m \in \ker \varphi$ we can get a better description of the cone $T$.

**Corollary 3.7** (Second description of $S_0$). *Let $\varphi$ be the grading function of $S$ and $\psi$ the kernel embedding of $\varphi$. Then the sequence of $\mathbb{Z}$-modules*

$$0 \to M \xrightarrow{\psi} \mathbb{Z}^n \xrightarrow{\varphi} G \to 0, \qquad (\dagger)$$

*is exact and the monoid*

$$T' := \{m \in M \mid \langle \psi(m), e_i\rangle \geqslant 0, i = 1, \dots, k\}$$

*is isomorphic to the monoid $T$ from Proposition V.3.6 via $\psi$.*

Proof. This follows from Proposition V.3.6 and the exactness of the sequence ($\dagger$), more precisely from the facts that $\psi$ is injective and $\varphi \circ \psi = 0$. □

Remark 3.8. If $S$ is be the Cox ring of a toric variety $X_\Sigma$, the lattice $M$ from Corollary V.3.7 will be the character lattice of $X_\Sigma$.

**Definition 3.9.** We call the monoid $T'$ from Corollary V.3.7 the **monomial cone** of $S_0$.

**Notation.** From now on, $\psi : M \to \mathbb{Z}^n$ denotes the kernel of $\varphi$. Furthermore, $\psi^*$ denotes its transposed, i.e.,

$$\psi^* : \mathbb{Z}^n \to M$$

such that for all pairs $(m, \nu) \in M \times \mathbb{Z}^n$ we have

$$\langle \psi (m), \nu \rangle = \langle m, \psi^* (\nu) \rangle .$$

To get a finite generating set of $S_0$ as $\mathbb{K}$-algebra, we need a finite generating set of $T'$ as affine semigroup.

**Proposition 3.10.** *The monomial cone $T'$ of $S_0$ of a graded polynomial ring $S$ is a* ***saturated affine semigroup****.*

Proof. We can rewrite $T'$ as

$$T' := \{ m \in M \mid \langle m, \psi^* (e_i) \rangle \geqslant 0, i = 1, \ldots, k \} .$$

So $T'$ is given by a set of linear homogeneous inequations, and therefore is a saturated affine semigroup. □

The property that $T'$ is a saturated affine semigroup is important since it means that a finite generating set for $T'$ exists.

Using the finite generating set of $T'$ we can write down a finite generating set for $S_0$ as $\mathbb{K}$-algebra.

**Theorem 3.11** (Generating set of $S_0$)**.** *Let $T'$ be the monomial cone of $S_0$ and $H$ a finite generating set of $T'$ as a semigroup. Furthermore, let $\chi$ be the character function of $S$ and $\psi : M \to \mathbb{Z}^n$ the kernel of the grading function of $S$. Then the set $\chi (\psi (H))$ generates $S_0$ as a $\mathbb{K}$-subalgebra of $S$, i.e., $S_0 = \mathbb{K} [\chi (\psi (H))] \subset S$.*

Proof. We can already deduce from Proposition V.3.6 and Corollary V.3.7 that $\chi (\psi (T'))$ generates $S_0$ as a $\mathbb{K}$-vector space and as a $\mathbb{K}$-subalgebra of $S$. Since every element of $T'$ can be expressed as a finite sum of elements of $H$, the set $\chi (\psi (H))$ generates $S_0$ as a $\mathbb{K}$-subalgebra of $S$. □

**Example 3.12** (V.3.2 cont.)**.** We are going to compute the degree zero part of $S_{x_3}$.

The degrees of the indeterminates were

$$\deg (x_1) = \deg (x_2) = 1, \ \deg (x_3) = 2,$$

so the degree sequence looks as follows

$$\begin{tikzcd} 0 \arrow[r] & \mathbb{Z}^2 \arrow[r, "\begin{pmatrix} -1 & 1 & 0 \\ -2 & 0 & 1 \end{pmatrix}"] & \mathbb{Z}^3 \arrow[r, "\begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix}"] & \mathbb{Z} \arrow[r] & 0 \end{tikzcd}$$

where the right map is the map $\varphi$ from the section above, and the left map is $\psi$. Now, we have

$$T' := \left\{ m \in \mathbb{Z}^2 \mid m_1 \geqslant 0,\ -m_1 - 2m_2 \geqslant 0 \right\},$$

and its Hilbert basis is

$$H = \{(2,-1),\ (1,-1),\ (0,-1)\}.$$

Using $\psi$, we see that

$$(S_{x_3})_0 = \mathbb{C}\left[\frac{x_1^2}{x_3}, \frac{x_1x_2}{x_3}, \frac{x_2^2}{x_3}\right].$$

To compute this example in `GAP` we use the `ToricSheaves` package. We first create the graded ring $S$:

```
gap> S := HomalgFieldOfRationalsInSingular() * "x1..3";
Q[x1,x2,x3]
gap> S := GradedRing( S );
Q[x1,x2,x3]
(weights: yet unset)
gap> SetWeightsOfIndeterminates( S, [1,1,2] );
```

$S$ is now defined to be the graded ring $\mathbb{Q}[x_1, x_2, x_3]$, with degrees of indeterminates as above.

Now we compute the generators of $(S_{x_3})_0$. The input for the function will be the ring $S$ and a list of indeterminates to localize at. Here, we only localize at the third indeterminate.

```
gap> DegreeZeroMonomialsOfLocalizedRing( S, [3] );
[ [ 0, 2, -1 ], [ 1, 1, -1 ], [ 2, 0, -1 ] ]
```

Since there is no data structure for Laurent polynomial ring monomials, we only get the list of exponents. $[0, 2, -1]$ translates to the monomial $\frac{x_2^2}{x_3}$, and so on. We see this list coincides with the list of ring generators given above, up to permutation.

**3.d. Relations between the generators of $S_0$.** We now compute all relations between the monomial generators of $S_0$. Once we have computed the relations, we will be able to present $S_0$ as quotient of a polynomial ring $R$ with a binomial ideal $I$.

**Proposition 3.13.** *Let $r > 0$ and $m \in \mathbb{Z}^r$. Then there is a decomposition $m = m_+ - m_-$, such that $m_{+,i} \geqslant 0$, $m_{-,i} \geqslant 0$ and $m_{+,i}m_{-,i} = 0$ for all $i$.*

REMARK 3.14. For $m \in \mathbb{Z}^r$ the decomposition $m = m_+ - m_-$ is unique.

**Theorem 3.15** (Writing $S_0$ as a quotient of a polynomial ring)**.** *As before let $S := \mathbb{K}\left[x_1, \ldots, x_k, x_{k+1}^{\pm 1}, \ldots, x_n^{\pm 1}\right]$ be graded by $G$ with grading function $\varphi$ and $\psi: M \to \mathbb{Z}^n$ be the kernel of $\varphi$. Furthermore, let $H \subseteq M$ be a generating set of the monomial cone of $S_0$ and $\psi(H) =: \{y_1, \ldots, y_r\}$ and $R := \mathbb{K}[y_1, \ldots, y_r]$ a free polynomial ring. Then the maps*

$$\kappa:\ \mathbb{Z}^r \to \mathbb{Z}^n,\ e_i \mapsto y_i$$

*and*

$$\chi_R : \ \{m \in \mathbb{Z}^r \mid \langle m, e_i \rangle \geqslant 0, i = 1, \dots, r\} \to \operatorname{Mon}(R)\, m \mapsto y^m.$$

*are well-defined and the map $\chi_R$ is the character function of $R$.*

*Furthermore, $S_0$ is the image of the map*

$$\beta : \ R \to S, \ y_i \mapsto \chi(y_i)$$

*and its kernel ideal is*

$$I := \langle \chi_R(m_+) - \chi_R(m_-) \mid m \in \ker \kappa \rangle .$$

Proof. First, the image of $\beta$ is $S_0$, since the $y_i's$ form a Hilbert basis of the monomial cone of $S_0$. The fact that $I$ is the kernel of $\beta$ is proved in [**CLS11**, Prop. 1.1.9]. □

Remark 3.16. The kernel ideal $I$ can be computed using a generating set $B \subset \ker \kappa$. We have

$$I = \langle \chi_R(m_+) - \chi_R(m_-) \mid m \in B \rangle : \left\langle \prod_{i=1}^{r} y_i \right\rangle .$$

For a proof, see [**Stu96**, p. 155].

**Example 3.17** (V.3.12 cont.)**.** Using the algorithms described above, we see that

$$(S_{x_3})_0 = \mathbb{C}\left[\frac{x_1^2}{x_3}, \frac{x_1 x_2}{x_3}, \frac{x_2^2}{x_3}\right] \cong \mathbb{C}[x,y,z] / \langle xz - y^2 \rangle .$$

In GAP, we can again use the ToricSheaves package:

```
gap> RI := DegreeZeroPartOfRingAsQuotient( S, [ 3 ] );
[ [ [ 0, 2, -1 ], [ 1, 1, -1 ], [ 2, 0, -1 ] ],
  Q[t1,t2,t3]/( t2^2-t1*t3 ) ]
```

The command DegreeZeroPartOfRingAsQuotient computes both the quotient ring $R/I$ as well as a data structure for the above isomorphism.

```
gap> RI[ 2 ];
Q[t1,t2,t3]/( t2^2-t1*t3 )
```

We see that the computed ring $R/I$ is isomorphic to $\mathbb{C}[x,y,z]/\langle xz - y^2\rangle$. Next we look at the isomorphism data structure:

```
gap> RI[ 1 ];
[ [ 0, 2, -1 ], [ 1, 1, -1 ], [ 2, 0, -1 ] ]
```

Here again, each tuple corresponds to a monomial in $(S_{x_3})_0$. Since the first computed tuple is $[0, 2, -1]$, the generator $t_1$ of the computed ring $R/I$ corresponds to the monomial $\frac{x_2^2}{x_3} \in (S_{x_3})_0$.

**Algorithm 3.18.** Computing $S_0$ as a monomorphism $R/I \to S$:

(1) Compute a resolution of the grading group $G$, i.e., a sequence

$$0 \to M \xrightarrow{\psi} \mathbb{Z}^n \xrightarrow{\varphi} G \to 0.$$

(2) Take a matrix $P \in \mathbb{Z}^{k\times n}$ which represents $\psi$ for a basis $(e_1, \dots, e_n)$ of $\mathbb{Z}^n$ that has $\varphi(e_i) = \deg(x_i)$. Create the cone $T'$ by the appropriate columns of $P$ as inequalities.
(3) Compute a Hilbert basis $H$ for the cone $T'$.
(4) Construct the mapping $\kappa$ representing it by a matrix $Q \in \mathbb{Z}^{r\times n}$ with rows $H$.
(5) Compute a generating set for the kernel of $\kappa$.
(6) Compute $I$ using Remark V.3.16.
(7) Compute the monomorphism $R/I \to S$ induced by $\beta$.

**3.e. The homogeneous parts of $S$.** We now want to compute a presentations for the $S_0$-modules $S_\alpha$ for $\alpha \in G$. To compute with the presentations of the $S_0$-modules $S_\alpha$ in the sense of Chapter III, we will present $S_\alpha$ as $R/I$-module, together with their embeddings in $S$. Since $S$ itself is not finitely generated as $S_0$ module, the embedding $S_\alpha \hookrightarrow S$ will be represented by the images of the generators of $S_\alpha$ in $S$. We start by establishing the necessary combinatorial notions.

**Proposition 3.19** (Tail cone decomposition)**.** *Let $P \subseteq \mathbb{Z}^n$ be a convex lattice polyhedron. Then there exists a cone $\mathcal{T}P$ and a polytope $P'$ such that $P = P' + \mathcal{T}P$. In this decomposition the cone $\mathcal{T}P$ is unique.*

**Definition 3.20.** The cone $\mathcal{T}P$ is called the **tail cone** of $P$.

Remark 3.21. Suppose, for $i = 1, \dots, k$, there are $a_i \in \mathbb{Z}^n$, $b_i \in \mathbb{Z}$ such that

$$P = \{m \in \mathbb{Z}^n \mid \langle m, a_i\rangle \geqslant b_i\}.$$

Then

$$\mathcal{T}P = \{m \in \mathbb{Z}^n \mid \langle m, a_i\rangle \geqslant 0\}.$$

Proof. Suppose $P' \subset P$, then $P' + \mathcal{T}P \subset P$ follows. The tail cone can be presented as the set

$$\{m \in \mathbb{Z}^n \mid \text{for all } p \in P: \ p + m \in P\}.$$

The claim follows by the linearity of the inequalities. □

**Proposition 3.22.** *Let $P = P' + \mathcal{T}P$ be a polyhedron and $H$ a generating set of $\mathcal{T}P$. Then, for every point $m \in P$ there is a point $m' \in P'$ and an $H$-indexed family $\{a_h \in \mathbb{Z}_{\geqslant 0} | h \in H\}$ with*

$$m = m' + \sum_{h\in H} a_h h.$$

Proof. Since $P = P' + \mathcal{T}P$ and since there is an $a \in \mathbb{Z}^H_{\geqslant 0}$ for every $c \in \mathcal{T}P$ such that

$$c = \sum_{h\in H} a_h h$$

the claim follows. □

Remark 3.23. Unfortunately, the decomposition in Proposition V.3.22 is not unique. Let

$$P := \left\{ m \in \mathbb{Z}^2 \mid m_2 \geqslant 0,\ m_2 - m_1 \geqslant -1 \right\}.$$

Then we have

$$\mathcal{T}\,P = \left\{ m \in \mathbb{Z}^2 \mid m_2 \geqslant 0,\ m_2 - m_1 \geqslant 0 \right\} = \operatorname{Cone}\left(e_2, e_1 + e_2\right)$$

and we can set

$$P' := \{0, e_1\}.$$

But then the point $e_1 + e_2$ can either be composed as

$$0 + (e_1 + e_2)$$

with $0 \in P$ and $e_1 + e_2 \in \mathcal{T}\,P$ or

$$(e_1) + (e_2)$$

with $e_1 \in P$ and $e_2 \in \mathcal{T}\,P$.

**3.f. A generating set for $S_\alpha$.** We will now describe the set $S_\alpha \cap \operatorname{Mon}(S) = \operatorname{Mon}(S_\alpha)$.

**Proposition 3.24.** *Let $a \in \varphi^{-1}(\alpha)$. Then the set*

$$\begin{aligned} G_\alpha &:= \left\{ m \in \mathbb{Z}^n \mid \varphi(m) = \alpha,\ \langle m, e_i \rangle \geqslant 0, i = 1, \ldots, k \right\} \\ &= \left\{ m + a \in \mathbb{Z}^n \mid \varphi(m) = 0,\ \langle m, e_i \rangle \geqslant \langle a, e_i \rangle, i = 1, \ldots, k \right\}. \end{aligned}$$

*defines a $\mathbb{K}$-basis of $S_\alpha$ via the morphism $\chi$.*

Proof. A $\mathbb{K}$-basis of $S_\alpha$ is given by the set $\operatorname{Mon}(S_\alpha)$. For a monomial

$$x^m = \chi(m) \in S_\alpha$$

the element $m \in \mathbb{Z}^n$ fulfills

(1) $\langle m, e_i \rangle \geqslant 0$ for all $i = 1, \ldots, k$ and
(2) $\varphi(m) = \alpha$.

Therefore $m \in G_\alpha$ and $\chi(G_\alpha) \supset \operatorname{Mon}(S_\alpha)$. Given a $m \in G_\alpha$, we have $\langle m, e_i \rangle \geqslant 0$ for $i = 1, \ldots, k$ and $\chi(m)$ is well-defined. Further

$$\deg(\chi(m)) = \varphi(m) = \alpha.$$

Thus $\chi(G_\alpha) \subset \operatorname{Mon}(S_\alpha)$ and finally $\chi(G_\alpha) = \operatorname{Mon}(S_\alpha)$. □

The set

$$G_\alpha - a = \left\{ m \in \mathbb{Z}^n \mid \varphi(m) = 0,\ \langle m, e_i \rangle \geqslant \langle a, e_i \rangle, i = 1, \ldots, k \right\}$$

is a polyhedron. Of course, $G_\alpha$ is also a polyhedron.

**Proposition 3.25.** *We have $G_\alpha - a \subset \ker \varphi$. Therefore we can rewrite $G_\alpha - a$ as*

$$G'_\alpha := \left\{ m \in M \mid \langle \psi(m), e_i \rangle \geqslant \langle a, e_i \rangle, i = 1, \ldots, k \right\}.$$

*$G'_\alpha$ is a polyhedron in $M$ with $\mathcal{T}\,G'_\alpha = T'$.*

Proof. We have $\psi(G'_\alpha) = G_\alpha - a$ since $\psi$ is a monomorphism, and by the inequations that define $T'$ the tail cone of $G'_\alpha$ is $T'$ (cf. Remark V.3.21). □

So we can write down a finite generating set of $S_\alpha$.

**Theorem 3.26.** *Let $G'_\alpha = G''_\alpha + T'$ a tail cone decomposition. Then the set*

$$B_\alpha := \chi\left(\psi\left(G''_\alpha\right) + a\right) \subseteq S$$

*is a generating set of $S_\alpha \subset S$ as an $S_0$-module.*

Proof. The set $\chi\left(\psi\left(G'_\alpha\right) + a\right)$ is a $\mathbb{K}$-Basis of $S_\alpha$. Since every point in $m' \in G'_\alpha$ is of the form

$$m' = m + t$$

with $m \in G''_\alpha$ and $t \in T'$ it follows that

$$\chi\left(\psi\left(m'\right) + a\right) = \chi\left(\psi\left(t\right) + \psi\left(m\right) + a\right) = \chi\left(\psi\left(m\right) + a\right) \chi\left(\psi\left(t\right)\right).$$

But $\chi\left(\psi\left(t\right)\right) \in \mathrm{Mon}\left(S_0\right)$ by definition of $T'$, so the set $B_\alpha$ is a generating set of $S_\alpha$. $\square$

**Example 3.27** (V.3.12 cont.)**.** The module $A$ is presented by the following map

$$S(0)^2 \xrightarrow{\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}} S(1).$$

We need to compute generating sets of $(S(0)_{x_3})_0$ and $(S(1)_{x_3})_0$, i.e., generators for $(S_{x_3})_0$ and $(S_{x_3})_1$. For $(S_{x_3})_0$ we have already computed $G'_0$, and we get

$$G''_0 = \{(0,0)\} \text{ and } B_0 := \{(0,0,0)\},$$

which is what we expected.

For $(S_{x_3})_1$ we choose the monomial corresponding to $a$ to be $x_1$, i.e., $a := (1,0,0)$, and get

$$G'_1 = \left\{m \in \mathbb{Z}^2 \mid -m_1 - 2m_2 \geqslant -1,\ m_1 \geqslant 0\right\}.$$

Then we compute

$$G''_1 = \{(0,0),\ (1,0)\},$$

using an integer linear program solver and obtain

$$B_1 = \{(1,0,0),\ (0,1,0)\},$$

which corresponds to the monomials $x_1$ and $x_2$. We therefore see that

$$(S(1)_{x_3})_0 \cong \langle x_1, x_2 \rangle_{(S_{x_3})_0}$$

as a $(S_{x_3})_0$-modules.

We use the function `MonomialsOfDegreePart` in `ToricSheaves` to compute the monomials of $(S(1)_{x_3})_0$. The first argument is again the graded polynomial ring $S$, the second is a list of variables to localize at, and the third is the degree part for which we want to compute monomial generators, i.e., 1.

```
gap> B1 := MonomialsOfDegreePart( S, [ 3 ], [ 1 ] );
[ [ 0, 1, 0 ], [ 1, 0, 0 ] ]
```

We see that $(S(1)_{x_3})_0$ is in fact generated by two monomials, namely $x_1$, which is represented by the tuple $[1, 0, 0]$, and $x_2$, which is represented by $[0, 1, 0]$.

**3.g. Relations between the generating monomials of** $\operatorname{Mon}(S_\alpha)$**.** Since we now have a finite generating set of $S_\alpha$ as an $S_0$-module, it remains to compute abstract relations between these generators of $S_\alpha$ in order to present $S_\alpha$ as an $R/I$-module. We first describe the type of relations that appear.

**Proposition 3.28.** *Let $\{x^{m_1}, \ldots, x^{m_h}\} \subset \operatorname{Mon}(S_\alpha)$ be a monomial generating set of $S_\alpha$ as an $S_0$-module. Then every relation*

$$(p_1, \ldots, p_h) \in S_0^h$$

*with*

$$\sum_{i=1}^{h} p_i x^{m_i} = 0 \in S$$

*is a $\mathbb{K}$-linear combination of relations of the form*

$$x^{s_i} e_i - x^{s_j} e_j \in S_0^h$$

*$i, j \in \{1, \ldots, h\}$ with $x^{s_i} \in \operatorname{Mon}(S_0)$ such that*

$$x^{s_i} x^{m_i} - x^{s_j} x^{m_j} = 0 \in S.$$

Proof. Let

$$p_i := \sum_{j \in \mathbb{Z}^n} p_{i,j} x^j.$$

Then

$$\begin{aligned} \sum_{i=1}^{h} p_i x^{m_i} &= \sum_{i=1}^{h} \sum_{j \in \mathbb{Z}^n} p_{i,j} x^j x^{m_i} \\ &= \sum_{i=1}^{h} \sum_{j \in \mathbb{Z}^n} p_{i,j-m_i} x^j \\ &= \sum_{j \in \mathbb{Z}^n} \left( \sum_{i=1}^{h} p_{i,j-m_i} \right) x^j \end{aligned}$$

which is zero if and only if

$$\left( \sum_{i=1}^{h} p_{i,j-m_i} \right) = 0$$

for all $j \in \mathbb{Z}^n$, since the monomials are $\mathbb{K}$-linear independent. So every relation is a $\mathbb{K}$-linear combination of relations of the form

$$(a_1 x^{n_1}, \ldots, a_h x^{n_h}) \in S_0^h$$

with $x^{n_i} \in \operatorname{Mon}(S_0)$ and $a_i \in k$. Now given a relation of this form, i.e.,

$$\sum_{j=1}^{h} a_j x^{n_j} x^{m_j} = 0. \tag{†}$$

We can w.l.o.g. assume that $n_i + m_i = n_j + m_j$ for all $i, j = 1, \ldots, h$, since otherwise we can separate the relations by the $\mathbb{K}$-linear independence of the monomials. We can also assume that $a_i \neq 0$ for all $i = 1, \ldots, h$. Then we have

$$\sum_{j=1}^{h} a_j x^{n_j} x^{m_j} = 0$$
$$\Leftrightarrow \sum_{j=1}^{h} a_j = 0.$$

Therefore (†) lies in the $\mathbb{K}$-linear span of the relations

$$\{x^{s_i} e_i - x^{s_j} e_j \mid x^{s_i} x^{m_i} = x^{s_j} x^{m_j},\ x^{s_k} \in \operatorname{Mon}(S_0)\}. \qquad \square$$

So we see that all relations between generators $g_i$ and $g_j$ of $S_\alpha$ are of the form

$$x^{n_i} g_i - x^{n_j} g_j$$

for some monomials $x^{n_i}$ and $x^{n_j}$ in $S_0$. Hence, to get the full set of relations of generators of $S_\alpha$ it is sufficient to compute all relations between pairs of two of them.

**Proposition 3.29.** *Given two points $g_1, g_2 \in G''_\alpha$. Then the polyhedron*

$$Q_{g_1,g_2} := (T' + g_1) \cap (T' + g_2)$$

*has tail cone $T'$.*

Proof. We have

$$\begin{aligned} T' + g_j &= \{m + g_j \mid m \in M,\ \langle m, \psi^*(e_i)\rangle \geqslant 0, i = 1, \ldots, k\} \\ &= \{m \in M \mid \langle m - g_j, \psi^*(e_i)\rangle \geqslant 0, i = 1, \ldots, k\} \\ &= \{m \in M \mid \langle m, \psi^*(e_i)\rangle \geqslant \langle g_j, \psi^*(e_i)\rangle, i = 1, \ldots, k\} \end{aligned}$$

for $j = 1, 2$. So

$$\begin{aligned} T' + g_1 \cap T' + g_2 = \{m \in M \mid \langle m, \psi^*(e_i)\rangle \geqslant \max(\langle g_1, \psi^*(e_i)\rangle, \langle g_2, \psi^*(e_i)\rangle), \\ i = 1, \ldots, k\}, \end{aligned}$$

and thus, by the inequalities that define the intersection, $T'$ is the tail cone of $Q_{g_1,g_2}$. $\square$

Using the polyhedron $Q_{g_1,g_2}$, we can describe all relations of two elements in the generating set $B_\alpha$ of $S_\alpha$.

**Theorem 3.30.** *Let $g_1, g_2 \in G''_\alpha$ like in Proposition V.3.29 and $Q_{g_1,g_2} = Q'_{g_1,g_2} + T'$ the tail cone decomposition of $Q_{g_1,g_2}$. Then the kernel of the $S_0$-module morphism*

$$\gamma :\ (S_0)^2 \to S_\alpha,\ e_i \mapsto \chi(\psi(g_i) + a)$$

*is the image of the map*

$$\delta: \left(S_0\right)^{Q'_{g_1,g_2}} \to \left(S_0\right)^2, \; e_q \mapsto \left(\chi\left(\psi\left(q-g_1\right)\right), -\chi\left(\psi\left(q-g_2\right)\right)\right).$$

Proof. We first show that $\gamma \circ \delta = 0$. Let $q \in Q_{g_1,g_2}$. Then

$$\begin{aligned}
\left(\gamma \circ \delta\right)\left(e_q\right) &= \gamma\left(\left(\chi\left(\psi\left(q-g_1\right)\right), -\chi\left(\psi\left(q-g_2\right)\right)\right)\right) \\
&= \chi\left(\psi\left(g_1\right)+a\right)\chi\left(\psi\left(q-g_1\right)\right) - \chi\left(\psi\left(g_2\right)+a\right)\chi\left(\psi\left(q-g_2\right)\right) \\
&= \chi\left(\psi\left(q\right)+a\right) - \chi\left(\psi\left(q\right)+a\right) \\
&= 0.
\end{aligned}$$

Now suppose $\left(x^{n_1}, -x^{n_2}\right)$ is in the kernel of $\gamma$. Then

$$\begin{aligned}
0 &= x^{n_1}\chi\left(\psi\left(g_1\right)+a\right) - x^{n_2}\chi\left(\psi\left(g_2\right)+a\right) \\
&= \chi\left(n_1+\psi\left(g_1\right)+a\right) - \chi\left(n_2+\psi\left(g_2\right)+a\right).
\end{aligned}$$

Since $\chi$ is injective, we get

$$n_1 + \psi\left(g_1\right) = n_2 + \psi\left(g_2\right).$$

The monomials $x^{n_1}, x^{n_2}$ are monomials in $S_0$, and therefore $n_1, n_2 \in T = \psi\left(T'\right)$. So there exist $n'_1, n'_2 \in T'$ with $\psi\left(n'_i\right) = n_i$. Also, since $\psi$ is injective, we have $n'_1 + g_1 = n'_2 + g_2$, hence $n'_1 + g_1 \in Q_{g_1,g_2}$. Therefore, there exist $q' \in Q'_{g_1,g_2}$ and $t' \in T'$ such that

$$q' + t' = n'_1 + g_1,$$

and

$$\begin{aligned}
\chi\left(\psi\left(t'\right)\right)\delta\left(q'\right) &= \chi\left(\psi\left(t'\right)\right)\left(\chi\left(\psi\left(q'-g_1\right)\right), -\chi\left(\psi\left(q'-g_2\right)\right)\right) \\
&= \left(\chi\left(\psi\left(t'+q'-g_1\right)\right), -\chi\left(\psi\left(t'+q'-g_2\right)\right)\right) \\
&= \left(x^{n_1}, -x^{n_2}\right).
\end{aligned}$$

Therefore $\left(x^{n_1}, -x^{n_2}\right)$ lies in the image of $\delta$, and the claim follows. □

**Theorem 3.31** (Presentation of $S_\alpha$). *$S_\alpha$ can be represented by the map*

$$\bigoplus_{g_i,g_j \in G''_\alpha} \left(S_0\right)^{Q'_{g_i,g_j}} \to \left(S_0\right)^{G''_\alpha}$$

*with*

$$e_{q_{i,j}} \mapsto x \text{ with } x_g := \begin{cases} \chi\left(\psi\left(q_{i,j}-g\right)\right) & g = g_i \\ -\chi\left(\psi\left(q_{i,j}-g\right)\right) & g = g_j \\ 0 & \end{cases}.$$

Proof. By Proposition V.3.28, all relations are linear combinations of binomial relations. So, by taking all the binomial relations, we get a presentation for $S_\alpha$. □

We now want to present $S_\alpha$ as an $R/I$-module, i.e., we have to translate the $S_0$-relations in $R'$ for the generators $G''_\alpha$ of $S_\alpha$ into relations in $R/I$.

Remark 3.32. Let $g_1, g_2 \in G''_\alpha$, $q \in Q'_{g_1,g_2}$ as above, and $\widetilde{m} \in \mathbb{Z}^r$ such that $\kappa(\widetilde{m}) = q - g_1$. Then every element $n$ in the set

$$L := \{n \in \mathbb{Z}^r \mid \kappa(n) = 0,\ \langle n, e_i\rangle \geqslant -\langle \widetilde{m}, e_i\rangle, i = 1, \dots, k\}$$

fulfills $\kappa(n + \widetilde{m}) = q - g_1$ and $\langle n + \widetilde{m}, e_i\rangle \geqslant 0$, where $\kappa$ was defined as

$$\kappa :\ \mathbb{Z}^r \to \mathbb{Z}^n,\ e_i \mapsto y_i.$$

Remember that $R := \mathbb{K}[y_1, \dots, y_r]$, and $y_i$ was the image of the $i$-th generator $h_i \in H$ of the monomial cone $T$ of $S_0$ under the mapping $\psi$. So, while $y_i$ is an indeterminate in a free polynomial ring, it is also an element of $\mathbb{Z}^n$.

**Proposition 3.33.** *For every $n \in L + \widetilde{m}$ we have*

$$(\beta \circ \chi_R)(n) = (\chi \circ \kappa)(n) = x^{q-g_1}.$$

We see that every element of the set $L + \widetilde{m}$ corresponds to both a monomial in $S$ and $R/I$ via the mappings in Proposition V.3.33. So the elements in $L$ provide the relations for the generators of $S_\alpha$ as an $(R/I)$-module.

Proof. We have

$$(\beta \circ \chi_R)(n) = \beta(y^n) = x^{\sum\langle n, e_i\rangle y_i}$$

and

$$(\chi \circ \kappa)(n) = x^{q-g_1}.$$

which proves the second equality. Furthermore

$$\sum \langle n, e_i\rangle y_i = \kappa(n) = q - g_1 \in R/I,$$

which concludes the proof. □

**Theorem 3.34** (Presentation of $S_\alpha$ as an $R/I$-module)**.** *Let $G''_\alpha$ be as above. Then $S_\alpha$ is isomorphic via $\beta$ to the cokernel of the mapping*

$$\bigoplus_{g_i, g_j \in G''_\alpha} (R/I)^{Q'_{g_1,g_2}} \to (R/I)^{G''_\alpha}$$

*with*

$$e_{q_{i,j}} \mapsto x \text{ with } x_g := \begin{cases} \chi_R(\ell_i) & g = g_i \\ -\chi_R(\ell_j) & g = g_j \\ 0 & \end{cases}.$$

*where $\ell_k$ is an element of the set $L + \widetilde{m}$ with $\alpha(\widetilde{m}) = q_{i,j} - g_k$ for $k = i, j$.*

Proof. By Theorem V.3.31, $S_\alpha$ is isomorphic to the cokernel of

$$\bigoplus_{g_i, g_j \in G''_\alpha} (S_0)^{Q'_{g_1,g_2}} \to (S_0)^{G''_\alpha}.$$

Since $\beta$ maps the relations from $S_0 \subset S$ to $R/I$, the claim follows. □

**Example 3.35** (V.3.27 cont.)**.** Since $(S(0)_{x_3})_0$ has only one generator as an $(S_{x_3})_0$-module, namely 1, $(S(0)_{x_3})_0$ is free. On the other hand, $(S(1)_{x_3})_0$ has two monomials as generators, so we need to compute their $(S_{x_3})_0$-relations.

We consider again the monomial cone of $(S_{x_3})_0$

$$\left\{m \in \mathbb{Z}^2 \mid -m_1 - 2m_2 \geqslant 0,\ m_1 \geqslant 0\right\},$$

and intersect the shifts by $g_1 := (0, 0)$ and $g_2 := (1, 0)$, respectively. The intersection polytope is

$$T' + g_1 \cap T' + g_2 = \left\{m \in \mathbb{Z}^2 \mid -m_1 - 2m_2 \geqslant 0,\ m_1 \geqslant 1\right\}.$$

The polytope is generated by the lattice points

$$Q'_{g_1,g_2} = \{(1, -1),\ (2, -1)\}.$$

which leads to the relations

$$\begin{pmatrix} \frac{x_1x_2}{x_3} & -\frac{x_1^2}{x_3} \end{pmatrix}$$

for $(1, -1)$ and

$$\begin{pmatrix} \frac{x_2^2}{x_3} & -\frac{x_1x_2}{x_3} \end{pmatrix}$$

for $(2, -1)$. Now, using the isomorphism

$$(S_{x_3})_0 \cong \mathbb{C}[x, y, z] / \left\langle xz - y^2 \right\rangle := R,$$

we can rewrite these relations as

$$\begin{pmatrix} y & -x \\ z & y \end{pmatrix} =: X.$$

Ultimately, we can present $(S(1)_{x_3})_0$ as the cokernel of

$$R^2 \xrightarrow{X} R^2.$$

Note that we know that the generators of this new representation correspond to $x_1$ and $x_2$.

Using `ToricSheaves`, we can compute the relation matrix for a presentation of $(S(1)_{x_3})_0$ as follows:

```
gap> S1 := DegreePartOfRing( S, [ 3 ], [ 1 ] );
[ [ [ 0, 1, 0 ], [ 1, 0, 0 ] ],
  <An unevaluated 2 x 2 matrix over a residue class ring> ]
```

The arguments for the function `DegreePartOfRing` are the same arguments as for the function`MonomialsOfDegreePart`. The output consist of the generating monomials seen in Example V.3.27, and a matrix., which describes the $R/I$-relations of the generators:

```
gap> Display( S1[ 2 ] );
t2, -t1,
-t3,-t2

modulo [ t2^2-t1*t3 ]
```

So we see, the relation matrix for $(S(1)_{x_3})_0$ is

$$\begin{pmatrix} t_2 & -t_1 \\ -t_3 & -t_2 \end{pmatrix}$$

over the ring $R/I := \mathbb{C}[t_1, t_2, t_3] / \langle t_2^2 - t_1 t_3 \rangle$.

**3.h. Homogeneous parts of finitely presented modules.** We now use the $R/I$-presentations of the graded parts $S_\alpha$ of $S$ to describe the homogeneous parts $A_\alpha$ of degree $\alpha \in G$ of a finitely presented graded $S$-module $A$. We first establish some well-known facts.

REMARK 3.36. Let $F \in \mathrm{Obj}_{S\text{-grpres}}$ be a free module. Then there are $\alpha_1, \ldots, \alpha_r \in G$ such that

$$F \cong \bigoplus_{i=1}^{s} S(\alpha_i).$$

$S(\alpha)$ is the shift of $S$ by $\alpha$, i.e., for any $\beta \in G$

$$S(\alpha)_\beta = S_{\alpha+\beta}.$$

**Notation.** For the rest of the section $\{y_1, \ldots, y_r\} = H \subset \mathrm{Mon}(S_0)$ will both be the generating set for $S_0$ and for the ring $R$ as $\mathbb{K}$-algebra.

Presenting the homogeneous parts $F_\alpha$ of a free $S$-module as $S_0$-modules can be done using the presentations of the homogeneous parts of $S$. Note that even if $F$ is a free $S$-module, the homogeneous parts $F_\alpha$ of $F$ are not free $S_0$-modules in general.

REMARK 3.37. Let $F \in \mathrm{Obj}_{S\text{-grpres}}$ be a free module, $\alpha \in G$, and

$$F \cong \bigoplus_{i=1}^{s} S(\alpha_i)$$

like in Remark V.3.36. Then

$$F_\alpha \cong \bigoplus_{i=1}^{s} S(\alpha_i)_\alpha = \bigoplus_{i=1}^{s} S_{\alpha_i+\alpha}$$

as $S_0$-modules.

**Proposition 3.38.** *Let $B_\alpha \subset \mathrm{Mon}(S)$ be an $S_0$-generating set of $S_\alpha$, $\widehat{x} \in \mathrm{Mon}(S_\alpha)$, and $H$ a finite generating set (as computed from $T'$) of the monomial cone $\mathrm{Mon}(S_0)$ of $S_0$. Then there is at least one $b \in B_\alpha$ such that a solution $a \in \mathbb{Z}_{\geqslant 0}^H$ with*

$$b \prod_{t \in H} t^{a_t} = \widehat{x}$$

*exists and can be computed.*

PROOF. Since $B_\alpha$ is an $S_0$-generating set of $S_\alpha$ and $H$ generates $S_0$ as a $\mathbb{K}$-algebra, the existence of $b$ and $a$ follows.

To find the solution $a$, assume $\widehat{x}$, $b$, and the elements of $H$ are presented by their exponent vectors. Then one needs to solve $Hz = \widehat{x} - b$ for $z \in \mathbb{Z}_{\geqslant 0}^H$. This can be done using an integer inequation solver. □

Now we can describe homogeneous parts of $A \in S$-grpres, i.e., a module $A$ given as the cokernel of a morphism of free graded modules

$$F_1 \xrightarrow{\varphi} F_0 \to A = \operatorname{coker}(\varphi) \to 0.$$

Since $\varphi$ is a graded $S$-module homomorphism, we get the following commutative diagram of $S_0$-modules:

$$\begin{tikzcd}
F_1 \arrow[r, "\varphi"] & F_0 \arrow[r, two heads, "\pi"] & A \arrow[r] & 0 \\
F_{1,\alpha} \arrow[u, hook, "\iota_1"] \arrow[r, "\widetilde{\varphi}"] & F_{0,\alpha} \arrow[u, hook, "\iota_0"] \arrow[r, two heads, "\widetilde{\pi}"] & A_\alpha \arrow[u, hook, "\iota"] \arrow[r] & 0
\end{tikzcd}$$

Let

$$F_1 \cong \bigoplus_{i=1}^{s_1} S(\alpha_i), \ F_0 \cong \bigoplus_{j=1}^{s_0} S(\beta_j),$$

and $B_{\alpha_i}, B_{\beta_j} \subset \operatorname{Mon}(S)$ the $S_0$-generating sets of $S(\alpha_i)_\alpha$ and $S(\beta_j)_\alpha$ respectively. The disjoint unions of the $B_{\alpha_i}$, $i = 1, \ldots, s_1$ and $B_{\beta_j}$, $j = 1, \ldots, s_0$ form generating sets for $F_{1,\alpha}$ and $F_{0,\alpha}$. Since we can compute the $R/I$-relations of $B_{\alpha_i}$ and $B_{\beta_i}$, we are able to present both $F_{0,\alpha}$ and $F_{1,\alpha}$ as $R/I$-modules by the methods in Subsection V.3.f.

We now want to write down a matrix for the map $\widetilde{\varphi}$ in terms of the generators of $F_{1,\alpha}$ and $F_{0,\alpha}$. Let $e_i$ be the generator of $F_1$ which corresponds to $S(\alpha_i)$, and

$$\{b_1, \ldots, b_u\} := B_{\alpha_i}.$$

Then we have

$$\varphi(b_\ell e_i) = b_\ell (f_{i,1}, \ldots, f_{i,s_0})$$

with

$$b_\ell f_{i,j} \in S(\beta_j)_\alpha,$$

where $f_{i,j}$ is the $i,j$-th entry of the matrix representing the morphism $\varphi$. Let

$$b_\ell f_{i,j} = \sum_{\widehat{x} \in \operatorname{Mon}\left(S_{\beta_j}\right)} c_{\widehat{x}} \widehat{x}.$$

For every $\widehat{x} \in \operatorname{Mon}\left(S_{\beta_j}\right)$ by Proposition V.3.38 we can compute a generator $b \in B_{\beta_j}$ and a monomial $s_{\widehat{x}} \in S_0$ such that

$$b_\ell f_{i,j} = \sum_{\widehat{x} \in \operatorname{Mon}\left(S_{\beta_j}\right)} c_{\widehat{x}} s_{\widehat{x}} b.$$

To write $b_\ell f_{i,j}$ as an $R/I$-linear combination consider again Proposition V.3.38. The monomial $s_{\widehat{x}} \in \operatorname{Mon}(S_0)$ comes with an $a_{\widehat{x}} \in \mathbb{Z}^r$ such that

$$s_{\widehat{x}} = \prod_{i=1}^{r} y_i^{a_{\widehat{x},i}} \in R/I.$$

Therefore, we can write $b_\ell f_{i,j}$ as an $R/I$-combination of the generators of $F_{0,\alpha}$. Now, for every $b \in \biguplus_{i=1}^{s_1} B_{\alpha_i}$ we know $\widetilde{\varphi}(b) \in F_{0,\alpha}$ written as an $R/I$-linear combinations in the generators $\biguplus_{j=1}^{s_0} B_{\beta_j}$ of $F_{0,\alpha}$ as $R/I$-module, and can therefore construct a matrix for $\widetilde{\varphi}$.

We summarize the steps to compute the $R/I$-presentation of $A_\alpha$ from the $S$-module $A$:

**Algorithm 3.39.** Computing $A_\alpha$ as an $R/I$-module from the graded $S$-module $A$:

(1) Compute $F_1$ and $F_0$ from $A$.
(2) Compute $R/I$, together with $\mathbb{K}$-algebra generators $H \subset \mathrm{Mon}(S_0)$.
(3) For $F_1$ and $F_0$, compute the $R/I$-generating sets $B_{\alpha_i}, i = 1, \dots, s_1$, and $B_{\beta_j}$, $j = 1, \dots, s_0$, together with $R/I$-presentations for $F_{1,\alpha}$ and $F_{0,\alpha}$.
(4) For each $b \in \biguplus_{i=1}^{r} B_{\alpha_i}$ compute the image $\widetilde{\varphi}(b)$ in terms of the $R/I$-generators $\biguplus_{j=1}^{s} B_{\beta_j}$ of $F_{0,\alpha}$.
(5) Compute $A_\alpha$ as the cokernel of $\widetilde{\varphi}$.

**Example 3.40** (V.3.35 cont.)**.** We now finish the computation of the new presentation matrix of $A$. Remember, $A$ was given by the cokernel of the map

$$S(0)^2 \xrightarrow{\begin{pmatrix} x_1 & x_2 \end{pmatrix}} S(1)$$

and we already know that, using the ring $R := \mathbb{C}[x, y, z] / \langle xy - z^2 \rangle$, we have

$$\begin{aligned} \left(S(0)_{x_3}\right)_0 &\cong R \text{ and} \\ \left(S(1)_{x_3}\right)_0 &\cong R^2/X \end{aligned}$$

with

$$X := \begin{pmatrix} y & -x \\ z & y \end{pmatrix}.$$

Now, using the presentation of $A$, we can reconstruct the localized map presenting $A$ on the 0-th degree level.

The first generator $e_1$ of $S(0)^2$ is mapped to $x_1$, which corresponds to our first generator of $\left(S(1)_{x_3}\right)_0$, which again corresponds to the first generator of $R^2/X$.

The second generator $e_2$ of $S(0)^2$ is mapped to $x_2$, which corresponds to the second generator of $R^2/X$.

So, we obtain the following presentation of $\left(A_{x_3}\right)_0$ as an $R$ module:

$$R^2 \xrightarrow{\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}} R^2/X$$

This morphism is clearly surjective, so its cokernel, which is isomorphic to $\left(A_{x_3}\right)_0$, is 0. This again concludes the fact that the module $A$ sheafifies to 0.

We now use `GAP` to compute a presentation matrix of $\left(A_{x_3}\right)_0$. We first create the Cap category of finitely presented graded modules, as described in Chapter III.

```
gap> Sgrmod := GradedLeftPresentations( S );
The category of graded f.p. modules over Q[x1,x2,x3]
  (with weights [ 1, 1, 2 ])
```

Now, to create the module, we first create its relation matrix $(x_1, x_2)$ over the ring $S$. We then create a graded module out of this matrix by specifying the degree of the generator of $A$. Since $A$ is a quotient of $S(1)$, the degree of the generator is $-1$.

```
gap> A := HomalgMatrix( "[ x1, x2 ]", 2, 1, S );
<A 2 x 1 matrix over a graded ring>
gap> A := AsGradedLeftPresentation( A,  [ [ -1 ] ] );
<An object in The category of graded f.p. modules over Q[x1,x2,x3]
  (with weights [ 1, 1, 2 ])>
```

Now we create the functor which computes the 0-th part of a graded $S_{x_3}$-module. As input we use the graded module category and a list that indicates which variables of the polynomial ring $S$ are localized, i.e., invertible.

```
gap> F := DegreeZeroPartOfLocalizationFunctor( Sgrmod, [ 3 ] );
Degree zero functor localized at [ 3 ]
```

Now we apply the functor to $A$ to get an $R/I$-module.

```
gap> A0 := ApplyFunctor( F, A );
<An object in Category of left presentations of
  Q[t1,t2,t3]/(t2^2-t1*t3 )>
```

When we examine the relation matrix of the module, we see that it is (up to ordering of the generators)[1] indeed the cokernel of the presentation map $(A_{x_3})_0$ above, in the sense we described the algorithm for the cokernel of a finitely presented module in Chapter III.

```
gap> Display( UnderlyingMatrix( A0 ) );
0,  1,
1,  0,
t2, -t1,
-t3,-t2

modulo [ t2^2-t1*t3 ]
```

[1]The ordering of the generators depends on the GAP session, since some of the underlying linear programming tools use randomness.

So, the relation matrix for $(A_{x_3})_0$ over the ring $R/I := \mathbb{C}[t_1, t_2, t_3] / \langle t_2^2 - t_1 t_3 \rangle$ is

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ t_2 & -t_1 \\ -t_3 & -t_2 \end{pmatrix}.$$

We now use these algorithms to model the category of coherent sheaves over a toric variety, since we can now decide whether a module sheafifies to 0.

**Example 3.41** (V.3.2 cont.)**.** We will now wrap up all examples from the previous sections together and show that the module $M$ from Example V.3.2 indeed sheafifies to 0.

We compute presentations for the affine sections of $M$. To compute $(M_{x_i})_0$, we first need to compute $(S_{x_i})_0$. For $i = 1, 2$ we have

$$(S_{x_1})_0 \cong (S_{x_2})_0 \cong \mathbb{C}[y_1, y_2] =: R_1,$$

since the subrings are generated by the monomials $x_1^{-1}x_2, x_1^{-2}x_3$ and $x_1 x_2^{-1}, x_2^{-2}x_3$. For both $i = 1, 2$ the module $(M_{x_i})_0$ can be presented by the $R_1$-module homomorphism

$$R_1^2 \xrightarrow{\begin{pmatrix} y_1 \\ 1 \end{pmatrix}} R_1^1,$$

which is an epimorphism, so $(M_{x_i})_0 \cong 0$ for $i = 1, 2$.

The ring $(S_{x_3})_0$ is generated by the monomials $x_1^2 x_3^{-1}, x_2^2 x_3^{-1}, x_1 x_2 x_3^{-1}$, and can therefore be presented as a quotient of a polynomial ring:

$$(S_{x_3})_0 \cong \mathbb{C}[y_1, y_2, y_3] / \left\langle y_1 y_2 - y_3^2 \right\rangle =: R_2.$$

The module $(M_{x_3})_0$ can be presented as the cokernel of the $R_2$-module epimorphism

$$R_2^2 \to R_2^2/N,$$

with

$$N := \langle (y_2, -y_3), (y_3, -y_1) \rangle.$$

Hence $(M_{x_3})_0 = 0$.

To carry out the example in `GAP`, we use the `ToricVarieties` package together with the `ToricSheaves` package. We first produce the fan of $\mathbb{P}(1,1,2)$ and create a toric variety out of it, using the `ToricVarieties` package.

```
gap> F := Fan( [ [ 0, 1 ], [ 1, 0 ], [ -1, -2 ] ],
>   [ [ 1, 2 ], [ 2, 3 ], [ 1, 3 ] ] );
<A fan in |R^2>
gap> P112 := ToricVariety( F );
<A toric variety of dimension 2>
```

Now `P112` represents $\mathbb{P}(1,1,2)$, and we use the Serre quotient category (cf. Chapter IV) to model the category of coherent sheaves on $\mathbb{P}(1,1,2)$.

```
gap> CohP112 := CategoryOfToricSheaves( P112 );
The Serre quotient category of The category of graded f.p. modules
 over Q[x_1,x_2,x_3] (with weights [ 2, 1, 1 ]) by zero sheaves
```

We will now compute the Cox ring of the variety.

```
gap> S := CoxRing( P112 );
Q[x_1,x_2,x_3]
(weights: [ 2, 1, 1 ])
```

Note that the order of the variables have changed, since now `x_1` has degree 2. As before, we first create the relation matrix for the module $M$, then create a module presentation out of it.

```
gap> M := HomalgMatrix( "[ x_2, x_3 ]", 2, 1, S );
<A 2 x 1 matrix over a graded ring>
gap> M := AsGradedLeftPresentation( M, [ -1 ] );
<An object in The category of graded f.p. modules over Q[x_1,x_2,x_3]
  (with weights [ 2, 1, 1 ])>
```

Now we sheafify the module $M$.

```
gap> SheafM := AsSerreQuotientCategoryObject( CohP112, M );
<An object in The Serre quotient category of The category of
 graded f.p. modules over Q[x_1,x_2,x_3] (with weights [ 2, 1, 1 ])
 by zero sheaves>
```

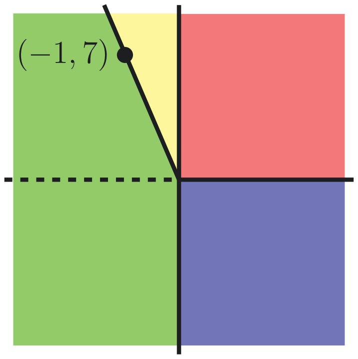
The variable `SheafM` now represents the sheafification of the module $M$. By testing whether the object `SheafM` is 0, we test if the module sheafifies to 0.

```
gap> IsZero( SheafM );
true
```

As already seen, the module sheafifies to 0.

We provide another example where we compute the degree zero part of each localizations $M_{x^{\hat{\sigma}}}$ belonging to a maximal cone $\sigma \in \Sigma$ for a f.p. graded module $M$ over the Cox ring $S$ of a toric variety $X_\Sigma$.

**Example 3.42.** Let $X_\Sigma$ be isomorphic to the Hirzebruch surface $\mathcal{H}_7$. Its fan $\Sigma$ looks like this:



Its Cox ring is $S := \mathbb{C}[x_1, \ldots, x_4]$, with

$$\begin{aligned} \deg(x_1) &= (1,-7),\ \deg(x_3) = (1,0) \\ \deg(x_2) &= \deg(x_4) = (0,1), \end{aligned}$$

and the irrelevant ideal is generated by the monomials

$$x_1x_2,\ x_1x_4,\ x_2x_3,\ \text{and } x_3x_4.$$

We compute the sections corresponding to the affine chart of $X_\Sigma$ given by the maximal cones of $\Sigma$ for the sheafification of the module presented by

$$S(-1,0) \oplus S(-1,-1) \xrightarrow{\begin{pmatrix} x_1x_2^7 & x_3 \\ x_1x_4^8 & 0 \end{pmatrix}} S(0)^2.$$

We first setup the module:

```
gap> S;
Q[x_1,x_2,x_3,x_4]
(weights: [ ( 1, -7 ), ( 0, 1 ), ( 1, 0 ), ( 0, 1 ) ])
gap> M := HomalgMatrix( "[ x_1*x_2^7, x_3, x_1*x_4^8, 0 ]", 2,2, S );
<A 2 x 2 matrix over a graded ring>
gap> M := AsGradedLeftPresentation( M );
<An object in The category of graded f.p. modules over Q[x_1,x_2,x_3,x_4]
(with weights [ [ 1, -7 ], [ 0, 1 ], [ 1, 0 ], [ 0, 1 ] ])>
gap> SMod := GradedLeftPresentations( S );
The category of graded f.p. modules over Q[x_1,x_2,x_3,x_4]
(with weights [ [ 1, -7 ], [ 0, 1 ], [ 1, 0 ], [ 0, 1 ] ])
```

Now we set up the functors mapping a module to the degree zero part of its localization. We denote the functors with $\Gamma_{i,j}$, where $i, j$ are the indices of the variables we are localizing.

```
gap> Gamma12 := DegreeZeroPartOfLocalizationFunctor( SMod, [ 1, 2 ] );
Degree zero functor localized by [ 1, 2 ]
gap> Gamma14 := DegreeZeroPartOfLocalizationFunctor( SMod, [ 1, 4 ] );
Degree zero functor localized by [ 1, 4 ]
```

```
gap> Gamma23 := DegreeZeroPartOfLocalizationFunctor( SMod, [ 2, 3 ] );
Degree zero functor localized by [ 2, 3 ]
gap> Gamma34 := DegreeZeroPartOfLocalizationFunctor( SMod, [ 3, 4 ] );
Degree zero functor localized by [ 3, 4 ]
```

We apply those functors to the module $M$.

```
gap> M12 := ApplyFunctor( Gamma12, M );
<An object in Category of left presentations of Q[t1,t2]>
gap> Display( M12 );
1,   t1,
t2^8,0

An object in Category of left presentations of Q[t1,t2]
```

We see that the ring $S_{x_1x_2}$ is isomorphic to $\mathbb{C}[t_1, t_2]$, i.e., the polynomial ring in two variables, and that $(M_{x_1x_2})$ is presented by the matrix

$$\begin{pmatrix} 1 & t_1 \\ t_2^8 & 0 \end{pmatrix}.$$

We compute the remaining degree zero parts:

```
gap> M14 := ApplyFunctor( Gamma14, M );
<An object in Category of left presentations of Q[t1,t2]>
gap> Display( M14 );
t2^7,t1,
1,   0

An object in Category of left presentations of Q[t1,t2]
gap> M23 := ApplyFunctor( Gamma23, M );
<An object in Category of left presentations of Q[t1,t2]>
gap> Display( M23 );
t1,     1,
t1*t2^8,0

An object in Category of left presentations of Q[t1,t2]
gap> M34 := ApplyFunctor( Gamma34, M );
<An object in Category of left presentations of Q[t1,t2]>
gap> Display( M34 );
t1*t2^7,1,
t1,     0

An object in Category of left presentations of Q[t1,t2]
```

CHAPTER VI

# Application

As an application of the categorical framework for implementing categories in Chapter II, the graded module presentations from Chapter III, and their application when modeling coherent sheaves over toric varieties in Chapter V using the Serre quotient categories described in Chapter IV, we give an algorithm to compute presentations of graded modules and coherent sheaves over toric varieties which are compatible with a special filtration thereof: the so-called *grade or purity filtration.*

We first establish further categorical notions we need throughout this chapter. Then we define the grade or purity filtration of a f.p. graded module and a coherent sheaf. Afterwards, we give the algorithm to compute both the grade filtration and the adjusted presentation of a f.p. graded module and a coherent sheaf. All algorithms will be stated in a categorical manner and can be applied to both f.p. graded modules and coherent sheaves. For the source code, see Appendix G.

## 1. Preliminaries

We will first give the necessary categorical notions to describe the grade filtration of f.p. graded modules and coherent sheaves. We start by defining projective objects.

**Definition 1.1** (Projective object)**.** Let $\mathcal{A}$ be a category.

(1) Let $A, B \in \mathrm{Obj}_{\mathcal{A}}$. An object $P \in \mathrm{Obj}_{\mathcal{A}}$ is called **projective** if for every morphism $\varphi : P \to A$ and every epimorphism $\pi : B \twoheadrightarrow A$ there exists a morphism $\lambda : P \to B$ such that $\lambda\pi \sim \varphi$.
(2) $\mathcal{A}$ is said to have **enough projectives** if for every $A \in \mathrm{Obj}_{\mathcal{A}}$ there is a projective object $P \in \mathrm{Obj}_{\mathcal{A}}$ and an epimorphism $\pi : P \twoheadrightarrow A$.

The dual notion of projective is injective.

**Definition 1.2** (Injective object)**.** Let $\mathcal{A}$ be a category.

(1) Let $A, B \in \mathrm{Obj}_{\mathcal{A}}$. An object $I \in \mathrm{Obj}_{\mathcal{A}}$ is called **injective** if for every morphism $\varphi : A \to I$ and every monomorphism $\iota : A \hookrightarrow B$ there exists a morphism $\lambda : B \to I$ such that $\iota\lambda \sim \varphi$.
(2) $\mathcal{A}$ is said to have **enough injectives** if for every $A \in \mathrm{Obj}_{\mathcal{A}}$ there is an injective object $I \in \mathrm{Obj}_{\mathcal{A}}$ and a monomorphism $\iota : A \hookrightarrow I$.

**Definition 1.3** (Computable projective)**.** Let $\mathcal{A}$ be a computable category with realization $\mathfrak{R}$. Then $\mathcal{A}$ has **computable enough projectives** if:

(1) for every object $A$ there is a projective object $P$ and an epimorphism $\varphi : P \twoheadrightarrow A$ such that the function

$$\text{EpiFromProjective} : \text{Obj}_{\mathcal{A}} \to \text{Mor}_{\mathcal{A}},\ A \mapsto \varphi$$

is computable by $\mathfrak{R}$;

(2) for every triple of objects $P, A, B \in \text{Obj}_{\mathcal{A}}$ where $P$ is projective and there are epimorphisms $\varphi : A \twoheadrightarrow P$ and $\pi : B \twoheadrightarrow A$ with a lift $\lambda : P \to B$ with $\lambda\pi \sim \varphi$ the function

$$\text{Lift} : \text{Mor}_{\mathcal{A}} \times \text{Mor}_{\mathcal{A}} \to \text{Mor}_{\mathcal{A}},\ (\varphi, \pi) \mapsto \lambda$$

is computable by $\mathfrak{R}$.

We show that the category $S$-grpres of graded module presentation (cf. Chapter III), which is equivalent to the category of f.p. graded modules -grmod has enough projectives.

**Definition 1.4** (Free module)**.** Let $S$ be a graded ring with degree group $G$. An $M \in \text{Obj}_{S\text{-grmod}}$ is **free of rank** $n \in \mathbb{N}$ if it is isomorphic to

$$\bigoplus_{i=1}^{n} S(\alpha_i)$$

for some $\alpha_i \in G$.

**Proposition 1.5.** *A free module in the category $S$-*grpres *is projective. Therefore, the category $S$-*grpres *has enough projectives.*

Remark 1.6. In the category $S$-grpres a module presented by a $0 \times n$ matrix for some $n \in \mathbb{N}$ is free of rank $n$. The converse is not true: A module presented by the $1 \times 2$ matrix $(1\ 0)$ is free of rank 1.

In fact, any projective object in $S$-grpres is isomorphic to exactly one object in the set

$$\left\{(0_n, \omega_n) \mid n \in \mathbb{N}, 0_n \in S^{0\times n}, \omega_n \in G^n\right\}.$$

**Theorem 1.7.** *The category $S$-*grpres *has computable enough projectives.*

Proof. We give algorithms for EpiFromProjective and Lift:

(1) For any object $M := (M', \omega) \in \text{Obj}_{S\text{-grpres}}$ set

$$P := (o, \omega),$$

where $o$ is the $0 \times n$ matrix over $S$ and $n :=$ `NumberOfGenerators` $(M)$. Now set

$$\varphi := (M, 1_n, P),$$

where $1_n$ is the $n \times n$ identity matrix. Then we can define

$$\text{EpiFromProjective}(M) := \varphi.$$

(2) Let $A, B, P \in \text{Obj}_{S\text{-grpres}}$, $P$ projective, $\pi := (B, M, A)$ an epimorphism, and $\varphi := (P, N, A)$. We set

$$L := \texttt{RightDivide}(N, M, A'),$$

which means that there is a matrix $X$ such that

$$LM + XA' = N,$$

and define

$$\lambda := \operatorname{Lift}(\varphi, \pi) := (P, L, B).$$

$P$ is projective, so $PL \in S^{0 \times g_B}$, so $B \geqslant_{\text{row}} PL$. Furthermore, from Proposition III.1.9 it follows that $\lambda$ is indeed a morphism.

For the universal property, consider

$$\lambda\pi := (P, LM, A).$$

We have $LM + XA' = N$, so $LM - N = -XA'$, and therefore

$$\lambda\pi \sim \varphi.$$ □

We now define (co)homological chain complexes in a preabelian category, since we need them to define the grade filtration.

**Definition 1.8** ((Co)Complex)**.** Let $\mathcal{A}$ be a preabelian category.

(1) A **homological complex or chain complex or complex** $(C_\bullet, \partial)$ in $\mathcal{A}$ is a series of morphisms

$$\partial_i : C_i \to C_{i-1}$$

for $i \in \mathbb{Z}$ such that

$$\operatorname{ImageObject}(\partial_i) \subseteq \operatorname{KernelObject}(\partial_{i-1})$$

for all $i$. We will often write it as

$$C_\bullet : \cdots \xleftarrow{\partial_0} C_0 \xleftarrow{\partial_1} C_1 \xleftarrow{\partial_2} \cdots.$$

We denote by

$$\begin{aligned} \mathrm{B}_i(C_\bullet) &:= \operatorname{ImageObject}(\partial_{i+1}), \\ \mathrm{Z}_i(C_\bullet) &:= \operatorname{KernelObject}(\partial_i), \\ \mathrm{H}_i(C_\bullet) &:= \mathrm{Z}_i(C_\bullet) / \mathrm{B}_i(C_\bullet). \end{aligned}$$

$\mathrm{H}_i(C_\bullet)$ is called the **$i$-th homology of** $C_\bullet$. The complex $C_\bullet$ is called **exact at homological degree** $i$ if $\mathrm{H}_i(C_\bullet)$ is zero.

(2) A **cohomological complex or cochain complex or cocomplex** $(C^\bullet, \partial)$ in $\mathcal{A}$ is a series of morphisms

$$\partial^i : C^i \to C^{i+1}$$

for $i \in \mathbb{Z}$ such that

$$\operatorname{ImageObject}\left(\partial^i\right) \subseteq \operatorname{KernelObject}\left(\partial^{i+1}\right)$$

for all $i$. We will often write it as

$$C^\bullet : \cdots \xrightarrow{\partial^{-2}} C^{-1} \xrightarrow{\partial^{-1}} C^0 \xrightarrow{\partial^0} \cdots.$$

We denote by

$$\begin{aligned}
\mathrm{B}^i\left(C^\bullet\right) &:= \text{ImageObject}\left(\partial^{i-1}\right),\\
\mathrm{Z}^i\left(C^\bullet\right) &:= \text{KernelObject}\left(\partial^{i}\right),\\
\mathrm{H}^i\left(C^\bullet\right) &:= \mathrm{Z}^i\left(C^\bullet\right)/\,\mathrm{B}^i\left(C^\bullet\right).
\end{aligned}$$

The cocomplex $C^\bullet$ is called **exact at cohomological degree** $i$ if $\mathrm{H}^i\left(C^\bullet\right)$ is zero.

**Definition 1.9** (Chain map)**.** Let $\mathcal{A}$ be a preabelian category and $(C_\bullet, \partial)$ and $(D_\bullet, \epsilon)$ chain complexes. A **chain map** $\varphi : (C_\bullet, \partial) \to (D_\bullet, \epsilon)$ is a collection of morphisms

$$\varphi_i : C_i \to D_i,\ i \in \mathbb{Z}$$

such that

$$\varphi_i \epsilon_i \sim \partial_i \varphi_{i-1} \text{ for all } i \in \mathbb{Z}.$$

The chain complexes in a preabelian $\mathcal{A}$ with chain maps as morphisms form a category.

**Definition 1.10** (Projective resolution)**.** Let $\mathcal{A}$ be an abelian category and $M \in \mathrm{Obj}_{\mathcal{A}}$.

(1) A **homological projective resolution** of $M$ is a chain complex

$$P_\bullet : \cdots \longleftarrow 0 \xleftarrow{\partial_0} P_0 \xleftarrow{\partial_1} P_1 \xleftarrow{\partial_2} \ldots$$

together with an **augmentation map** $\epsilon : P_0 \twoheadrightarrow M$ satisfying the following properties:

(a) $P_i$ is projective for all $i$,
(b) $P_\bullet$ is exact everywhere except at homological degree 0,
(c) $M \cong \text{CokernelObject}\left(\partial_1\right)$ via $\epsilon$,
(d) $\epsilon$ is an epimorphism,
(e) $\partial_1 \epsilon \sim 0_{P_1, M}$.

(2) A **cohomological projective resolution** of $M$ is a cochain complex (cocomplex)

$$P^\bullet : \cdots \longrightarrow P^{-2} \xrightarrow{\partial^{-2}} P^{-1} \xrightarrow{\partial^{-1}} P^0 \xrightarrow{\partial^0} 0 \longrightarrow \ldots$$

together with an **augmentation map** $\epsilon : P^0 \twoheadrightarrow M$ satisfying the following properties:

(a) $P^i$ is projective for all $i$,
(b) $P^\bullet$ is exact everywhere except at cohomological degree 0,
(c) $M \cong \text{CokernelObject}\left(\partial^{-1}\right)$ via $\epsilon$,
(d) $\epsilon$ is an epimorphism,
(e) $\partial_0 \epsilon \sim 0_{P^{-1}, M}$.

We often write $\epsilon : (P^\bullet, \partial) \twoheadrightarrow M$ for the augmentation map.

**Definition 1.11** (Injective Resolution)**.** Let $M$ be an object in an abelian category $\mathcal{A}$. A **homological injective resolution** of $M$ is a cochain complex

$$I^\bullet : \cdots \longrightarrow 0 \longrightarrow I^0 \xrightarrow{\partial^0} I^1 \xrightarrow{\partial^1} \ldots$$

together with an augmentation map $\iota : M \hookrightarrow I^0$ satisfying following properties:

(1) $I^i$ is projective for all $i$,
(2) $I^\bullet$ is exact everywhere except at cohomological degree 0.
(3) $M \cong \text{KernelObject}\left(\partial^0\right)$ via $\iota$,
(4) $\iota$ is a monomorphism,
(5) $\iota\partial^0 \sim 0_{M,I^1}$.

Projective resolutions are computable in computable abelian categories with computable enough projectives.

**Definition 1.12.** Let $\mathcal{A}$ be a computable abelian category with computable enough projectives and $M \in \text{Obj}_{\mathcal{A}}$. Let

$$\alpha := \text{EpiFromProjective}\left(M\right).$$

The chain complex defined by the morphisms

$$\begin{aligned}
\delta_1 &:= \text{PreCompose}\left(\text{EpiFromProjective}\left(\text{KernelObject}\left(\alpha\right)\right),\ \text{KernelEmbedding}\left(\alpha\right)\right),\\
\delta_i &:= \text{PreCompose}\left(\text{EpiFromProjective}\left(\text{KernelObject}\left(\delta_{i-1}\right)\right), \text{KernelEmbedding}\left(\alpha\right)\right),\\
\delta_j &:= \text{UniversalMorphismIntoZeroObject}\left(\text{Range}\left(\delta_{j+1}\right)\right),
\end{aligned}$$

for $i > 1$ and $j < 0$, together with $\alpha$ as augmentation map is a homological projective resolution of $M$. We set the operators

$$\begin{aligned}
\text{ProjectiveResolutionComplex}\left(M\right) &:= \left(P_\bullet, \partial\right),\\
\text{AugmentationMap}\left(M\right) &:= \alpha.
\end{aligned}$$

$$\begin{tikzcd}
 & P_0 \arrow[dl, "\alpha"'] & & P_1 \arrow[ll, "\delta_1"'] \arrow[dl, two heads, "\pi_1"] & & P_2 \arrow[ll, "\delta_2"'] \arrow[dl, two heads, "\pi_2"] \\
M & & K_1 \arrow[ul, hook, "\kappa_0"] & & K_2 \arrow[ul, hook, "\kappa_1"] &
\end{tikzcd}$$

with

$$\begin{aligned}
\kappa_0 &:= \text{KernelEmbedding}\left(\alpha\right),\\
\kappa_1 &:= \text{KernelEmbedding}\left(\delta_1\right),\\
\pi_i &:= \text{EpiFromProjective}\left(K_i\right),\ i = 1, 2.
\end{aligned}$$

Indeed, this is a projective resolution, as directly seen from the construction. By multiplying all indices by $-1$, we can define the operator

$$\text{ProjectiveResolutionCocomplex}.$$

**Corollary 1.13.** *Let $S$ be a $G$-graded ring. Then every object in the category $S$-grpres admits a projective resolution.*

**Definition 1.14** (Ascending/Homological Filtration)**.** Let $\mathcal{A}$ be an abelian category and $A \in \mathrm{Obj}_{\mathcal{A}}$. We call $A$ $(n+1)$**-filtered** if there is a chain of subobjects

$$0 = A_{-n-1} \hookrightarrow A_{-n} \hookrightarrow A_{-n+1} \hookrightarrow \cdots \hookrightarrow A_0 = A.$$

We define

$$F_i A := A_i,$$

and say $A$ is $(n+1)$**-filtered by** $F_\bullet A$. Further, we define

$$\mathrm{gr}_F^i A := A_i / A_{i-1},$$

which we call the ***i*-th graded part** of $A$.

A morphism $\varphi : A \to B$ of two ascendingly filtered objects $A, B$ filtered by $F_\bullet A$ and $F_\bullet B$ is called a **filtered morphism** if for every $i$ we have

$$\varphi \left( F_i A \right) \subseteq F_i B.$$

**Definition 1.15.** Let $(C_\bullet, \partial)$ a homological complex in an abelian category $\mathcal{A}$. $(C_\bullet, \partial)$ is $(n+1)$**-filtered** if each object $C_i$ admits an $(n+1)$-filtration $F_\bullet C_i$ such that each $\partial_i$ is filtered.

**Proposition 1.16.** *Let $(C_\bullet, \partial)$ be a filtered complex with filtration $F_\bullet C_\bullet$. Then there is an induced filtration on the homology $\mathrm{H}_i \left( C_\bullet \right)$, which we will again denote by $F_\bullet \mathrm{H}_i \left( C_\bullet \right)$ and is defined by*

$$F_n \mathrm{H}_i \left( C_\bullet \right) := \mathrm{H}_i \left( F_n C_\bullet \right),$$

*together with the natural isomorphisms for the identification as subobjects.*

Proof. We have

$$\mathrm{B}_j \left( F_i C_\bullet \right) = \mathrm{B}_j \left( C_\bullet \right) \cap F_i C_j$$
$$\mathrm{Z}_j \left( F_i C_\bullet \right) = \mathrm{Z}_j \left( C_\bullet \right) \cap F_i C_j,$$

so the embedding $F_i C_j \hookrightarrow F_{i+1} C_j$ gives rise to a well-defined monomorphism

$$\mathrm{H}_j \left( F_i C_\bullet \right) \hookrightarrow \mathrm{H}_j \left( F_{i+1} C_\bullet \right).$$

Therefore we have an induced filtration on homologies. □

Generalized morphisms (cf. Chapter IV) deliver a tool to relate homology of a complex $C_\bullet$ to the objects in $C_\bullet$.

**Definition 1.17.** Let $\mathcal{A}$ be an abelian category, $A, B, C \in \mathrm{Obj}_{\mathcal{A}}$, and $\iota : B \hookrightarrow A$, $\varphi : C \hookrightarrow B$ monomorphisms. Then the generalized morphism by spans

$$\begin{tikzcd}
B/C \arrow[rr, hook, dashed] & & A \\
& B \arrow[ul, "\mathrm{CokernelProjection}\left(\varphi\right)"] \arrow[ur, hook, "\iota"'] &
\end{tikzcd}$$

is called the **subfactor embedding** of $B/C$ in $A$. Its pseudo-inverse is called the **subfactor projection** of $A$ onto $B/C$.

**Proposition 1.18.** *Let $\mathcal{A}$ be an abelian category, $(C_\bullet, \partial)$, $(D_\bullet, \epsilon)$ chain complexes over $\mathcal{A}$, and $\varphi : C_\bullet \to D_\bullet$ a chain morphism. Let furthermore*

$$\alpha : \mathrm{H}_i(C) \to C_i,$$
$$\beta : D_i \to \mathrm{H}_i(D)$$

*be the $i$-th embedding of the homology of $C_\bullet$ and the $i$-th projection of the homology of $D_\bullet$, respectively. Then the morphism*

$$\mathrm{H}_i(\varphi) : \mathrm{H}_i(C) \to \mathrm{H}_i(D)$$

*induced by the functoriality of $\mathrm{H}$ can be computed as honest representative of composition of generalized morphisms*

$$\alpha\varphi_i\beta.$$

*If $\varphi$ is a quasi-isomorphism, $\mathrm{H}_i(\varphi)$ is an isomorphism.*

Proof. Let

$$\begin{aligned}\kappa_C &:= \mathrm{KernelEmbedding}(\partial_i),\\ K_C &:= \mathrm{KernelObject}(\partial_i),\\ \kappa_D &:= \mathrm{KernelEmbedding}(\epsilon_i),\\ K_D &:= \mathrm{KernelObject}(\epsilon_i).\end{aligned}$$

Then we can compute the functoriality of the kernel by Proposition IV.3.4 via

$$\kappa := \mathrm{HonestRepresentative}(\kappa_C\varphi_i\,\mathrm{GeneralizedInverse}(\kappa_D)).$$

We now have $\kappa : K_C \to K_D$ with $\kappa_C\varphi_i \sim \kappa\kappa_D$. Now, let

$$\begin{aligned}H_C &:= \mathrm{H}_i(C_\bullet),\\ H_D &:= \mathrm{H}_i(D_\bullet),\\ \pi_C &: K_C \to H_C,\\ \pi_D &: K_D \to H_D.\end{aligned}$$

So we get the diagram with commuting squares

$$\begin{tikzcd}
H_C \arrow[d, dotted, "\mathrm{H}_i(\varphi)"'] & K_C \arrow[l, two heads, "\pi_C"'] \arrow[r, hook, "\kappa_C"] \arrow[d, "\kappa"'] & C_i \arrow[r, "\partial_i"] \arrow[d, "\varphi_i"'] & C_{i+1} \arrow[d, "\varphi_{i+1}"'] \\
H_D & K_D \arrow[l, two heads, "\pi_D"'] \arrow[r, hook, "\kappa_D"] & D_i \arrow[r, "\epsilon_i"] & D_{i+1}
\end{tikzcd}$$

We can now fill the first column by

$$\mathrm{H}_i(\varphi) := \mathrm{GeneralizedInverse}(\pi_C)\,\kappa\pi_D,$$

and get

$$\mathrm{H}_i(\varphi) = \mathrm{GeneralizedInverse}(\pi_C)\,\kappa_C\varphi_i\,\mathrm{GeneralizedInverse}(\kappa_D)\,\pi_D,$$

but we have

$$\alpha \sim \mathrm{GeneralizedInverse}(\pi_C)\,\kappa_C,$$
$$\beta \sim \mathrm{GeneralizedInverse}(\kappa_D)\,\pi_D,$$

which concludes the proof. □

## 2. Bicomplexes

We now introduce bicomplexes, which are a special kind of filtered complex. After we define bicomplexes, we will give an algorithm to associate a special bicomplex to the projective resolution of an object $A$. This bicomplex is called Cartan-Eilenberg resolution. It will introduce a filtration on the projective resolution and therefore by Proposition VI.1.16 induce a filtration on the 0-th homology of the projective resolution, which is the object $A$. This filtration will later become the grade filtration.

**Definition 2.1** (Homological bicomplex)**.** Let $\mathcal{A}$ be a category. A **homological bicomplex** $C_{\bullet\bullet}$ over $\mathcal{A}$ is a 2-dimensional grid $C_{i,j}$, $i, j \in \mathbb{Z}$ of objects in $\mathcal{A}$, together with morphisms

$$\partial^h_{i,j} : C_{i,j} \to C_{i-1,j}$$

and

$$\partial^v_{i,j} : C_{i,j} \to C_{i,j-1}$$

such that

$$\partial^v\partial^v = \partial^h\partial^h = \partial^h\partial^v + \partial^v\partial^h = 0.$$

Since $\partial^h\partial^v + \partial^v\partial^h = 0$, a bicomplex is not a complex of complexes.

Throughout the whole of this thesis, bicomplexes $B_{\bullet\bullet}$ are assumed finite, i.e., there is some $N \in \mathbb{N}$ such that for all $n \geqslant N$ it is

$$B_{n,n} = B_{-n,n} = B_{n,-n} = B_{-n,-n} = 0.$$

**Definition 2.2** (Total complex)**.** Let $\mathcal{A}$ be an abelian category and $(C_{\bullet\bullet}, \partial)$ a homological bicomplex. Then the **total complex** $\mathrm{Tot}^{\oplus}(C_{\bullet\bullet})$ is a homological complex defined as follows:

(1) The objects are

$$\mathrm{Tot}^{\oplus}(C_{\bullet\bullet})_n := \bigoplus_{p+q=n} C_{p,q}.$$

(2) The differentials are

$$\partial_n := \left(\bigoplus_{p+q=n} \partial^h_{p,q}\right) + \left(\bigoplus_{p+q=n} \partial^v_{p,q}\right).$$

**Theorem 2.3** ([Wei94, p.141])**.** *Let $\mathcal{A}$ be an abelian category and $C_{\bullet\bullet}$ a homological bicomplex. Then the following are filtrations on* $\mathrm{Tot}^{\oplus}(C_{\bullet\bullet})$*:*

*(1)*

$$^{I}F_i \operatorname{Tot}^{\oplus}(C_{\bullet\bullet})_n := \bigoplus_{\substack{p+q=n \\ p\leqslant i}} C_{p,q};$$

*(2)*

$$^{II}F_i \operatorname{Tot}^{\oplus}(C_{\bullet\bullet})_n := \bigoplus_{\substack{p+q=n \\ q\leqslant i}} C_{p,q}.$$

**Definition 2.4.** Let $\mathcal{A}$ be a category and $(C_{\bullet\bullet}, \partial)$ a homological bicomplex. The **transposed bicomplex** $(C^{\mathrm{tr}}_{\bullet\bullet}, \gamma)$ is a homological bicomplex such that

$$\begin{aligned} C^{\mathrm{tr}}_{pq} &:= C_{qp}, \\ \gamma^h_{pq} &:= \partial^v_{qp}, \\ \gamma^v_{pq} &:= \partial^h_{qp}. \end{aligned}$$

Both $C_{\bullet\bullet}$ and $C^{\mathrm{tr}}_{\bullet\bullet}$ have the same total complex. The first filtration $^{I}F_\bullet$ on the total complex $\operatorname{Tot}^{\oplus}(C_{\bullet\bullet})$ of $C_{\bullet\bullet}$ is the second filtration $^{II}F_\bullet$ on the total complex $\operatorname{Tot}^{\oplus}(C^{\mathrm{tr}}_{\bullet\bullet})$ of $C^{\mathrm{tr}}_{\bullet\bullet}$, and vice versa.

**Theorem 2.5** ([Wei94, Def. 5.7.1 and Lemma 5.7.2]). *Let $\mathcal{A}$ be an abelian category with enough projectives and $P_\bullet$ a complex over $\mathcal{A}$. Then there is a homological bicomplex $(Q_{\bullet\bullet}, \partial)$ together with a **connecting chain map** $\epsilon : Q_{0\bullet} \to P_\bullet$ such that for every $i$ the following holds:*

*(1) If $P_i$ is zero, the column $Q_{i,\bullet}$ is zero;*
*(2) The induced maps on the boundaries and homology*

$$\begin{aligned} \mathrm{B}_i(\epsilon) &: \mathrm{B}_i\left(Q_{i,\bullet}, \partial^h\right) \to \mathrm{B}_i(P), \\ \mathrm{H}_i(\epsilon) &: \mathrm{H}_i\left(Q_{i,\bullet}, \partial^h\right) \to \mathrm{H}_i(P) \end{aligned}$$

*are homological projective resolutions in $\mathcal{A}$.*

*The bicomplex $Q_{\bullet\bullet}$ is called a* **Cartan-Eilenberg *resolution*** *of $P_\bullet$.*

To give the necessary construction for the proof we need the horseshoe lemma.

**Lemma 2.6** (Horseshoe lemma). *Let $\mathcal{A}$ be an abelian category and*

$$0 \longrightarrow A' \xrightarrow{\iota_A} A \xrightarrow{\pi_A} A'' \longrightarrow 0$$

*a short exact sequence. Let furthermore $(P'_\bullet, \partial')$ and $(P''_\bullet, \partial'')$ be projective resolutions of $A'$ and $A''$. Then there is a projective resolution $(P_\bullet, \partial)$ of $A$ such that there is a short exact sequences of complexes*

$$0 \to P'_\bullet \xrightarrow{\alpha} P_\bullet \xrightarrow{\beta} P''_\bullet \to 0$$

*with $P_i \cong P'_i \oplus P''_i$.*

Proof. Since $P'_\bullet$ and $P''_\bullet$ are projective resolutions of $A'$ and $A''$, respectively, there are epimorphisms

$$\epsilon' : P'_0 \twoheadrightarrow A',$$
$$\epsilon'' : P''_0 \twoheadrightarrow A''.$$

Since $\pi_A$ is epi and $P''_0$ is projective, we can define

$$\lambda'' := \mathrm{Lift}\left(\epsilon'', \iota_A\right),$$

so we have $\lambda'' : P''_0 \to A$. Together with

$$\lambda' := \mathrm{PreCompose}\left(\epsilon', \iota_A\right)$$

we can define

$$\epsilon := \mathrm{UniversalMorphismFromDirectSum}\left(\lambda', \lambda''\right).$$

By setting $P_0 := P'_0 \oplus P''_0$ we get an epimorphism

$$\epsilon : P_0 \to A.$$

We set

$$\alpha_0 := \mathrm{InjectionOfCofactorOfDirectSum}\left(\left(P'_0, P''_0\right), 1\right),$$
$$\beta_0 := \mathrm{ProjectionInFactorOfDirectSum}\left(\left(P'_0, P''_0\right), 2\right),$$

and therefore have

$$\alpha_0\epsilon \sim \epsilon'\iota_A,$$
$$\beta_0\epsilon'' \sim \epsilon\pi_A.$$

The sequence

$$0 \to P'_0 \xrightarrow{\alpha_0} P_0 \xrightarrow{\beta_0} P''_0 \to 0$$

is exact, and therefore, with

$$K_0 := \mathrm{KernelObject}\left(\epsilon\right),$$
$$K'_0 := \mathrm{KernelObject}\left(\epsilon'\right),$$
$$K''_0 := \mathrm{KernelObject}\left(\epsilon''\right),$$

we get another exact sequence

$$0 \to K'_0 \to K_0 \to K''_0 \to 0.$$

Now we can iterate the construction above to get $(P_\bullet, \partial)$, $\alpha$, and $\beta$. □

For the implementation, see Appendix G.8.

Proof of Theorem VI.2.5. For each object $P_i$ in $P_\bullet$, take the induced short exact sequence

$$0 \to \mathrm{B}_i\left(P\right) \to \mathrm{Z}_i\left(P\right) \to \mathrm{H}_i\left(P\right) \to 0$$

and compute projective resolutions $P^{\mathrm{B}}_{i,\bullet}$ and $P^{\mathrm{H}}_{i,\bullet}$ of $\mathrm{B}_i(P)$ and $\mathrm{H}_i(P)$. Using the Horseshoe lemma VI.2.6 construct a compatible projective resolution $P^{\mathrm{Z}}_{i,\bullet}$, which leads to a short exact sequence of complexes

$$0 \to P^{\mathrm{B}}_{i,\bullet} \to P^{\mathrm{Z}}_{i,\bullet} \to P^{\mathrm{H}}_{i,\bullet} \to 0.$$

Using the computed resolutions $P^{\mathrm{Z}}_{i,\bullet}$ and $P^{\mathrm{B}}_{i-1,\bullet}$, we can apply the Horseshoe lemma again to the short exact sequence

$$0 \to \mathrm{Z}_i(P) \to P_i \to \mathrm{B}_{i-1}(P) \to 0,$$

and get a compatible resolution $P_{i,\bullet}$ for $P_i$, together with a short exact sequence of complexes

$$0 \to P^{\mathrm{Z}}_{i,\bullet} \to P_{i,\bullet} \to P^{\mathrm{B}}_{i-1,\bullet} \to 0.$$

For the Cartan-Eilenberg resolution $Q_{\bullet\bullet}$ the $P_{ij}$ are the objects and the vertical differentials are the differentials from the $P_{i,\bullet}$, multiplied with $(-1)^i$. The horizontal differentials are constructed by the induced mappings

$$P_{i+1,\bullet} \to P^{\mathrm{B}}_{i,\bullet} \to P^{\mathrm{Z}}_{i,\bullet} \to P_{i,\bullet}.$$

A proof that this construction fulfills the desired properties can be found in [**Wei94**, Lemma 5.7.2]. □

We will denote the Cartan-Eilenberg resolution $Q_{\bullet\bullet}$ of a complex $P_\bullet$ constructed in the above proof by $\mathrm{CE}_{\bullet\bullet}(P_\bullet)$. The construction of the Cartan-Eilenberg resolution given above is for a homological bicomplex. In this case, the Cartan-Eilenberg resolution is a bicomplex in which the only non-zero objects lie in the upper half-plane, i.e., all objects below the $p$-axis are zero. In the cohomological case, the Cartan-Eilenberg resolution lives in the lower half-plane, meaning that all objects above the $p$-axis are zero.

The code for the CAP implementation of the construction of the Cartan-Eilenberg resolution can be found in Appendix G.9.

One can recover the homologies of the initial complex from the Cartan-Eilenberg resolution.

**Theorem 2.7** ([**Wei94**, Thm. 5.7.2]). *Let $P_\bullet$ be a chain complex. Then the total complex $\mathrm{Tot}^{\oplus}(\mathrm{CE}_{\bullet\bullet}(P_\bullet))$ of the Cartan-Eilenberg resolution of $P_\bullet$ is quasi-isomorphic to $P_\bullet$ via the connecting chain map $\epsilon$.*

## 3. Internal Hom and Ext

To compute the bicomplex which will later define the grade filtration of an object, we need the internal Hom and Ext functors. We will define those two functors only for graded modules and for coherent sheaves.

**Definition 3.1** (Additive Functor). Let $\mathcal{A}$ and $\mathcal{B}$ be preadditive categories. A functor $F : \mathcal{A} \to \mathcal{B}$ is called additive if it induces homomorphisms of abelian groups

$$F_{M,N} : \mathrm{Hom}_{\mathcal{A}}(M, N) \to \mathrm{Hom}_{\mathcal{B}}(F(M), F(N)).$$

**Definition 3.2** (Right derived functor)**.** Let $\mathcal{A}$ be an abelian category with enough injectives, $\mathcal{B}$ another category, and $F : \mathcal{A} \to \mathcal{B}$ a left-exact additive functor. Furthermore, let $A \in \mathrm{Obj}_{\mathcal{A}}$ and $I^\bullet$ an injective resolution of $A$. The **$i$-th right derived functor $R^iF$ of $F$** is defined on objects by

$$R^iF(A) := \mathrm{H}^i(F(I^\bullet)).$$

**Definition 3.3** (Right derived contravariant functor)**.** Let $\mathcal{A}$ be an abelian category with enough projectives, $\mathcal{B}$ another abelian category, and $F : \mathcal{A}^{\mathrm{op}} \to \mathcal{B}$ a left-exact additive functor. Furthermore, let $A \in \mathrm{Obj}_{\mathcal{A}}$ and $P_\bullet$ a projective resolution of $A$. The **$i$-th right derived functor $R^iF$ of $F$** is defined on objects by

$$R^iF(A) := \mathrm{H}^i(F(P_\bullet)).$$

Derived functors are independent of the chosen resolution.

**Definition 3.4.** Let $\mathcal{A}$ be an additive category, Ab the category of abelian groups, $A, B, C, D \in \mathrm{Obj}_{\mathcal{A}}$, $\varphi : A \to B$, and $\psi : C \to D$. The functor

$$\mathrm{Hom} : \mathcal{A}^{\mathrm{op}} \times \mathcal{A} \to \mathrm{Ab}$$

defined by

$$\begin{aligned}\mathrm{Hom}_0((A^{\mathrm{op}}, B)) &:= \mathrm{Hom}_{\mathcal{A}}(A, B)\\ \mathrm{Hom}_1((\varphi^{\mathrm{op}}, \psi)) &:= (\mathrm{Hom}_{\mathcal{A}}(B, C) \to \mathrm{Hom}_{\mathcal{A}}(A, D),\ \alpha \mapsto \varphi\alpha\psi)\end{aligned}$$

is called the **Hom-Functor** of $\mathcal{A}$. We write

$$\mathrm{Hom}(A, B) := \mathrm{Hom}_0((A^{\mathrm{op}}, B))$$

and

$$\mathrm{Hom}(\varphi, \psi) := \mathrm{Hom}_1((\varphi^{\mathrm{op}}, \psi)).$$

**Definition 3.5.** Let $S$ be a $G$-graded ring and $M \in S$-grmod. Then the functor

$$\mathrm{Hom}(-, M) : \ S\text{-grmod}^{\mathrm{op}} \to \mathrm{Ab}$$

is left exact and additive, and we call its right derived functor the **Ext-Functor** of $S$-grmod and $M$, and denote it by

$$\mathrm{Ext}^c(-, M) := R^c\,\mathrm{Hom}(-, M).$$

**Definition 3.6.** Let $S$ be a $G$-graded ring. The **internal Hom-Functor**

$$\mathcal{H}\mathrm{om} : S\text{-grmod}^{\mathrm{op}} \times S\text{-grmod} \to S\text{-grmod}$$

in $S$-grmod is defined on objects as in Definition III.1.3, and on morphisms via

$$\mathcal{H}\mathrm{om}_1((\varphi^{\mathrm{op}}, \psi)) := (\mathcal{H}\mathrm{om}(B, C) \to \mathcal{H}\mathrm{om}(A, D),\ \alpha \mapsto \varphi\alpha\psi),$$

with $\varphi : A \to B$ and $\psi : C \to D$ in $\mathrm{Mor}_{S\text{-grmod}}$.

For an object $A \in \mathrm{Obj}_{\mathcal{A}}$ and a morphism $\varphi \in \mathrm{Mor}_{\mathcal{A}}$ we define

$$\mathcal{H}\mathrm{om}(\varphi, A) := \mathcal{H}\mathrm{om}(\varphi, \mathrm{id}_A)$$

and

$$\mathcal{H}\mathrm{om}\,(A,\varphi) := \mathcal{H}\mathrm{om}\,(\mathrm{id}_A,\varphi)\,.$$

**Definition 3.7.** Let $S$ be a $G$-graded ring and $M \in S$-grmod. Then the functor

$$\mathcal{H}\mathrm{om}\,(-,M):\ S\text{-grmod}^{\mathrm{op}} \to S\text{-grmod}$$

is left exact and additive, and we call its right derived functor the **internal Ext-Functor** of $S$-grmod and $M$, and denote it by

$$\mathcal{E}\mathrm{xt}^c\,(-,M) := R^c\,\mathcal{H}\mathrm{om}\,(-,M)\,.$$

**Definition 3.8.** Let $n,m \in \mathbb{N}$, $S$ be a computable $G$-graded ring, $0_n \in S^{0\times n}$, $\omega_n \in G^n$, $S^1 := (0_1,(0)) \in \mathrm{Obj}_{S\text{-grpres}}$, and $P_0 := (0_n,\omega_n) \in \mathrm{Obj}_{S\text{-grpres}}$. We define

$$\mathcal{H}\mathrm{om}\left(P_0,S^1\right) := (0_n,-\omega_n)\,,$$

so this particular internal Hom is computable. Let furthermore $m \in \mathbb{N}$ and $P_1 := (0_m,\omega_m) \in \mathrm{Obj}_{S\text{-grpres}}$, $A \in S^{n\times m}$, and $\varphi := (P_0,A,P_1) \in \mathrm{Mor}_{S\text{-grpres}}$. We define

$$\mathcal{H}\mathrm{om}\left(\varphi,S^1\right) := \left(\mathcal{H}\mathrm{om}\left(P_1,S^1\right),A^{\mathrm{tr}},\mathcal{H}\mathrm{om}\left(P_0,S^1\right)\right)$$

so this particular internal Hom is computable.

Indeed, those internal Homs coincide with the internal Homs defined in Definition VI.3.6.

**Definition 3.9** (Internal Hom on sheaves)**.** Let $X_\Sigma$ be a normal toric variety. Then we define the **internal Hom** for two sheaves $F,G$ in $\mathfrak{qCoh}\,(X_\Sigma)$ as the sheaf

$$\mathcal{H}\mathrm{om}\,(F,G) : U \mapsto \mathcal{H}\mathrm{om}_{\mathcal{O}_{X_\Sigma}(U)}\,(F\,(U)\,,G\,(U))\,,$$

so the internal Hom is again a quasi-coherent sheaf of $\mathcal{O}_{X_\Sigma}$-modules.

**Definition 3.10** (Internal Ext on sheaves)**.** Let $X_\Sigma$ be a normal toric variety, and $M \in \mathrm{Obj}_{\mathfrak{qCoh}(X_\Sigma)}$. Then the covariant internal Hom functor

$$\mathcal{H}\mathrm{om}\,(M,-) : \mathfrak{qCoh}\,(X_\Sigma) \to \mathfrak{qCoh}\,(X_\Sigma)$$

is left-exact and additive, and we call its right derived functor the **internal Ext-Functor** of $\mathfrak{qCoh}\,(X_\Sigma)$ and $M$, and notate it by

$$\mathcal{E}\mathrm{xt}^c\,(M,-) := R^c\,\mathcal{H}\mathrm{om}\,(M,-)\,.$$

## 4. Grade filtration

We now introduce the grade filtration for a graded module $M$ over a $G$-graded ring $S$. We will first define the *grade* of a module, then define the *grade filtration*. Afterwards, we give a (purely categorical) algorithms to compute a presentation for $M$ adjusted to the filtration, and compute it for examples.

**Definition 4.1** (Codimension of a module)**.** Let $S$ be a $G$-graded ring and $0 \neq M \in \mathrm{Obj}_{S\text{-grpres}}$. The **codimension** or **grade** of $M$ is

$$\min\left\{c \geqslant 0 \mid\ \mathcal{E}\mathrm{xt}^c\,(M,S) \neq 0\right\}.$$

**Definition 4.2** (Grade filtration)**.** Let $S$ be a $G$-graded polynomial ring and $M \in \mathrm{Obj}_{S\text{-grpres}}$. The filtration $t_\bullet : t_{-n-1}M \leqslant t_{-n}M \leqslant \cdots \leqslant t_0M = M$ is called **grade filtration** or **purity filtration** if $t_{-i}M$ is the maximal submodule of $M$ of grade greater or equal to $i$.

REMARK 4.3. In this setting the graded parts $t_{-i}M/t_{-i-1}M$ are pure of grade $i$, which means that each nontrivial submodule of $t_{-i}M/t_{-i-1}M$ has grade $i$.

**Definition 4.4** (Dual)**.** Let $S$ be a computable $G$-graded ring.

(1) The **dual** $M^\vee$ of an object $M \in \mathrm{Obj}_{\mathcal{S}\text{-grpres}}$ is $\mathcal{H}\mathrm{om}\,(M, S^1)$.
(2) The **dual** $\varphi^\vee$ of a morphism $\varphi \in \mathrm{Mor}_{\mathcal{S}\text{-grpres}}$ is $\mathcal{H}\mathrm{om}\,(\varphi, S^1)$.

REMARK 4.5. For every object $M \in \mathrm{Obj}_{\mathcal{S}\text{-grpres}}$ there is a canonical morphism

$$M \to (M^\vee)^\vee\,.$$

**Proposition 4.6.** *Let $S$ be a computable $G$-graded ring. Then the functions*

$$\mathrm{DualOnObjects} : \mathrm{Obj}_{S\text{-grpres}} \to \mathrm{Obj}_{S\text{-grpres}},\ M \mapsto M^\vee,$$
$$\mathrm{DualOnMorphisms} : \mathrm{Mor}_{S\text{-grpres}} \to \mathrm{Mor}_{S\text{-grpres}},\ \varphi \mapsto \varphi^\vee,$$
$$\mathrm{MorphismIntoBidual} : \mathrm{Obj}_{S\text{-grpres}} \to \mathrm{Mor}_{S\text{-grpres}},\ M \mapsto (M \mapsto (M^\vee)^\vee)$$

*are computable in $S$-*grpres.

PROOF. Let $U := (0_1, (0))$ with $0_1 \in S^{0\times 1}$. By using EpiFromProjective and KernelEmbedding, for every $M \in \mathrm{Obj}_{S\text{-grpres}}$ we can construct a morphism of free objects $F_0, F_1$

$$\alpha : F_1 \to F_0$$

such that $M \cong \mathrm{CokernelObject}\,(\alpha)$, where the isomorphism can be computed using the cokernel colift. Since $\mathcal{H}\mathrm{om}\,(-, U)$ is contravariant and left exact, therefore, it suffices to compute duals of free objects, since

$$\mathrm{KernelObject}\,(\mathcal{H}\mathrm{om}\,(\alpha, U)) \cong \mathcal{H}\mathrm{om}\,(M, U)\,.$$

This was already done in Proposition VI.3.8, therefore both DualOnObjects and DualOnMorphisms are computable in $S$-grpres.

We now construct the morphism into the bidual. Let now $F := (0_n, \omega_n) \in \mathrm{Obj}_{S\text{-grpres}}$ a free object, with $n \in \mathbb{N}$, $0_n \in S^{0\times n}$, and $\omega \in G^n$. Then we can set

$$(F^\vee)^\vee := F,$$

and

$$\mathrm{MorphismIntoBidual}\,(F) := \mathrm{IdentityMorphism}\,(F)\,.$$

Then, for an arbitrary object $M$ as above,

$$\mathrm{MorphismIntoBidual}\,(M)$$

can be computed by the morphism $\alpha$ above and the functoriality of $\mathcal{H}\mathrm{om}$. ☐

We will also define notions for a dual complex and a dual bicomplex.

**Definition 4.7.** Let $S$ be a computable $G$-graded ring and $(C_\bullet, \partial)$ a chain complex in $S$-grpres. Then the **dual** $((C^\vee)^\bullet, \partial^\vee)$ is a cochain complex such that

$$(C^\vee)^i := (C_i)^\vee ,$$
$$(\partial^\vee)^i := (\partial_{i-1})^\vee .$$

Dualizing preserves the indices of the objects, and therefore shifts the indices of the differentials.

**Definition 4.8.** Let $S$ be a computable $G$-graded ring and $(C_{\bullet\bullet}, \partial)$ a homological bicomplex in $S$-grpres. Then the **dual** $((C^\vee)^{\bullet\bullet}, \partial^\vee)$ is a cohomological bicomplex such that

$$(C^\vee)^{i,j} := (C_{i,j})^\vee ,$$
$$(\partial^\vee)^{i,j} := (\partial_{i-1,j-1})^\vee$$

for both horizontal and vertical differentials.

**Definition 4.9** (Bidualizing resolution)**.** Let $S$ be a $G$-graded ring, and $M \in \mathrm{Obj}_{S\text{-grpres}}$. We define the **bidualizing resolution** of $M$ as the homological bicomplex

$$\mathrm{BidualizingBicomplex}_{\bullet\bullet}(M) := (\mathrm{CE}_{\bullet\bullet}(\mathrm{ProjectiveResolutionComplex}(M)^\vee))^\vee ,$$

i.e., the dual of the Cartan-Eilenberg resolution of the dual of the projective resolution of $M$.

**Theorem 4.10** ([Bar09b, p.32])**.** *Let $S$ be a $G$-graded ring, $M \in \mathrm{Obj}_{S\text{-grpres}}$, and*

$$Q_{\bullet\bullet} := \mathrm{BidualizingBicomplex}_{\bullet\bullet}(M) .$$

*Then the following holds:*

*(1) $\mathrm{Tot}^{\oplus}(Q_{\bullet\bullet})$ is exact everywhere except at homological degree 0.*
*(2) There exists a natural isomorphism*

$$\mathrm{H}_0\left(\mathrm{Tot}^{\oplus}(Q_{\bullet\bullet})\right) \cong M.$$

*(3) The induced filtration of ${}^{II}F\,\mathrm{Tot}^{\oplus}(Q_{\bullet\bullet})$ on $M$ is the grade filtration of $M$.*

**Theorem 4.11.** *Let $S$ be a computable $G$-graded ring and $M \in \mathrm{Obj}_{S\text{-grmod}}$. Then the isomorphism*

$$\mathrm{H}_0\left(\mathrm{Tot}^{\oplus}(\mathrm{BidualizingBicomplex}_{\bullet\bullet}(M))\right) \cong M$$

*is computable.*

Proof. Let $Q_{\bullet\bullet} := \mathrm{BidualizingBicomplex}_{\bullet\bullet}(M)$ and $C_{\bullet\bullet} := \mathrm{CE}_{\bullet\bullet}(M^\vee)$. Furthermore, let $P_\bullet := \mathrm{ProjectiveResolutionComplex}(M)$, and $\alpha := \mathrm{AugmentationMap}(M)$. Then we have

$$M = \mathrm{H}_0(P) .$$

For an object in $S$-grpres the canonical morphism to the bidual is computable and is an isomorphism for free objects, so the isomorphism $\gamma : P_0 \to (P_0^\vee)^\vee$ is computable. Using this isomorphism and a cokernel colift we get an isomorphism

$$\delta : \mathrm{H}_0\left((P^\vee)^\vee\right) \to M.$$

The Cartan-Eilenberg resolution comes with an augmentation $C_{\bullet\bullet} \to P^\vee$, which induces a quasi-isomorphism

$$\varphi : \mathrm{Tot}^{\oplus}\left(C_{\bullet\bullet}\right) \to P^\vee.$$

Taking its dual we get a quasi-isomorphism

$$\psi := \varphi^\vee : (P^\vee)^\vee \to \mathrm{Tot}^{\oplus}\left(Q_{\bullet\bullet}\right),$$

since duals of quasi-isomorphisms between complexes consisting of free objects are quasi-isomorphisms. So we get an isomorphism

$$\chi : \mathrm{H}_0\left(\mathrm{Tot}^{\oplus}\left(Q_{\bullet\bullet}\right)\right) \to \mathrm{H}_0\left((P^\vee)^\vee\right)$$

by applying Proposition VI.1.18. Composing $\chi$ and $\delta$ we get the desired isomorphism. □

An implementation of this isomorphism can be found in Appendices G.16 - G.22.

## 5. Spectral sequences

We now define spectral sequences and use them to compute the graded parts of the grade filtration of a module over a $G$-graded ring $S$. Spectral sequences can be associated to a filtered complex $F_\bullet C_\bullet$ or a bicomplex $C_{\bullet\bullet}$ (as a special case of a filtered complex) and can then be used to compute the graded parts of the induced filtration $F_\bullet H_\bullet(C)$ on the homologies of the filtered complex.

**Definition 5.1** (Homological spectral sequence)**.** Let $\mathcal{A}$ be an abelian category. A **homological spectral sequence** $E_{p,q}$ starting at **page** $a$ consists of the following data:

(1) A family of objects $E^r_{p,q}$ for all $p, q \in \mathbb{Z}$, $r \in \mathbb{Z}_{\geqslant a}$;
(2) Morphisms

$$d^r_{p,q} : E^r_{p,q} \to E^r_{p-r,q+r-1}$$

such that $d^r d^r = 0$.
(3) Isomorphisms

$$\epsilon^r_{p,q} : \mathrm{H}_{p,q}\left(E^r_{\bullet\bullet}\right) \xrightarrow{\sim} E^{r+1}_{p,q}$$

between the homology of $E^r_{\bullet\bullet}$ at $E^r_{p,q}$ and $E^{r+1}_{p,q}$.

Each object $E^r_{p,q}$ is by definition a certain subfactor of the corresponding object on the previous pages $E^s_{p,q}$, $s \leqslant r$.

**Example 5.2.** One can visualize the pages 0, 1, and 2 as follows:

$$\begin{tikzcd}
E^0_{0,2} \arrow[d] \arrow[dr, dotted, no head] & E^0_{1,2} \arrow[d] \arrow[dr, dotted, no head] & E^0_{2,2} \arrow[d] \\
E^0_{0,1} \arrow[d] \arrow[dr, dotted, no head] & E^0_{1,1} \arrow[d] \arrow[dr, dotted, no head] & E^0_{2,1} \arrow[d] \\
E^0_{0,0} & E^0_{1,0} & E^0_{2,0}
\end{tikzcd}
\qquad
\begin{tikzcd}
E^1_{0,2} \arrow[dr, dotted, no head] & E^1_{1,2} \arrow[l] \arrow[dr, dotted, no head] & E^1_{2,2} \arrow[l] \\
E^1_{0,1} \arrow[dr, dotted, no head] & E^1_{1,1} \arrow[l] \arrow[dr, dotted, no head] & E^1_{2,1} \arrow[l] \\
E^1_{0,0} & E^1_{1,0} \arrow[l] & E^1_{2,0} \arrow[l]
\end{tikzcd}
\qquad
\begin{tikzcd}
E^2_{0,2} \arrow[dr, dotted, no head] & E^2_{1,2} \arrow[dr, dotted, no head] & E^2_{2,2} \\
E^2_{0,1} \arrow[dr, dotted, no head] & E^2_{1,1} \arrow[dr, dotted, no head] & E^2_{2,1} \arrow[ull] \\
E^2_{0,0} & E^2_{1,0} & E^2_{2,0} \arrow[ull]
\end{tikzcd}$$

**Definition 5.3** (Boundness and convergence)**.** Let $\mathcal{A}$ be an abelian category. A homological spectral sequence $E_{p,q}$ is **bounded** if for each $n$ there are only finitely many non-zero $E^a_{p,q}$ with $p+q=n$. In this case for each $p,q$ there is an $r_0 \geqslant 0$ such that for all $r \geqslant r_0$ it is

$$E^r_{p,q} \cong E^{r+1}_{p,q}$$

via the isomorphisms $\epsilon^r_{p,q}$, since all in- and outgoing morphisms of $E^r_{p,q}$ start or end at zero objects. We write $E^\infty_{p,q}$ for this stable value.

We say that the spectral sequence **converges** to $H_\bullet$ if there is a family of objects $H_n$ with an ascending filtration

$$0 = F_s H_n \subseteq \cdots \subseteq F_t H_n = H_n$$

and isomorphisms

$$E^\infty_{p,q} \cong F_p H_{p+q} / F_{p-1} H_{p+q}.$$

We write

$$E^a_{p,q} \Rightarrow H_{p+q}.$$

We can use the notion of subfactor embeddings from Definition VI.1.17 to define the spectral sequence of a filtered complex.

**Theorem 5.4.** *Let $\mathcal{A}$ be an abelian category and $C_\bullet$ a filtered complex with filtration $F_\bullet$. The filtered complex $C_\bullet$ determines a spectral sequence $E\left(F_\bullet C_\bullet\right)$ as follows:*

*(1) For the 0-th page, we set*

$$E^0\left(F_\bullet C_\bullet\right)_{p,q} := F_p C_{p+q} / F_{p-1} C_{p+q}.$$

*(2) We recursively define $d^r_{p,q}$ to be a honest representative of the composition of generalized morphisms*

$$E^r\left(F_\bullet C_\bullet\right)_{p,q} \hookrightarrow C_{p+q} \overset{\partial_{p+q}}{\to} C_{p+q-1} \twoheadrightarrow E^r\left(F_\bullet C_\bullet\right)_{p-r,q+r-1}. \tag{†}$$

The proof that all morphisms (†) are honest and this object is indeed a spectral sequence can be found in [**Bar09a**, Section 1.6].

**Corollary 5.5.** *Let $\mathcal{A}$ be a computable abelian category, and $C_\bullet$ a filtered complex. Then the generalized embeddings*

$$E^r\left(F_\bullet C_\bullet\right)_{p,q} \hookrightarrow C_{p+q}$$

*are computable for all* $p, q, r$*.*

The fact that the generalized embeddings $E^r\left(F_\bullet C_\bullet\right)_{p,q} \hookrightarrow C_{p+q}$ are computable follows from the definition of $E\left(F_\bullet C_\bullet\right)$ in Theorem VI.5.4. Beside the obvious recursive algorithm, there is a non-recursive algorithm to compute the morphisms $E^r\left(F_\bullet C_\bullet\right)_{p,q} \hookrightarrow C_{p+q}$ which can be found in [**Pos17**, Subsection II.2.4].

If the filtration $F_\bullet C_\bullet$ of $C_\bullet$ is finite, the spectral sequence $E\left(C_\bullet\right)$ converges.

**Theorem 5.6** ([**Wei94**, p.135])**.** *Let $\mathcal{A}$ be an abelian category and $C_\bullet$ an $(n+1)$-filtered complex with filtration $F_\bullet C_\bullet$. Then the induced spectral sequence converges to the induced filtration of the homology of $C_\bullet$ (cf. Prop. VI.1.16), i.e.*

$$E^r\left(F_\bullet C_\bullet\right)_{p,q} \Rightarrow \mathrm{H}_{p+q}\left(C_\bullet\right).$$

The spectral sequence of a bicomplex can be defined using the total complex and an induced filtration.

**Definition 5.7.** Let $\mathcal{A}$ be an abelian category and $C_{\bullet\bullet}$ a homological bicomplex. Then we define

$$\begin{aligned} {}^{I}E\left(C_{\bullet\bullet}\right) &:= E\left({}^{I}F\,\mathrm{Tot}^{\oplus}\left(C_{\bullet\bullet}\right)\right) \\ {}^{II}E\left(C_{\bullet\bullet}\right) &:= E\left({}^{II}F\,\mathrm{Tot}^{\oplus}\left(C_{\bullet\bullet}\right)\right) = E\left({}^{I}F\,\mathrm{Tot}^{\oplus}\left(C_{\bullet\bullet}^{\mathrm{tr}}\right)\right) \end{aligned}$$

the **first and second spectral sequence of** $C_{\bullet\bullet}$, respectively.

In fact, for all $p, q$ we have

$${}^{I}E^0\left(C_{\bullet\bullet}\right)_{p,q} \cong C_{p,q}$$

and

$${}^{II}E^0\left(C_{\bullet\bullet}\right)_{p,q} \cong C_{p,q}^{\mathrm{tr}}.$$

For a computable abelian category, the algorithm in [**Pos17**, Subsection II.2.4] computes for given $C_{\bullet\bullet}$ and $p, q, r$ the generalized embedding

$${}^{I}E^r\left(C_{\bullet\bullet}\right)_{p,q} \hookrightarrow C_{p,q}.$$

**Definition 5.8** (Bidualizing spectral sequence)**.** Let $S$ be a $G$-graded ring, $M \in \mathrm{Obj}_{S\text{-grpres}}$, and $P := \mathrm{ProjectiveResolutionComplex}\left(M\right)$. The **bidualizing spectral sequence**

$$\mathrm{BidualizingSS}\left(M\right)$$

is the spectral sequence

$${}^{II}E\left(\mathrm{BidualizingBicomplex}_{\bullet\bullet}\left(M\right)\right) := E\left({}^{II}F\,\mathrm{Tot}^{\oplus}\left(\mathcal{H}\mathrm{om}\left(\mathrm{CE}_{\bullet\bullet}\left(\mathcal{H}\mathrm{om}\left(P, S^1\right)\right), S^1\right)\right)\right).$$

This spectral sequence converges to the grade filtration of $M$.

**Theorem 5.9** ( [**Bar09a**, Thm. 9.1.3])**.** *Let $S$ be a $G$-graded ring, $M \in \mathrm{Obj}_{S\text{-grpres}}$, and $E := \mathrm{BidualizingSS}\left(M\right)$ the bidualizing spectral sequence of $M$. Then we have*

$$E_{pq}^2 = \mathcal{E}\mathrm{xt}^{-p}\left(\mathcal{E}\mathrm{xt}^{q}\left(M, S\right), S\right),$$

*and*

$$E^r_{pq} \Rightarrow \begin{cases} M, & p+q=0, \\ 0, & \textit{otherwise,} \end{cases}$$

*where the filtration of $M$ is the grade filtration.*

An implementation for computing the generalized embedding of $E^r_{-p,p}$ into $M$ can be found in Appendix G.21, using algorithm [**Pos17**, Subsection II.2.4].

## 6. Filtered presentation

In the last section we saw how we can compute presentations for the graded parts of the grade filtration of a f.p. graded module $M$, together with their generalized embeddings in $M$. We will now use these generalized embeddings to compute a presentation for $M$ which honors the filtration, the so-called filtered presentation. We first define the filtered presentation of a filtered module.

**Theorem 6.1.** *Let $S$ be a $G$-graded ring and $M \in \mathrm{Obj}_{S\text{-grpres}}$ an $(n+1)$-filtered graded module presentation with filtration $F_\bullet M$ such that each graded part $\mathrm{gr}^i M$ has the presentation*

$$\mathrm{gr}^i M = (M'_i, \omega_i)\,.$$

*Then there exists an object*

$$M_F := (M'_F, (\omega_0, \ldots, \omega_{-n}))$$

*such that $M \cong M_F$ and the relation matrix $M'_F$ is in upper triangular form with matrices $M'_0, \ldots, M'_{-n}$ as diagonal blocks.*

We call such a presentation $M_F$ a **filtered presentation**.

Proof. For the construction of the matrix $M'_F$ and the isomorphism $M \xrightarrow{\sim} M_F$, we use a recursive algorithm. We will assume that the presentation of a submodule $F_{i-1}M$ is already of the desired form, and use the projection

$$F_iM \twoheadrightarrow \mathrm{gr}^i M$$

and the injection

$$F_{i-1}M \hookrightarrow F_iM$$

to construct an isomorphism from a module $M_{F_i}$ with a filtered presentation to $F_iM$.

We set

$$M_i := \mathrm{gr}^i M,$$

and

$$\pi_i : F_iM \to \mathrm{gr}^i M.$$

Furthermore, let

$$\iota_i : F_{i-1}M \to F_iM$$

and suppose $F_{i-1}M$ is already presented by an upper triangular matrix of the desired form. To compute a filtered presentation for $F_iM$, we compute

$$\begin{aligned}
\nu &:= \mathrm{EpiFromProjective}\left(M_i\right),\\
\mu_i &:= \mathrm{KernelEmbedding}\left(\nu\right),\\
\eta_0 &:= \mathrm{Lift}\left(\nu, \pi_i\right),\\
\eta &:= \mathrm{LiftAlongMonomorphism}\left(\mathrm{PreCompose}\left(\mu_i, \eta_0\right), \iota_i\right),
\end{aligned}$$

and get the following commutative diagram with exact rows:

$$\begin{tikzcd}
0 & M_i \arrow[l] \arrow[d, dash] & P \arrow[l, "\nu"'] \arrow[d, "\eta_0"] & K \arrow[l, "\mu_i"'] \arrow[d, "\eta"] & 0 \arrow[l] \\
0 & M_i \arrow[l] & F_iM \arrow[l, "\pi_i"'] & F_{i-1}M \arrow[l, "\iota_i"'] & 0 \arrow[l]
\end{tikzcd}$$

Since $P$ is a free object, and $\nu$ is given by the identity matrix, we can assume that $\mu_i$ is represented by the relation matrix of $M_i$. Now consider the short exact sequence

$$\begin{tikzcd}
0 \arrow[r] & K \arrow[r, "(\mu_i\ \eta)"] & P \oplus F_{i-1}M \arrow[r, "\begin{pmatrix} -\eta_0 \\ \iota_i \end{pmatrix}"] & F_iM \arrow[r] & 0.
\end{tikzcd}$$

The cokernel object of the first map $(\mu_i\ \eta)$ computed by the algorithms for $S$-grpres is presented by

$$\begin{pmatrix} M_i & A \\ 0 & F_{i-1}M' \end{pmatrix},$$

where $A$ represents the morphism $\eta := (K, A, F_{i-1}M)$ and $F_{i-1}M'$ is the relation matrix of $F_{i-1}M$. Since the sequence is exact, we can compute an isomorphism from the object represented by the above matrix to the old presentation of $F_iM$ via CokernelColift. The claim follows. □

To put all steps together and define a filtered presentation for the grade filtration, we need the combined image of a generalized morphism, which we now define.

**Definition 6.2.** Let $\mathcal{A}$ be a abelian category and $\varphi : A \to B$ in $\mathrm{Mor}_{\mathrm{G^S}(\mathcal{A})}$. We define

$$\begin{aligned}
\mathrm{CombinedImage}_{\mathrm{G^S}(\mathcal{A})}\left(\varphi\right) &:= \mathrm{ImageObject}_{\mathcal{A}}\left(\mathrm{Arrow}\left(\varphi\right)\right),\\
\mathrm{CombinedImageEmbedding}_{\mathrm{G^S}(\mathcal{A})}\left(\varphi\right) &:= \mathrm{ImageEmbedding}_{\mathcal{A}}\left(\mathrm{Arrow}\left(\varphi\right)\right)
\end{aligned}$$

the **combined image** and **combined image embedding** of $\varphi$.

If $\mathcal{A}$ is a computable abelian category, both CombinedImage and CombinedImageEmbedding are computable.

**Proposition 6.3.** *Let $\mathcal{A}$ be a computable abelian category, $M \in \mathrm{Obj}_{\mathcal{A}}$, $M'' \leqslant M' \leqslant M$ subobjects, and $\iota : M'/M'' \twoheadrightarrow M$ the subfactor embedding of $M'/M''$ into $M$. Then*

$$\mathrm{CombinedImage}\,(\varphi) = M'.$$

Proof. By definition, we have $\varphi : M'/M'' \twoheadleftarrow M' \hookrightarrow M$, so the claim follows. □

Now, to summarize all previous parts of this chapter, we give an algorithm to compute the filtered presentation for the grade filtration of a graded module $M \in \mathrm{Obj}_{S\text{-grpres}}$ for a computable $G$-graded ring $S$.

**Algorithm 6.4.** Let $M \in \mathrm{Obj}_{S\text{-grpres}}$, and $E := \mathrm{BidualizingSS}\,(M)$. Let furthermore $Q_{\bullet\bullet} := \mathrm{BidualizingBicomplex}_{\bullet\bullet}\,(M)^{\mathrm{tr}}$. Now a filtered presentation of $M$ adjusted to the grade filtration $t_{\bullet}M$ can be computed as follows:

(1) Compute the boundaries of $Q_{\bullet\bullet}$, i.e., find $r \in \mathbb{N}$ such that $Q_{i,j} = 0$ for all $i \leqslant -r$, $j \geqslant r$. This $r$ works as bound for computing the total complex. We also have $E^{\infty} = E^{r}$.
(2) Compute $\mathrm{Tot}^{\oplus}\,(Q_{\bullet\bullet})$, using $r$ as bounds for the direct sums.
(3) Compute the isomorphism $\varphi : M \to \mathrm{H}_0\left(\mathrm{Tot}^{\oplus}\,(Q_{\bullet\bullet})\right)$ as in Theorem VI.4.11.
(4) For $q = 0, \dots, r$ compute the generalized embedding
$$\iota_q : E^{r}_{-q,q} \hookrightarrow \mathrm{Tot}^{\oplus}{}_0\,(Q_{\bullet\bullet})$$
via the algorithm in [**Pos17**, Algorithm II.4.2].
(5) If $E^{r}_{-q,q}$ is not 0 compute the generalized embedding
$$\alpha_q : E^{r}_{-q,q} \hookrightarrow M,$$
as composition of $\iota_q$, the cokernel projection
$$\pi : \mathrm{Tot}^{\oplus}{}_0\,(Q_{\bullet\bullet}) \to \mathrm{H}_0\,\mathrm{Tot}^{\oplus}\,(Q_{\bullet\bullet})$$
and the isomorphism $\varphi$. We get a morphism
$$E^{r}_{-q,q} \overset{\alpha}{\hookrightarrow} \mathrm{Tot}^{\oplus}{}_0\,(Q_{\bullet\bullet}) \xrightarrow{\pi} \mathrm{H}_0\,\mathrm{Tot}^{\oplus}\,(Q_{\bullet\bullet}) \xrightarrow{\varphi} M.$$
(6) Compute
$$\beta_q := \mathrm{CombinedImageEmbedding}\,(\alpha_q)\,.$$
We now have $\beta_q : t_{-q}M \hookrightarrow M$.
(7) Compute
$$\gamma'_q := \mathrm{PreCompose}\,(\beta_q, \mathrm{PseudoInverse}\,(\alpha_q))\,.$$
and set
$$\gamma_q := \mathrm{HonestRepresentative}\left(\gamma'_q\right).$$
This map is now the epimorphism
$$\gamma_q : t_{-q}M \to t_{-q}M/t_{-q-1}M.$$
(8) Use the algorithm for the filtered presentation VI.6.1 to construct the filtered presentation of $M$ for the grade filtration.

An implementation for the entire algorithm can be found in Appendix G.22.

**Example 6.5.** Using the Algorithm VI.6.4, we can compute the filtered presentation for the grade filtration of a module $M$, presented by the matrix

$$\begin{pmatrix} -x^2z+xyz+xz^2 & y^2z & -xz+yz & x-y & \cdot & \cdot \\ -x^3+x^2y+x^2z & xy^2 & -x^2+xy & \cdot & x-y & -xy \\ \cdot & \cdot & \cdot & xy & -yz & \cdot \\ \cdot & \cdot & \cdot & x^2 & -xz & \cdot \\ \cdot & \cdot & \cdot & xz & -z^2 & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & z \end{pmatrix} \in \mathbb{Q}[x,y,z].$$

We first set up the ring, the module, and the necessary categories for the computation.

```
gap> LoadPackage( "ModulePresentationsForCAP" );
gap> LoadPackage( "HomologicalAlgebraForCAP" );
gap> SwitchGeneralizedMorphismStandard( "span" );
gap>
gap> Q := HomalgFieldOfRationalsInSingular();
Q
gap> S := GradedRing( Q * "x,y,z" );;
gap> WeightsOfIndeterminates( S );
[ 1, 1, 1 ]
gap>
gap> mat := HomalgMatrix( "[ \
> -x^2*z+x*y*z+x*z^2,y^2*z,-x*z+y*z,x-y,0,   0,   \
> -x^3+x^2*y+x^2*z,  x*y^2,-x^2+x*y,0,  x-y, -x*y,\
> 0,                 0,    0,       x*y,-y*z,0,   \
> 0,                 0,    0,       x^2,-x*z,0,   \
> 0,                 0,    0,       x*z,-z^2,0,   \
> 0,                 0,    0,       0,  0,   z    \
> ]", 6, 6, S );
<A 6 x 6 matrix over a graded ring>
gap> S0 := GradedFreeLeftPresentation( 1, S );
<An object in The category of graded f.p. modules over Q[x,y, z]
  (with weights [ 1, 1, 1 ])>
gap> SetIsAdditiveCategory( CocomplexCategory( CapCategory(S0) ), true );
gap> SetIsAdditiveCategory( ComplexCategory( CapCategory( S0 ) ), true );
gap> M := AsGradedLeftPresentation( mat, [ 0, 0, 1, 2, 2, 1 ] );
<An object in The category of graded f.p. modules over Q[x,y,z]
  (with weights [ 1, 1, 1 ])>
```

We now compute a free resolution `res` of the module $M$, and compute its dual `homres`.

```
gap> res1 := FreeResolutionComplex( M );
[ <An object in Complex category of The category of graded f.p.
  modules over Q[x,y,z] (with weights [ 1, 1, 1 ])>,
  <A morphism in The category of graded f.p. modules over Q[x,y,z]
  (with weights [ 1, 1, 1 ])> ]
gap> res := res1[ 1 ];
<An object in Complex category of The category of graded f.p.
  modules over Q[x,y,z] (with weights [ 1, 1, 1 ])>
gap> homres := DualOnComplex( res );
<An object in Cocomplex category of The category of graded f.p.
  modules over Q[x,y,z] (with weights [ 1, 1, 1 ])>
```

We now compute the Cartan-Eilenberg resolution `CE` of `homres`, and then compute the dual of `CE`, `homCE`.

```
gap> CE := CartanEilenbergResolution( homres, FreeResolutionCocomplex );
<An object in Cocomplex category of Cocomplex category of The
  category of graded f.p. modules over Q[x,y,z]
  (with weights [ 1, 1, 1 ])>
gap> homCE := DualOnCocomplexCocomplex( CE );
<An object in Complex category of Complex category of The category
  of graded f.p. modules over Q[x,y,z] (with weights [ 1, 1, 1 ])>
```

Since we want to work with the second filtration of `homCE`, we compute the transposed of `homCE`, which we denote by `trhomCE`.

```
gap> trhomCE := TransposeComplexOfComplex( homCE );
<An object in Complex category of Complex category of The category
  of graded f.p. modules over Q[x,y,z] (with weights [ 1, 1, 1 ])>
```

We can now compute the filtered presentation of the module $M$.

```
gap> filtration := PurityFiltrationBySpectralSequence( trhomCE, 4,
> homCE, homres, res1[ 2 ] );
<A morphism in The category of graded f.p. modules over Q[x,y,z]
  (with weights [ 1, 1, 1 ])>
```

We can now check that the computed morphism from the filtered presentation of $M$ to the original one is indeed an isomorphism, and look at both the filtered presentation and the matrix presenting the isomorphism.

```
gap> IsIsomorphism( filtration );
true
gap> Display( Source( filtration ) );
x, -z, 0,    0,     0,       0,  1,
-y,z,  y^2*z,-y*z^2,-x*z+y*z,0,  -1,
0, x-y,x*y^2,-x*y*z,-x^2+x*y,x*y,0,
0, 0,  0,    0,     0,       z,  0,
0, 0,  0,    0,     0,       0,  z,
0, 0,  0,    0,     0,       0,  y,
0, 0,  0,    0,     0,       0,  x
(over a graded ring)
gap> Display( filtration );
0,   0, 0, -1,0, 0,
0,   0, 0, 0, -1,0,
0,   -1,0, 0, 0, 0,
1,   0, 0, 0, 0, 0,
-x+z,0, -1,0, 0, 0,
0,   0, 0, 0, 0, 1,
0,   0, 0, x, -z,0
(over a graded ring)
```

We get the following blocks in the resulting matrix:

$$\left(\begin{array}{ccccc|c|c}
x & -z & \cdot & \cdot & \cdot & \cdot & 1 \\
-y & z & y^2z & -yz^2 & -xz+yz & \cdot & -1 \\
\cdot & x-y & xy^2 & -xyz & -x^2+xy & xy & \cdot \\
\hline
\cdot & \cdot & \cdot & \cdot & \cdot & z & \cdot \\
\hline
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & z \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & y \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & x
\end{array}\right)$$

The isomorphism from the filtered presentation of the module $M$ to its original representation is given by the matrix

$$\begin{pmatrix}
\cdot & \cdot & \cdot & -1 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & -1 & \cdot \\
\cdot & -1 & \cdot & \cdot & \cdot & \cdot \\
1 & \cdot & \cdot & \cdot & \cdot & \cdot \\
-x+z & \cdot & -1 & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & 1 \\
\cdot & \cdot & \cdot & x & -z & \cdot
\end{pmatrix}.$$

In Figure VI.1 we visualize the relevant part of the zeroth to fourth page of the spectral sequence with binary matrices. The columns range from $-3$ to $0$ from left to right and rows range from $0$ to $3$ from the bottom to the top. This means, e.g., $E_{-3,2}$ is displayed at position $(1,2)$ of the matrices. If an entry is `.`, it means that the object is already zero. A `*` means that the object at that position is not zero. We see that on the third page the entry $E^3_{-3,2}$ is not zero, and therefore the entry $E^3_{0,0}$ is not the limit $E^\infty_{0,0}$. So the computation of the grade filtration on page 3 would not have been possible, but we indeed had to compute it using the entries on page 4.

## 7. Coherent sheaves

We want to apply Algorithm VI.6.4 to construct a presentation induced by the grade filtration of coherent sheaves over the Cox ring $S$ of a toric variety $X_\Sigma$ realized as an object

```
gap> DisplaySpectralSequencePage( trhomCE, 0, [ -3 .. 0 ], [ 0 .. 3 ] );
 * * * *
 * * * *
 * * * *
 . * * *
gap> DisplaySpectralSequencePage( trhomCE, 1, [ -3 .. 0 ], [ 0 .. 3 ] );
 * * * *
 * * * *
 * * * *
 . . * *
gap> DisplaySpectralSequencePage( trhomCE, 2, [ -3 .. 0 ], [ 0 .. 3 ] );
 * . . .
 * . . .
 * * * .
 . . * *
gap> DisplaySpectralSequencePage( trhomCE, 3, [ -3 .. 0 ], [ 0 .. 3 ] );
 * . . .
 * . . .
 . . * .
 . . . *
gap> DisplaySpectralSequencePage( trhomCE, 4, [ -3 .. 0 ], [ 0 .. 3 ] );
 * . . .
 . . . .
 . . * .
 . . . *
```

FIGURE VI.1. Spectral sequence of the graded module $M$, see VI.6.5.

in the Serre quotient of the f.p. graded module category over $S$. From now on, we will always work over a toric variety $X_\Sigma$ with no torus factors and Cox ring $S$. We identify the Serre quotient $(\mathcal{A} := S\text{-grpres}) / \left(\mathcal{C} := S\text{-grpres}^0\right)$ with $\mathfrak{Coh}\,(X_\Sigma)$ and denote by

$$\mathrm{Sh} : S\text{-grpres} \to \mathfrak{Coh}\,(X_\Sigma)$$

the sheafification functor. We take $\overline{\mathrm{G}}_{\mathcal{C}}\,(\mathcal{A})$ to be $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}\,(\mathcal{A})$.

**Notation.** By identifying $\mathfrak{Coh}\,(X_\Sigma)$ with the Serre quotient $S$-grpres/$S$-grpres$^0$ which is modeled by $\overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}\,(\mathcal{A})$, a morphism in $\mathfrak{Coh}\,(X_\Sigma)$ has an arrow and a reversed arrow. Furthermore, if we talk about an honest morphism, it is the image of a morphism in $S$-grpres under the functor Sh. The honest representative of a morphism $\varphi \in \mathrm{Mor}_{\mathfrak{Coh}(X_\Sigma)}$ will be a morphism in $S$-grpres.

We first give a proper alternative for projective objects and projective resolutions, since $\mathfrak{Coh}\left(X_\Sigma\right)$ does not have enough projectives in general.

**Definition 7.1** (Locally free objects)**.** An object $F \in \mathrm{Obj}_{\mathfrak{Coh}(X_\Sigma)}$ is called **locally free** if there exists some index set $I$ and for every $x \in X_\Sigma$ there is an open subset $x \in U \subseteq X_\Sigma$ such that

$$F|_U \cong \bigoplus_{i \in I} \mathcal{O}_X|_U.$$

**Corollary 7.2.** *The sheafification of a free module is locally free.*

Proof. The sheafification respects the direct sum, and the localization is blind to twists. The claim follows. □

**Proposition 7.3** (Homological locally free resolution)**.** *Let $F \in \mathrm{Obj}_{\mathfrak{Coh}(X_\Sigma)}$ with $F = \mathrm{Sh}\left(M\right)$. Then the homological locally free resolution of $F$ can be computed as*

$$\mathrm{Sh}\left(\mathrm{ProjectiveResolutionComplex}_{S\text{-grpres}}\left(M\right)\right).$$

*The dual is true for the cohomological locally free resolution.*

Since all of the morphisms in such a homological locally free resolution are represented by honest generalized morphisms, i.e., come from morphisms in $S$-grpres, we can also compute a Cartan-Eilenberg resolution of this locally free resolution. We are going to define a locally free lift along honest morphisms, which replaces the projective lift used in the horseshoe lemma.

**Definition 7.4.** Let $\alpha' : A \to C$ and $\beta' : B \to C \in \mathrm{Mor}_{\mathfrak{Coh}(X_\Sigma)}$ be honest and $\alpha, \beta \in \mathrm{Mor}_{S\text{-grpres}}$ their honest representatives, such that $\mathrm{Lift}\left(\alpha, \beta\right)$ exists. Then we define

$$\mathrm{Lift}_{\mathfrak{Coh}(X_\Sigma)}\left(\alpha', \beta'\right) := \mathrm{Sh}\left(\mathrm{Lift}_{S\text{-grpres}}\left(\alpha, \beta\right)\right).$$

**Proposition 7.5.** *Using the lift of sheaf morphisms defined in Definition VI.7.4 we are able to compute the Cartan-Eilenberg resolution of a complex of sheaves, using locally free resolutions.*

Proof. The computed resolutions in the horseshoe lemma only contain honest morphisms: All morphisms in the locally free resolution are honest, the injection and projections of and to factors of direct sums are honest, compositions of honest morphisms are honest, and the lifts are honest. So all involved morphisms are honest. □

Remark 7.6 (Existence of lifts). When using locally free objects due to the lack of projectives in the proof of Theorem VI.6.1, the needed substitution of the projective lift might not exist: if a surjective morphism $\varphi : A \twoheadrightarrow B$ in $\mathrm{Mor}_{\mathfrak{Coh}(X_\Sigma)}$ is represented by the honest span of $\varphi' \in \mathrm{Mor}_{S\text{-grpres}}$, then

$$\mathrm{CokernelObject}_{S\text{-grpres}}\left(\mathrm{Arrow}\left(\varphi\right)\right) \in S\text{-grpres}^0.$$

But the cokernel object of $\mathrm{Arrow}\left(\varphi\right)$ itself is not zero in $\mathrm{Obj}_{S\text{-grpres}}$. Therefore the lift method described in Definition VI.7.4 does not apply, since $\mathrm{Arrow}\left(\varphi\right)$ is not an epimorphism in $S$-grpres.

**Example 7.7.** Let $S := \mathbb{C}[x,y]$ be the polynomial ring in two indeterminates graded with the standard grading and

$$\varphi : S(-1)^{2\times 1} \xrightarrow{(x,y)} S^{1\times 1}.$$

The sheafification of this morphism is surjective, since the cokernel is finite dimensional as $\mathbb{C}$-vector space and therefore sheafifies to 0. If we compute the epimorphism from a projective to $S^{2\times 1}$, $\pi : S^{2\times 1} \xrightarrow{\mathrm{id}} S^{2\times 1}$, the lift does not exist, since $\varphi$ is not surjective as a morphism of graded $S$-modules.

We can solve this by using compatible lifts, i.e., computing epimorphisms from locally free objects such that the lift exists.

**Definition 7.8** (Compatible lift)**.** Let $\varphi : A \to B$ be an honest epimorphism in $\mathfrak{Coh}(X_\Sigma) \cong \overline{\mathrm{G}}^{\mathrm{S}}_{\mathcal{C}}(\mathcal{A})$. Then the **compatible locally free lift** is a tuple

$$(\pi : P \twoheadrightarrow B, \psi : P \to A),$$

such that

(1) $P$ is locally free,
(2) $\pi$ is epi,
(3) and $\pi \sim \psi\varphi$.[1]

**Theorem 7.9.** *The compatible lift of an honest epimorphism $\varphi : A \twoheadrightarrow B$ always exists.*

Proof. We are going to construct the necessary morphisms. Let

$$\eta := \mathrm{ImageEmbedding}_{S\text{-grpres}}(\mathrm{HonestRepresentative}(\varphi)).$$

Since $\varphi$ is an honest epimorphism, $\mathrm{Sh}(\eta)$ is an isomorphism (but $\eta$ is not necessarily). Let $\pi' : P \twoheadrightarrow I$ be an epimorphism from a locally free object to $\mathrm{Source}(\mathrm{Sh}(\eta))$ and set

$$\pi := \mathrm{PreCompose}_{\mathfrak{Coh}(X_\Sigma)}(\pi', \mathrm{Sh}(\eta)).$$

This morphism is now serving as epi from a locally free object. For the lift, we compute

$$\varphi' := \mathrm{PreCompose}_{\mathfrak{Coh}(X_\Sigma)}\left(\varphi, \mathrm{Inverse}_{\mathfrak{Coh}(X_\Sigma)}(\mathrm{Sh}(\eta))\right).$$

Since $\mathrm{HonestRepresentative}(\varphi')$ is surjective, we can now use the lift from Definition VI.7.4 and set

$$\iota := \mathrm{Lift}_{\mathfrak{Coh}(X_\Sigma)}(\pi', \varphi').$$

The pair $(\pi, \iota)$ is the desired lift tuple. The correctness follows from the construction. □

We also need the notion of a dual object and morphism in $\mathfrak{Coh}(X_\Sigma)$.

**Definition 7.10.** Let $F \in \mathrm{Obj}_{\mathfrak{Coh}(X_\Sigma)}$ with $F = \mathrm{Sh}(M)$ and $\varphi : A \to B \in \mathrm{Mor}_{\mathfrak{Coh}(X_\Sigma)}$. We set

$$\mathrm{DualOnObjects}_{\mathfrak{Coh}(X_\Sigma)}(F) := \mathrm{Sh}\left(\mathrm{DualOnObjects}_{S\text{-grpres}}(M)\right),$$

[1] Recall, $\sim$ is the equivalence relation on the Hom-sets in the category model II.2.2.

and, with $\varphi_1 := \mathrm{ReversedArrow}\left(\varphi\right)$, $\varphi_2 := \mathrm{Arrow}\left(\varphi\right)$, and

$$\psi_i := \mathrm{ProjectionInFactorOfFiberProduct}_{S\text{-grpres}}\left(\left(\varphi_1, \varphi_2\right), i\right),$$

we set

$$\mathrm{Arrow}\left(\mathrm{DualOnMorphisms}_{\mathfrak{Coh}(X_\Sigma)}\left(\varphi\right)\right) := \psi_1,$$
$$\mathrm{ReversedArrow}\left(\mathrm{DualOnMorphisms}_{\mathfrak{Coh}(X_\Sigma)}\left(\varphi\right)\right) := \psi_2.$$

For the morphism to the bidual, we take the sheafification of the morphism of graded module, i.e.,

$$\mathrm{MorphismIntoBidual}_{\mathfrak{Coh}(X_\Sigma)}\left(F\right) := \mathrm{Sh}\left(\mathrm{MorphismIntoBidual}_{S\text{-grpres}}\left(M\right)\right).$$

Now, we can use the compatible lift in Theorem VI.6.1 to compute a filtered presentation. We cannot expect a nice small resulting matrix as for modules.

**Example 7.11.** We take the same module $M$ as in Example VI.6.5, but do the complete computation with its sheafification $\widetilde{M}$. At the end we look at the sizes of the resulting matrices. We cannot expect the same matrices as in Example VI.6.5 since the cokernel of the augmented maps $(\mu_i\ \eta)$ in the construction of a filtered presentation in the proof of Theorem VI.6.1 is not computed by stacked matrices of the map and the range, but according to the construction given for the cokernel of a Serre quotient morphism in the proof of Theorem IV.5.10.

```
gap> LoadPackage( "ModulePresentationsForCAP" );
true
gap> LoadPackage( "HomologicalAlgebraForCAP" );
true
gap> LoadPackage( "ToricSheaves" );
true
gap> SetRecursionTrapInterval( 1000000 );
gap> SwitchGeneralizedMorphismStandard( "span" );
gap> Q := HomalgFieldOfRationalsInSingular();
Q
gap> S := GradedRing( Q * "x,y,z" );;
gap>
gap> WeightsOfIndeterminates( S );
[ 1, 1, 1 ]
gap>
gap> mat := HomalgMatrix( "[ \
> -x^2*z+x*y*z+x*z^2,y^2*z,-x*z+y*z,x-y,0,   0,   \
> -x^3+x^2*y+x^2*z,  x*y^2,-x^2+x*y,0,  x-y, -x*y,\
> 0,                 0,    0,       x*y,-y*z,0,   \
> 0,                 0,    0,       x^2,-x*z,0,   \
> 0,                 0,    0,       x*z,-z^2,0,   \
> 0,                 0,    0,       0,  0,   z    \
> ]", 6, 6, S );
<A 6 x 6 matrix over a graded ring>
gap> is_artinian_left := function( module )
>   local mat;
>
>     mat := UnderlyingMatrix( module );
>
>     return IsZero( HilbertPolynomial(
>                     UnderlyingMatrixOverNonGradedRing( mat ) ) );
>
> end;
function( module ) ... end
gap> Coh := GradedLeftPresentations( S ) / is_artinian_left;
The Serre quotient category of The category of graded f.p. modules
 over Q[x,y,z] (with weights [ 1, 1, 1 ]) by test funct
 ion with name: is_artinian_left
gap>
gap> SetIsAdditiveCategory( CocomplexCategory( Coh ), true );
gap> SetIsAdditiveCategory( ComplexCategory( Coh ), true );
gap> M := AsGradedLeftPresentation( mat, [ 0, 0, 1, 2, 2, 1 ] );
<An object in The category of graded f.p. modules over Q[x,y,z]
 (with weights [ 1, 1, 1 ])>
gap> ShM := AsSerreQuotientCategoryObject( Coh, M );
<An object in The Serre quotient category of The category of
 graded f.p. modules over Q[x,y,z] (with weights [ 1, 1,1 ])
 by test function with name: is_artinian_left>
gap> res1 := FreeResolutionComplex( ShM );
[ <An object in Complex category of The Serre quotient category
 of The category of graded f.p. modules over Q[x,y,z]
 (with weights [ 1, 1, 1 ]) by test function with name:
 is_artinian_left>,
  <A morphism in The Serre quotient category of The category
 of graded f.p. modules over Q[x,y,z] (with weights [ 1, 1, 1 ])
 by test function with name: is_artinian_left> ]
gap> res := res1[ 1 ];
<An object in Complex category of The Serre quotient category
  of The category of graded f.p. modules over Q[x,y,z]
  (with weights [ 1, 1, 1 ]) by test function
  with name: is_artinian_left>
gap> homres := DualOnComplex( res );
<An object in Cocomplex category of The Serre quotient catego
 ry of The category of graded f.p. modules over Q[x,y,z]
 (with weights [ 1, 1, 1 ]) by test function with name:
 is_artinian_left>
gap> CE := CartanEilenbergResolution( homres, FreeResolutionCocomplex );
<An object in Cocomplex category of Cocomplex category of The
  Serre quotient category of The category of graded f.p. modules
  over Q[x,y,z] (with weights [ 1, 1, 1 ]) by test function
  with name: is_artinian_left>
gap> homCE := DualOnCocomplexCocomplex( CE );
<An object in Complex category of Complex category of The Serre
 quotient category of The category of graded f.p. modules over Q[x,y,z]
 (with weights [ 1, 1, 1 ]) by test function with name:
 is_artinian_left>
gap> trhomCE := TransposeComplexOfComplex( homCE );
<An object in Complex category of Complex category of The Serre
 quotient category of The category of graded f.p. modules over Q[x,y,z]
 (with weights [ 1, 1, 1 ]) by test function with name:
 is_artinian_left>
gap> homhomres := DualOnCocomplex( homres );
<An object in Complex category of The Serre quotient category
 of The category of graded f.p. modules over Q[x,y,z]
 (with weights [ 1, 1, 1 ]) by test function with name:
 is_artinian_left>
gap> filtration := PurityFiltrationBySpectralSequence( trhomCE, 3, homCE,
> homres, res1[ 2 ] );
<A morphism in The Serre quotient category of The category of
  graded f.p. modules over Q[x,y,z] (with weights [ 1, 1, 1 ])
  by test function with name: is_artinian_left>
gap> IsIsomorphism( filtration );
true
```

Since we have computed an isomorphism to a new presentation, we look at the sizes of the resulting matrices.

```
 gap> UnderlyingMatrix( UnderlyingHonestObject( Source( filtration ) ) );
<An unevaluated non-zero 92 x 83 matrix over a graded ring>
gap> UnderlyingMatrix( Arrow(
>     UnderlyingGeneralizedMorphism( filtration ) ) );
<An unevaluated 137 x 6 matrix over a graded ring>
gap> UnderlyingMatrix( ReversedArrow(
>     UnderlyingGeneralizedMorphism( filtration ) ) );
<An unevaluated 137 x 83 matrix over a graded ring>
```

In Figure VI.2 we visualize the relevant part of the zeroth to third page of the spectral sequence with a binary matrix. As in Figure VI.1 if an entry is ., it means that the object

```
gap> DisplaySpectralSequencePage( trhomCE, 0, [ -3 .. 0 ], [ 0 .. 3 ] );
 * * * *
 * * * *
 * * * *
 . * * *
gap> DisplaySpectralSequencePage( trhomCE, 1, [ -3 .. 0 ], [ 0 .. 3 ] );
 * * * *
 * * * *
 * * * *
 . . * *
gap> DisplaySpectralSequencePage( trhomCE, 2, [ -3 .. 0 ], [ 0 .. 3 ] );
 . . . .
 . . . .
 . * * .
 . . . *
gap> DisplaySpectralSequencePage( trhomCE, 3, [ -3 .. 0 ], [ 0 .. 3 ] );
 . . . .
 . . . .
 . . * .
 . . . *
```

FIGURE VI.2. Spectral sequence of the coherent sheaf $\widetilde{M}$

is already zero. A $*$ means that the object at that position is not zero. We see that on the third page of the spectral sequence all diagonal entries are stable, and therefore we do not have to compute the fourth page of the spectral sequence as in the corresponding example VI.6.5 for modules, but could terminate the computation of the grade filtration at page 3. This different in the page numbers where all objects are stable between modules and their sheafification rises with the dimension of the irrelevant ideal $B$ of the Cox ring $S$ of the toric variety $X_\Sigma$. Therefore computing the spectral sequence with sheaves instead of computing with modules and sheafifying the result leads to shorter computations.

CHAPTER VII

# Implementation of computable categories

In this chapter we will describe the philosophy, design, and features of the categorical programming language Cap via its implementation in the computer algebra system `GAP`, alongside with the motivation behind the design decisions and the most important features.

Much like category theory, Cap is a powerful bookkeeping and organizational tool for high-level algorithms and can cover a wide range of computational setups as it is designed independent of any specific application. To this end the **main design goal** of Cap is *data structure agnosticism* (cf. Section VII.2).

Moreover, it can be used to organize preexisting data structures and fundamental algorithms into a proper categorical setup as shown in Chapter II. While implementing a category or a type of categories it should be left to the programmer to specify which constructions are *basic*, i.e., to be explicitly implemented, and which are *derived*, i.e., to be automatically composed by Cap from the basic ones. This is the **main feature** of Cap (cf. Section VII.2).

## 1. The concept of categorical programming

We start by introducing the concept of **categorical programming**. Many computations and constructions in mathematics can be carried out using only the basic categorical constructions mentioned in Chapter II. So those constructions form a language in which algorithms can be expressed.

We are going to show the concept of categorical programming by computing the intersection of two subobjects in a computable category $\mathcal{A}$. In Theorem IV.1.3 it was already shown that in a computable preabelian category the fiber product of two morphisms is computable. Using the fiber product we can provide a generic algorithm to compute intersections of subobjects.

**Example 1.1.** Let $\mathcal{A}$ be a preabelian category and $M \in \mathrm{Obj}_{\mathcal{A}}$ an object together with two monomorphisms $\varphi : A \hookrightarrow M$ and $\psi : B \hookrightarrow M$ which are representatives of the classes of monomorphisms defining two subobjects (cf. Section II.10).

We want to compute the intersection of the images of $\varphi$ and $\psi$ in $M$, i.e., a mono $A \cap B \hookrightarrow M$ which is a representative for the class of monos defining the intersection of the images of $\varphi$ and $\psi$. We can compute this monomorphism as

$$\mathrm{PreCompose}\left(\mathrm{ProjectionInFactorOfFiberProduct}\left(\left(\varphi, \psi\right), 1\right), \varphi\right).$$

$$\begin{tikzcd}
 & M & \\
A \arrow[ur, hook, "\varphi"] & & B \arrow[ul, hook', "\psi"'] \\
 & \mathrm{FiberProduct}\left(A, B\right) \arrow[ul, hook, "\tau"] \arrow[uu, hook, "\tau\varphi"'] \arrow[ur, hook, dashed] &
\end{tikzcd}$$

with $\tau := \mathrm{ProjectionInFactorOfFiberProduct}\left(\left(\varphi, \psi\right), 1\right)$. If $\psi : B \hookrightarrow M$ is a monomorphism, the projection $\tau$ is also a monomorphism. So the composition

$$\tau\varphi = \mathrm{PreCompose}\left(\mathrm{ProjectionInFactorOfFiberProduct}\left(\left(\varphi, \psi\right), 1\right), \varphi\right)$$

is a monomorphism and by the universal property of the fiber product a representative for the desired intersection.

As the example shows, there is a generic algorithm for intersection of subobjects if a category is computable preabelian. This means that if there is an implementation of a computable abelian category as in Chapter II, i.e., a realization together with the necessary computable functions, one can use the above intersection algorithm which is implemented in an abstract and categorical way. The framework which enables the usage of such generic categorical algorithms is the main goal of CAP.

**Example 1.2.** We use the category of rational vector spaces from Example II.4.4 and compute the intersection of two subobjects, each represented by a monomorphism.

We compute the intersection of two subobjects of a three dimensional vector space, given by the monomorphisms

$$\alpha_1 := \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

and

$$\alpha_2 := \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

```
gap> Q := HomalgFieldOfRationals();
Q
gap> Q3 := VectorSpaceObject( 3, Q );
<A vector space object over Q of dimension 3>
gap> Q2 := VectorSpaceObject( 2, Q );
<A vector space object over Q of dimension 2>
gap> alpha1 := HomalgMatrix( [[1,1,0],[0,1,1]], Q );
<A matrix over an internal ring>
gap> alpha1 := VectorSpaceMorphism( Q2, alpha1, Q3 );
<A morphism in Category of matrices over Q>
gap> alpha2 := HomalgMatrix( [[1,0,0],[0,0,1]], Q );
```

```
<A matrix over an internal ring>
gap> alpha2 := VectorSpaceMorphism( Q2, alpha2, Q3 );
<A morphism in Category of matrices over Q>
gap> pi1 := ProjectionInFactorOfFiberProduct( [ alpha1, alpha2 ], 1 );
<A morphism in Category of matrices over Q>
gap> subobj := PreCompose( pi1, alpha1 );
<A morphism in Category of matrices over Q>
gap> Display( UnderlyingMatrix( subobj ) );
[ [  -1,   0,   1 ] ]
```

The intersection is the subobject represented by the monomorphism

$$\begin{pmatrix} -1 & 0 & 1 \end{pmatrix}.$$

More sophisticated examples of categorical programming are used to build the basic operations for *generalized morphism categories* and *Serre quotient categories* in Chapter IV, as well as Algorithm VI.6.4 to compute the grade filtration of a finitely presented graded module or a coherent sheaf in Chapter VI.

## 2. Main design goal and feature

In category theory, all constructions boil down to the existential quantifiers in the definition of a category (cf. Chapter II).

**2.a. Data structure agnosticism.** CAP is completely agnostic of the given realization $\mathfrak{R}$ of a category:

(1) The user is completely free to use the data structure which is most suitable for the category he wants to implement, with the obvious restriction that a morphism has to have a Source and a Range.
(2) Equality notions for objects and morphisms are completely free to choose. In fact, they are treated in the same way as a basic categorical construction (cf. Chapter II).

These two paradigms ensure the most possible flexibility. Any GAP-object can be an object or morphism in exactly one CAP category. To tell a GAP-object $X$ it is an object or morphism in a CAP category $\mathcal{A}$, one "adds" the GAP-object $X$ as object or morphism to $\mathcal{A}$. This process of adding the $X$ to $\mathcal{A}$ enriches the data structure of $X$ with the information that $X$ is an object or morphism in $\mathcal{A}$.

This allows to declare the category membership of a data structure even a posteriori and make it possible to use existing data structures in a categorical framework. One can also use the same object or morphism class for several categories, by deciding a posteriori which category a specific instance of a class is added to.

**2.b. Selection options of basic categorical constructions.** As seen in the example of the fiber product in Section VII.1, the list of *basic categorical constructions* described in Chapter II is not unique. CAP wants to cover all possible sets of basic categorical constructions which can be used to define a computable category. Therefore any set of basic constructions for a certain type of category is a valid set of basic operations for that type of category. All other categorical constructions implemented in CAP for that type of category will be *derived* by CAP. The system for the derivations will be described in Section VII.7.

## 3. Error messages for categorical operations

Most categorical operations need stronger typing than the `GAP` declaration of the categorical operation requires:

**Example 3.1.** The definition of the computable function PreCompose for a computable category $\mathcal{A}$ is

$$\text{PreCompose} : \text{Hom}_{\mathcal{A}}(A, B) \times \text{Hom}_{\mathcal{A}}(B, C) \to \text{Hom}_{\mathcal{A}}(A, C),$$

whereas the declaration of this function in the `GAP` implementation of CAP is

$$\texttt{PreCompose} : \text{Mor}_{\mathcal{A}} \times \text{Mor}_{\mathcal{A}} \to \text{Mor}_{\mathcal{A}}.$$

So the type of the arguments in the declaration of `PreCompose` requires two arguments which are morphisms in the same category. It is not required that the range of the first argument is equal to the source of the second. Since `GAP` does not support dependent types, further specification in the declaration of the function `PreCompose` is not possible.

On the other hand the specification of the function states that for the arguments $(\varphi, \psi)$

$$(\dagger) \qquad \text{Range}(\varphi) = \text{Source}(\psi)$$

has to hold, and the behavior for the implemented function is not specified for input that does not fulfill this property. Therefore it needs to be checked whether the range of the first argument equals the source of the second. When a function is specified for the categorical operation PreCompose CAP checks the equality (†) using the IsEqualForObjects function. If the range of the first argument of PreCompose is not equal to the source of the second, an error is raised.

As seen in the above Example VII.3.1, even if the declaration of a categorical function in the `GAP` implementation of CAP does not specify all requirements on the types of the arguments, CAP itself implements a system which enriches all implemented categorical operations with checks for all their requirements.

## 4. Undecidable realizations

A realization $\mathfrak{R}$ of a computable category in the `GAP` implementation of CAP is allowed to be undecidable, i.e., there are no computable functions for IsEqualForObjects and IsEqualForMorphisms in the given realization $\mathfrak{R}$. We give a simple example.

**Definition 4.1.** Let $\mathcal{A}$ be an abelian category. We denote by $\text{Ch}(\mathcal{A})$ the category of chain complexes: The objects in $\text{Ch}(\mathcal{A})$ are chain complexes in $\mathcal{A}$ (cf. Definition VI.1.8) and the morphisms $\text{Ch}(\mathcal{A})$ are chain maps (cf. Defintion VI.1.9).

**Definition 4.2.** A **functionally defined chain complex** over a category $\mathcal{A}$ is a computable function

$$\delta : \mathbb{Z} \to \text{Mor}_{\mathcal{A}}$$

such that $\mathrm{Range}(\delta(i)) = \mathrm{Source}(\delta(i+1))$ and $\mathrm{PreCompose}(\delta(i), \delta(i+1)) = 0$ for all $i \in \mathbb{Z}$.

Using this functorial realization for the category of functionally defined complexes implies that equality of objects is impossible to decide. This is a well-known restriction for computers.[1]

If $\mathcal{A}$ is computable abelian, the category of chain complexes $\mathrm{Ch}(\mathcal{A})$ is computable abelian, with the data structure in Definition VII.4.2 as realization for objects.

**Example 4.3.** Let $\mathcal{A}$ be a category and $\mathrm{Ch}(\mathcal{A})$ the category of functionally defined chain complexes over $\mathcal{A}$. Furthermore, for $\varphi \in \mathrm{Mor}_{\mathrm{Ch}(\mathcal{A})}$ let

$$K := \mathrm{KernelObject}(\varphi),$$
$$\kappa := \mathrm{KernelEmbedding}(\varphi).$$

The objects $K$ and $\mathrm{Source}(\kappa)$ are now two distinct functions. Since it is undecidable if two functions produce the same output for every input, the question whether

$$\mathrm{IsEqualForObjects}(K, \mathrm{Source}(\kappa)) = \texttt{true}$$

cannot generally be decided by the computer.

If the implementation of a category in CAP is not able to strictly meet the requirements of a realization, it is possible to have the equality functions output a third value, namely `fail`. For the computation this value will be treated like `false`, i.e., the computation is not carried out. It will nevertheless raise a different error message, with the explanatory text “This equality is undecidable”.

If one wants to use CAP for such a category, the next two sections describe two possible solutions for the problems of such undecidable realizations, both implemented in CAP. The first approach are so-called “`WithGiven` operations” which solve the problem arising in Example VII.4.3. The second approach is *caching*, which guarantees identical output if a function is called several times with identical input.

## 5. Ensuring compatibility: `WithGiven` operations

**Example 5.1** (VII.4.3 cont.). In Example VII.4.3 we saw that computing both

$$K := \mathrm{KernelObject}(\varphi),$$
$$\kappa := \mathrm{KernelEmbedding}(\varphi)$$

in the category of functionally defined chain complexes leads to incompatible results, i.e., the equality

$$K = \mathrm{Source}(\kappa)$$

is undecidable. CAP has the following strategy to solve this problem: When $K$ has already been computed, CAP redirects the computation of KernelEmbedding to a second

[1] It is also undecidable if such an object is well-defined, as, for example, the equality $\mathrm{PreCompose}(\delta(i), \delta(i+1)) = 0$ is impossible to decide for all $i \in \mathbb{Z}$.

categorical construction, KernelEmbeddingWithGivenKernelObject:

$$\kappa := \text{KernelEmbeddingWithGivenKernelObject}\,(\varphi, K)\,.$$

This categorical construction is a second version of the algorithm to compute the embedding, which does not compute the source of $\kappa$ again, but instead uses $K$ as source. So we have

$$(\dagger) \qquad \text{Source}\,(\kappa) = K,$$

The Source $(\kappa)$ and $K$ are now represented by the same part of the memory. So the equality (†) is true regardless of the implementation of IsEqualForObjects.

The concept of such `WithGiven` operations is implemented for every categorical operation that produces a morphism where source or range is not part of the input data.

**Example 5.2.** Let $\mathcal{A}$ be a computable category and $\varphi : A \to B \in \text{Mor}_{\mathcal{A}}$. Then we have

$$\text{KernelEmbedding}\,(\varphi) =: \kappa : K \to A,$$

and the object $K$ is not defined by the input $\varphi$ of the call of KernelEmbedding. Let

$$K' := \text{KernelObject}\,(\varphi)\,.$$

Redirecting the computation of KernelEmbedding to

$$\text{KernelEmbeddingWithGivenKernelObject}\,(\varphi, K') := \kappa' : K' \to A$$

one computes a kernel embedding which source and range are predefined by the input of the function. Therefore we have ensured that

$$\text{Source}\,(\text{KernelEmbedding}\,(\varphi)) = \text{KernelObject}\,(\varphi)\,.$$

**Notation.** Let `MorphismOp` be a categorical operation which produces a morphism, for which its source or range are not part of the input data, e.g., KernelEmbedding, and `ObjectOp` the categorical operation which produces the corresponding source or range. Then the categorical operation `MorphismOpWithGivenObjectOp` has the object as last argument, and produces the morphism corresponding to `MorphismOp` for which source and range are given as input.

The `WithGiven` operations are a feature to keep the computations in sync. To implement them correctly in a category, it is important to understand how categorical operations and their `WithGiven` counterpart work together. We again take a look at Example VII.4.3:

**Example 5.3** (VII.4.3 cont.)**.** There are three methods which can be used to add functions for the kernel object and kernel embedding:

- `AddKernelObject`
- `AddKernelEmbedding`
- `AddKernelEmbeddingWithGivenKernelObject`

It is not necessary to install functions for all three of them. The two following ways ensure compatibility between calls of KernelObject and KernelEmbedding:

(1) Only provide a function for KernelEmbedding.

(2) Provide two functions, one for KernelObject and one for KernelEmbeddingWithGivenKernelObject.

In case (1), if KernelObject is called by the user with a morphism $\varphi$ as argument, CAP will invoke KernelEmbedding $(\varphi)$, and the source of the kernel embedding is returned as the result of KernelObject. The computed resulting morphism of the call of KernelEmbedding is stored in the data structure of $\varphi$, so the results of KernelObject and KernelEmbedding will stay compatible in future invocation of these commands.

In case (2), a call of KernelEmbedding with a morphism $\varphi$ as argument will first invoke KernelObject $(\varphi)$, and the resulting object $K$ of the call of KernelObject will then be passed as second argument to KernelEmbeddingWithGivenKernelObject, together with $\varphi$ as first argument. Since the result $K$ of KernelObject is also stored in $\varphi$, the results of KernelObject $(\varphi)$ and KernelEmbedding $(\varphi)$ will be compatible if one of the operations is called later.

If an universal object (e.g., the kernel object) is already computed without their corresponding universal morphism (e.g., kernel embedding) the corresponding `WithGiven` operation is always called when the corresponding morphism is computed.

## 6. Caching

Many categorical constructions are carried out by very long calculations, even more if the data structure has several stacked categories, e.g., Serre quotients (cf. Chapter IV). Also, categorical constructions often come in pairs, so that the result of one computation might be part of another, e.g., KernelObject and KernelEmbedding. For these two reasons, each categorical operation, each functor, and each natural transformation in CAP is equipped with a cache which stores the computed results.

The caches in CAP come in two flavors, weak and crisp (or strong) caches. While crisp caches store the result permanently, weak caches only hold a weak pointer to the result.

Weak pointers are a feature of garbage collected languages like `GAP`: They are a pointer to the result, but do not prevent the garbage collector from deleting the object, assuming no other pointer holds the object. This means that for compatibility reasons, weak caches are completely sufficient, since the result is stored as long as it is used somewhere else, and only recomputed if it cannot be accessed from another point, so no wrong comparisons can be performed. For performance reasons, crisp caches might be preferable, especially if the structures computed are relatively small in comparisons to the time they need to be computed. On the other hand, crisp caches produce memory leaks, since no computed result is ever deleted. CAP allows the behavior of caches to be switched at any time for any operation in any category, so one can choose a very fine granulation of which values should be stored and which not.

**6.a. Pointers and Garbage Collection.** Computer algebra systems and programming languages in general either use explicit or implicit memory management. When creating a new object, e.g., a list or a matrix, the computer needs memory to store it.

Once an object is not used in further computations, the memory it uses can be reused by different data. There are two strategies to indicate that memory can be reused:

(1) Explicit freeing of memory: When data is not used anymore, it has to be explicitly deleted. In `C/C++` this is done via the `free` command or by the `delete` keyword, which calls the destructor of an object.
(2) Implicit freeing of memory, i.e., Garbage Collection: The system analyzes whether there is still a reference to that part of memory, i.e., the object stored there can be accessed from the current session, and, if not, indicates that the memory can be used again. As long anything is referring to an object, this object remains in the memory and cannot be deleted, and the part of the memory the object uses stays occupied.

A reference to an object is called a **pointer** to such an object. A pointer can either be a variable which was set by the user in a compute session, a global variable in a program, or part of the data structure of another object.

Sometimes it is necessary to hold a pointer to an object without preventing it from being deleted. This is achieved by so-called **weak pointers**. They are references to objects without preventing the garbage collector from deleted them. This means that a weak pointer can be **valid** or **invalid**, where valid means that it still points to an object, and invalid means that the object it pointed to has been deleted.[2]

**6.b. Caches.** As mentioned above it is helpful to store the results of computations for performance or compatibility reasons. The right data structure for such a storage should be small, flexible, and ideally not causing memory leaks, i.e., making their stored data accidentally undeletable.[3] We are going to describe the data structure implemented in Cap.

**Data structure 6.1.** A **cache** $C$ **of length** $n$ consists of $n$ lists of weak pointers $key_1, \dots, key_n$, and a list $keys$ of translations between keys and results. Furthermore, it consists of $n$ equality functions $=_1, \dots, =_n$, and, depending on the type of the cache, there is

(1) a list of weak pointers, *results*, for **weak caches**;
(2) a list of pointers, *results*, for **crisp caches**.

To manipulate the cache, the following operations exists:

(1) `SetCacheValue`;
(2) `GetCacheValue`;
(3) `HasCacheValue`.

Remark 6.2. At any given time the cache $C$ can be interpreted as a map

$$\widetilde{C} : \bigtimes_{i=1}^{n} \mathbb{N} \to \mathbb{N} \uplus \{\texttt{false}\} .$$

[2]We assume the implementation of weak pointers recognizes that the object was deleted and therefore a weak pointer cannot accidentally point to a new object at the same place in memory.

[3]Crisp caches always make their data undeletable.

Since there are objects deleted from the weak pointer lists or new results stored in the cache, the map $\widetilde{C}$ which the cache $C$ represents changes during the lifetime of the cache $C$.

Remark 6.3. The `false` that $\widetilde{C}$ could possibly return is different from the value `false`, since caches should be able to store boolean values. In the Cap implementation, caches return a pair $(\texttt{true}, a)$ or the singleton $(\texttt{false})$, indicating the value $a$ is stored in the cache, or no value is stored in the cache to the input given. This way, caches can store boolean values.

We describe the three operations for caches.

**Algorithm 6.4** (`SetCacheValue`)**.** `SetCacheValue` sets a value for the cache, i.e., adds elements to the $A_i$ in the underlying map.

- **Input:** $n + 1$ objects $a_1, \ldots, a_n, a$.
- **Output:** `true` or `false`.
- **Algorithm:**
  1. Check, using $=_i$, if $a_i$ is already in $key_i$. If so, let $b_i$ the position of $a_i$ in $key_i$. If not, add $a_i$ to $key_i$, and set $b_i$ to the position where $a_i$ was added to $key_i$.
  2. Check whether the tuple $b_i := (b_1, \ldots, b_n)$ is in $keys$. If so, let $c$ the position of the tuple $b$ in $keys$ . If not, add the tuple $b$ to $keys$, and let $c$ the position where it was added.
  3. Look up whether $results_c$ is already set. If not, set $results_c$ to $a$ and return `true`. If it is set, compare the value to $a$. If they are equal, return `true`, and `false` otherwise.

**Algorithm 6.5** (`GetCacheValue` and `HasCacheValue`)**.** `GetCacheValue` gets a value from the cache, i.e., applies the map to a list of objects. `HasCacheValue` looks up whether an result is stored in the cache, i.e., the map returns something different from `false`.

- **Input:** $n$ objects $a_1, \ldots, a_n$.
- **Output:**
  - `HasCacheValue`: `true` or `false`.
  - `GetCacheValue`: An object $a$ or `false`.
- **Algorithm:**
  1. Check, using $=_i$, if $a_i$ is in $key_i$. If so, let $b_i$ the position of $a_i$ in $key_i$. If not, return `false`.
  2. Check whether the tuple $b_i := (b_1, \ldots, b_n)$ is in $keys$. If so, let $c$ the position of the tuple $b$ in $keys$ If not, return `false`.
  3. 
     - `HasCacheValue`: Look up whether $results_c$ is set, and, in case if it is a weak pointer, is valid. If so, return `true`, and `false` otherwise.
     - `GetCacheValue`: Look up whether $results_c$ is set, and, in case if it is a weak pointer, is valid. If so, return the value, and `false` otherwise.

**6.c. Caching in `GAP`.** `GAP` itself already has caching features, so-called Attributes, and, as a special case, Properties. Attributes are special unary operations, which store their output inside the argument object. To make an operation in `GAP` an attribute, one needs to declare it not via `DeclareOperation`, but via `DeclareAttribute`. Then each method installed for such an operation will store its result in a pointer (not weak) in the argument object, and will always return the stored object instead of recompute it. Those `GAP` caches have three major differences from the caches described above:

(1) They can only store objects for a single key, not for a fixed length of keys.
(2) They can only compare keys by `IsIdenticalObj` and not by another more appropriate equality function.
(3) They are always crisp, i.e., might produce memory leaks.

On the other hand, as these internal `GAP` caches are very fast, CAP takes advantage of them.

**6.d. Caching in Cap.** In CAP, each categorical operation, e.g., PreCompose or DirectSum, is by default equipped with a cache of appropriate length.

REMARK 6.6. Equipping a computable function $f: A \to B$ with a cache *changes* the function, and these changes even depend on the current `GAP` session. When implementing an algorithm for a basic categorical construction it is necessary to keep in mind that CAP uses weak caching by default.

Each call of a categorical operation first looks up the cache, and possibly returns a cached object instead of recomputing it. The corresponding equality notions can be set via `AddIsEqualForCacheForObjects` and `AddIsEqualForCacheForMorphisms`, respectively. These equality notions are by default set to the `IsEqualForObjects` and `IsEqualForMorphisms`. For simplicity reasons, it is not possible to set the equality for each cache/operation.

The behavior of the caches can be controlled for each operation in each category:

- `SetCachingWeak`: The arguments are either a category $\mathcal{C}$ or a category $\mathcal{C}$ and a string $s$. If only a category is given, all caches in the category will be set to weak. Otherwise, only the cache corresponding to the operation $s$ will be set to weak.
- `SetCachingCrisp`: The arguments are either a category $\mathcal{C}$ or a category $\mathcal{C}$ and a string $s$. If only a category is given, all caches in the category will be set to crisp. Otherwise, only the cache corresponding to the operation $s$ will be set to crisp.
- `DeactivateCaching`: The arguments are either a category $\mathcal{C}$ or a category $\mathcal{C}$ and a string $s$. If only a category is given, all caches in the category will be deactivated, i.e., neither store new values nor return any values. Otherwise, only the cache corresponding to the operation $s$ will be deactivated.

For performance reasons, unary categorical operations are implemented as Attributes, which means that the results are stored inside the object passed to the function as the single argument. Since the comparison function for Attributes is always `IsIdenticalObj` and not the appropriate notion `IsEqualForCache`, even Attributes are equipped with a second cache in the above sense. So CAP has two cache layers:

(1) If the operation is unary, e.g., KernelEmbedding, it is declared as an Attribute. This means that if the operation has already been computed for the (single) argument, the result has already been computed and stored in the specific argument, so the previously computed result is returned.
(2) If the operation has more than one argument, or has not been computed before, the cache is looked up with the category specific equality notions. If there is a result matching the argument stored in the cache, this result is returned instead of a new computed one. Otherwise the result is computed from scratch.

As mentioned above, there are two main reasons to cache results of computations: result compatibility and performance.

**6.e. Avoiding the setoid.** Using caches it is possible to go back from the definition of categories with Hom-setoids (cf. Definition II.2.2) to the classical Definition II.2.1.

Categorical constructions are functions up to the equality on the morphism sets, but the compatibility properties are defined for the congruence of morphisms. Example III.2.13 shows that using the equality of morphisms instead of congruence can lead to algorithms incompatible with the specifications of the corresponding categorical constructions. Using caches we can work around this.

**Theorem 6.7.** *Suppose a computable abelian category $\mathcal{A}$ where all categorical operations in the implementation of $\mathcal{A}$ are equipped with crisp caches. Then it is possible to set* IsEqualForMorphisms *in that category to* IsCongruentForMorphisms, *and therefore going to the reduction $\mathcal{A}'$ of $\mathcal{A}$ (cf. Theorem II.3.5) and obtaining a category in the classical sense.*

Proof. For a categorical operation $F$ two different argument lists $(a_1, \dots, a_n)$ and $(a'_1, \dots, a'_n)$ with

$$\text{IsEqualForObjects}\,(a_i, a'_i) = \texttt{true}$$

if $a_i$ is an object and

$$\text{IsCongruentForMorphisms}\,(a_i, a'_i) = \texttt{true}$$

if $a_i$ is a morphism now have to produce identical output, since the second call will always return a value identical to the first result. So for morphisms those values are again equal, and the object comparison does not matter. Furthermore, all categorical operations become functions with respect to the new IsEqualForMorphisms := IsCongruentForMorphisms. □

Now, using Theorem VII.6.7, we can achieve an implementation of $S$-grpres in the classical sense, i.e., as a classical category which has Hom-sets instead of Hom-setoids.[4]

[4]Another system that by default implements its abelian categories as classical categories is `homalg` (cf. [**hom17**]).

**6.f. Caching for compatibility.** For the category of functionally defined chain complexes from Definition VII.4.2, it is impossible to implement equality notions for objects and morphisms that are compatible with the mathematical notion of equality of chain complexes. Here caching leads to predictable and compatible output of categorical operations.

**Example 6.8.** Let $\mathcal{A}$ be the category of functionally defined chain complexes (cf. Definition VII.4.2) and $\varphi$ a morphism in $\mathcal{A}$. Let KernelEmbedding$(\varphi)$ be called twice with results $\kappa$ and $\kappa'$. Without storing the first result of the call of KernelEmbedding and returning a different `GAP`-object the second time KernelEmbedding is called, $\kappa$ and $\kappa'$ will not be equal, since

$$\mathrm{Source}(\kappa) = \mathrm{Source}(\kappa')$$

is undecidable. So it is important to store the result of the first call to avoid creating several different object which are theoretically equal but cannot be decided as such. Furthermore, other caches needs to be filled with parts of the result: If after the call of KernelEmbedding$(\varphi)$ the operation KernelObject$(\varphi)$ is called, the result $K$ should be equal to Source$(\kappa)$. This is achieved by storing the Source of the result of the KernelEmbedding in the cache of KernelObject.[5]

Cap caches ensure such a compatibility as it fills all appropriate caches with the already computed results. To ensure such a compatibility a weak cache is always sufficient, since incompatibility can only occur if the resulting object can be compared to an object another pointer points to.

**6.g. Caching for performance.** A second reason for caching is increasing the performance.

**Example 6.9.** Consider the category of rational vector spaces (cf. Example II.4.4), with caching equalities being the equalities of the category, i.e., the integer comparison for the object, and for morphism entrywise comparison of the matrix. Now, if two different morphisms with the same matrix entries are given to KernelEmbedding, there will not be any Gaussian elimination performed for the second morphism, but the cache returns the same data as for the first call of KernelEmbedding. This leads to a faster computation, and less memory usage, since the same matrix would not be computed twice.

Performance enhancing caches can be weak or crisp, which is a trade off between computation time for the result and space it needs to be stored. A fine granulated equality notion and fine tuned caches for each categorical operation can be used to enhance the performance of Cap.

**Example 6.10.** We consider again the category of rational vector spaces (cf. Example II.4.4). A cache for DirectSum will not increase performance, since the computation needed for DirectSum is just addition of two integers. On the other hand, a cache for the KernelEmbedding can reduce computation time for large matrices.

[5]Since KernelObject is an Attribute, the attribute of KernelObject of $\varphi$ will also be set to the source of the result of the call of KernelEmbedding.

## 7. Primitive and derived categorical operations

A powerful feature of CAP is deriving categorical constructions from other basic ones. In Chapter II some categorical constructions to make an abelian category computable are explicitly listed. It is already mentioned there that not all constructions are needed to be implemented explicitly but some can be derived from others (cf. Remark II.7.13). Furthermore, some very popular constructions are not mentioned in the list of basic categorical operations in Chapter II, e.g., Pullback or Pushout, nevertheless it should be possible to give explicit algorithms for these categorical constructions. To decide which categorical operations can be derived from an explicitly implemented set of basic ones and which derivations are supposed to deliver the best performance, we are going to introduce the derivation graph as a system for finding all possible derivations for a given set of explicitly implemented categorical constructions.

**7.a. Why the graph is necessary: Circular dependencies.** When implementing several categorical constructions, there are often several ways to compute a specific object or morphism. We have already seen an example in II.7.13. We give a second trivial example:

**Example 7.1.** Let $\mathcal{A}$ be a computable category and $\varphi : A \to B, \psi : B \to C$ in $\mathrm{Mor}_{\mathcal{A}}$. CAP offers two functions to compute the composition $\varphi\psi$:

$$\begin{aligned} \varphi\psi =& \mathrm{PreCompose}\left(\varphi, \psi\right), \\ \varphi\psi =& \mathrm{PostCompose}\left(\psi, \varphi\right). \end{aligned}$$

When following the definitions in II, one would ideally implement an algorithm to compute PreCompose and let the system derive PostCompose. But since the system is supposed to be modular, it should also be possible to derive PreCompose from an installed version of PostCompose.

Such derivations need to be carried out in a way which prevents the occurrence of circular dependencies. An example of a circular dependency is to derive PreCompose from PostCompose and to derive PostCompose from PreCompose.

**7.b. The derivation graph.** To organize all ways the different categorical operations can imply each other, to store the functions that derive categorical constructions from other, and to track the circular dependencies, we introduce the **derivation graph**.

This graph is not bound to a specific category but intended to be used by all categories at the same time, since it only contains the functions which can be derived by other categorical operations.

We give a definition of this graph which differs from the `GAP` implementation of CAP at certain points. We will point out the differences at the end of the section.

**Definition 7.2** (Method derivation graph)**.** Let $\mathcal{M}$ be a set of categorical operations (cf. Appendix C for the set of categorical operations implemented in CAP). The **method**

**derivation graph** is a directed hypergraph $(V, E)$[6] where the vertices $V$ are given by $\mathcal{M} \uplus \{c\}$ and the set of edges consists of the following:

(1) For every categorical operation $m \in \mathcal{M}$ there is an edge from $c$ to $m$ in $E$.
(2) One hyperedge for every derived method of a categorical operation $m \in \mathcal{M}$ consisting of the following data:
   (a) The **range** $m \in \mathcal{M}$;
   (b) The **sources** $m_1, \dots, m_n \in \mathcal{M}$, which are the categorical operations used in the derived algorithm for $m$;
   (c) The **exclusions** $\overline{m}_1, \dots, \overline{m}_k \in \mathcal{M}$ which describe operations are not allowed to be installed to derive $m$ using the derivation the current edge represents;
   (d) A **boolean function** which checks whether the category in which the derivation should be installed has a certain property, e.g., if the category is abelian;
   (e) The **source weights** $i_1, \dots, i_n \in \mathbb{Z}_{\geqslant 0}$ indicating that $m_j$ is used $i_j$ times in the derivation.

Remark 7.3. For Cap, it is possible to enrich the set of categorical operations at any time. This allows users to extend the Cap kernel, for example to implement triangulated categories. Only categories created after the extension will have access to the new derivations. Appendix C shows the full list of all categorical operations currently available in the Cap kernel.

We give examples for edges of the method derivation graph.

**Example 7.4.** For all categories, the methods PreCompose and IdentityMorphism must be given primitively. This means that the graph has an edge from $c$ to PreCompose and one from $c$ to IdentityMorphism. Obviously, in most cases there is an edge from $c$ to every element in $\mathcal{M}$.

**Example 7.5.** The categorical operation PostCompose is defined by

$$\mathrm{PostCompose}\,(\alpha, \beta) := \mathrm{PreCompose}\,(\beta, \alpha)\,.$$

Using this definition, which is already a derivation, we give an example for the second type of edges. The list of sources of the edge representing this derivation is only the operation PreCompose, the range is PostCompose. There are no exclusions for this edge, and the boolean function just returns `true`, since this derivation can be installed for every category. The integer $i_1$ is 1, since PreCompose is called once in this derivation.

**Example 7.6.** For abelian categories, one can implement a functorial direct sum, i.e., the direct sum of two morphisms, which is denoted by DirectSumFunctorial. We give a derivation for this as an example:

```
function( morphism_list )
    local direct_sum_diagram, sink, diagram;
```

[6]In a hypergraph it is $E \subseteq \mathrm{Pot}\,(V) \times \mathrm{Pot}\,(V)$ instead of $E \subseteq V \times V$.

```
        direct_sum_diagram := List( morphism_list, mor -> Range( mor ) );

        sink := List( [ 1 .. Length( morphism_list ) ], i ->
            PreCompose( morphism_list[i],
                InjectionOfCofactorOfDirectSum( direct_sum_diagram, i ) ) );

        diagram := List( morphism_list, mor -> Source( mor ) );

        return UniversalMorphismFromDirectSum( diagram, sink );

end
```

This derivation uses the operations PreCompose, InjectionOfCofactorOfDirectSum, and UniversalMorphismFromDirectSum, so those would be $m_1$, $m_2$, and $m_3$. We $i_1 := i_2 := 2$ and $i_3 := 1$. The values of $i_2$ and $i_2$, i.e., the number of times PreCompose and InjectionOfCofactorOfDirectSum are called in this derivation, depend on the input size. Since the installation of a method happens before a call of the operation, there is no way to determine the actual number of calls of the operations PreCompose and InjectionOfCofactorOfDirectSum. Most of the time this construction is called for two morphisms, so setting $i_1$ and $i_2$ to 2 is sufficient.

In the GAP implementation of Cap it is always be possible for a user to add nodes or edges to the derivation graph.

## 7.c. Installation and derivation of operations.

**Algorithm 7.7** (Derivation graph marking algorithm)**.** Suppose we have a computable category $\mathcal{A}$ for which primitive operations $p_1, \dots, p_k \in \mathcal{M}$ are installed (with weight $w_i$). To install all possible derivations we use the following algorithm:

(1) Mark all nodes with infinity and $c$ with 0.
(2) Mark all edges $(c, p_i)$ and mark nodes $p_i$ with $w_i$.
(3) Now start at $c$ and find the edge with the smallest weight which is not already marked. The weight is calculated by $\sum_{i=1}^{k} i_{m_i} w_{m_i}$, where $w_{m_i}$ is the current weight of $m_i$. Now four possibilities apply:
   (a) If there is no such edge with weight smaller than infinity, terminate.
   (b) If the range $m$ of the edge already has weight smaller infinity, remove the edge and go back to the beginning of step 3.
   (c) If any exclusion $\overline{m}_i$ of the edge already has a weight smaller than infinity, remove the edge and go back to the beginning of step 3.
   (d) Otherwise, mark the range $m$ with the calculated weight of the edge, install the corresponding derivation, remove the edge, and go back to the beginning of step 3.

Since there are only finitely many edges in the graph and each step in the algorithm removes one edge, the algorithm terminates.

This marking graph and algorithm ensures that from all the derivations for a categorical operation only the one with the smallest weight is installed. The actual CAP implementation does not follow this algorithm: Edges that do not have exclusions are installed directly if reachable from $c$, while the ones with exclusions are installed when the category is finalized, i.e., no more operations will be installed primitively.

The development of this algorithm and its CAP implementation was joint work with Øystein Skartsæterhagen.

## 8. Logic Propagation: ToDoLists

Certain properties of objects or morphisms can lead to easier computations in categorical operations:

**Example 8.1.** Let $\mathcal{V}$ be the category of finite vector spaces (cf. Example II.4.4) and $\varphi :\ A \to B$ a monomorphism in $\mathcal{V}$. For a monomorphism in $\mathcal{V}$ the kernel embedding

$$\kappa := \text{KernelEmbedding}\,(\varphi)$$

can be computed as

$$\kappa := \text{UniversalMorphismFromZeroObject}\,(\text{Source}\,(\varphi))\,,$$

i.e., without performing a Gaussian elimination. Therefore, the kernel embedding and kernel object of a morphism $\varphi$ in $\mathcal{V}$ can be computed more efficiently if $\varphi$ is known to be a monomorphism.

As the above example shows, knowledge about special properties of objects can reduce computation time. Therefore, it is important to propagate such knowledge between the objects as far and extensive as possible. We will now show how such knowledge is propagated in CAP computations.

### 8.a. Property propagation: ToDoLists.

Since knowledge of basic properties can speed up of computations, CAP offers a tool to propagate knowledge between objects when possible: ToDoLists.

**Data structure 8.2** (ToDoList entry)**.** A **ToDoList entry** $E$ consists of a list of `GAP` objects $A_i, \dots, A_n, A$, a list of properties $P_1, \dots, P_n, P$, and a list of values $a_1 \dots, a_n, a \in \{\texttt{true}, \texttt{false}\}$. A ToDoList entry with this data stored encapsulates the proposition

$$\bigwedge_{i=1}^{n} P_i\,(A_i) = a_i \Rightarrow P\,(A) = a.$$

We call the $A_i$'s the **sources** of $E$ and $A$ the **range** of $E$. We call $E$ **fulfilled** if

$$HasP_i\,(A_i) = \texttt{true} \wedge P_i\,(A_i) = a_i$$

for all $i$.

REMARK 8.3. The condition $HasP_i\,(A_i) = \texttt{true}$ in the definition of 'fulfilled' ensures that a ToDoList entry never computes a property. If the $HasP$ filter is not fulfilled, $P$ is not evaluated. The $HasP$ filter is true only if the property $P$ has already been evaluated.

ToDoList entries form the entities stored in ToDoLists.

**Data structure 8.4** (ToDoList)**.** Let $A$ be a `GAP` object. The **ToDoList** of $A$ is a list of ToDoList entries $E_i$ such that $A$ is a source of $E$.

ToDoLists keep track of entries that can be applied once the conditions are known to be satisfied. For the application of ToDoList entries the following algorithm is used:

**Algorithm 8.5.** Let $A$ be a `GAP` object and $P$ be a property. If $P$ becomes known, the applicable entries in the ToDoList $T$ of $A$ are applied with the following algorithm:

(1) Store all fulfilled entries in $T$ in a list $T_f$ and delete them from $T$.
(2) Apply all entries $E$ from $T_f$, by setting the property $P$ of the range object of the entry $E$ to $a$.

Remark 8.6. Deleting fulfilled entries from the ToDoList before applying any of them ensures that no recursion in applying entries occurs. Each entry is only applied once it becomes applicable.

**8.b. Creating ToDoList entries: The LaTeXlogic parser.** Cap provides a parser to read LaTeX files in a certain format containing theorems which are then used to create ToDoList entries.

**Example 8.7.** A theorem looks like this:

```
\begin{sequent}
\begin{align*}
  \alpha:\Mor, \beta:\Mor  ~&|~ \IsMonomorphism( \beta ) \\
  &\vdash \IsMonomorphism (
     \ProjectionInFactorOfFiberProduct( [\alpha, \beta], 1 ) )
\end{align*}
\end{sequent}
```

and in its LaTeX compiled version:

**Sequent.**

$$\begin{aligned} \alpha : \mathrm{Mor}, \beta : \mathrm{Mor} \mid \mathrm{IsMonomorphism}(\beta) \\ &\vdash \mathrm{IsMonomorphism}\big(\mathrm{ProjectionInFactorOfFiberProduct}([\alpha,\beta],1)\big) \end{aligned}$$

This logical implication states that for two morphisms $\alpha, \beta$ of a category $\mathcal{A}$, if $\beta$ is known to be a monomorphism, i.e., known to fulfill the property IsMonomorphism, then the result of

$$\mathrm{ProjectionInFactorOfFiberProduct}((\alpha,\beta),1)$$

also fulfills IsMonomorphism. The syntax for the sequents is described in the Cap manual.

In the `GAP` implementation of Cap one can add files containing such sequents and attach them to a category or type of categories. The files are then read by the Cap theorem parser, and the sequents are added to the category. When computing a categorical operation, all applicable sequents are then stored as ToDoList entries to the appropriate argument objects and in the result of the called operation.

**Example 8.8.** The following `GAP` session shows the application of the ToDoList entries coming from the sequent in Example VII.8.7 on the result of ProjectionInFactorOfFiberProduct.

```
gap> A := VectorSpaceObject( 3, Q );
<A vector space object over Q of dimension 3>
gap> B := VectorSpaceObject( 1, Q );
<A vector space object over Q of dimension 1>
gap> alpha := HomalgMatrix( [1,0,0],1,3,Q );
<A matrix over an internal ring>
gap> beta := HomalgMatrix( [0,1,0], 1,3,Q );
<A matrix over an internal ring>
gap> alpha := VectorSpaceMorphism( A, alpha, B );
gap> alpha := VectorSpaceMorphism( B, alpha, A );
<A morphism in Category of matrices over Q>
gap> beta := VectorSpaceMorphism( B, beta, A );
<A morphism in Category of matrices over Q>
gap> gamma := ProjectionInFactorOfFiberProduct( [alpha,beta], 1 );
<A morphism in Category of matrices over Q>
gap> HasIsMonomorphism( gamma );
false
gap> IsMonomorphism( beta );
true
gap> HasIsMonomorphism( gamma );
true
gap> IsMonomorphism( gamma );
true
```

Many implications are already implemented in CAP, a complete (LaTeX compiled) list can be found in Appendix B.

# Bibliography

[Bar09a] Mohamed Barakat, *The homomorphism theorem and effective computations*, Habilitation thesis, Department of Mathematics, RWTH-Aachen University, April 2009. 143, 144

[Bar09b] ______, *Spectral filtrations via generalized morphisms*, submitted (`arXiv:0904.0240`) (v2 in preparation), 2009. 141

[BIR$^+$] W. Bruns, B. Ichim, T. Römer, R. Sieg, and C. Söger, *Normaliz. algorithms for rational cones and affine monoids*, Available at `https://www.normaliz.uni-osnabrueck.de`.

[BLH11] Mohamed Barakat and Markus Lange-Hegermann, *An axiomatic setup for algorithmic homological algebra and an alternative approach to localization*, J. Algebra Appl. **10** (2011), no. 2, 269–293, (`arXiv:1003.1943`). MR 2795737 (2012f:18022) 13, 21, 23, 30, 36

[BLH14a] ______, *Characterizing Serre quotients with no section functor and applications to coherent sheaves*, Appl. Categ. Structures **22** (2014), no. 3, 457–466, (`arXiv:1210.1425`). MR 3200455 103

[BLH14b] ______, *Gabriel morphisms and the computability of Serre quotients with applications to coherent sheaves*, (`arXiv:1409.2028`), 2014. 69, 76, 79, 94

[BR08] Mohamed Barakat and Daniel Robertz, `homalg` *– A meta-package for homological algebra*, J. Algebra Appl. **7** (2008), no. 3, 299–317, (`arXiv:math.AC/0701146`). MR 2431811 (2009f:16010) 38

[CLS11] David A. Cox, John B. Little, and Henry K. Schenck, *Toric varieties*, Graduate Studies in Mathematics, vol. 124, American Mathematical Society, Providence, RI, 2011. MR 2810322 (2012g:14094) 99, 103, 104, 109

[DGPS16] Wolfram Decker, Gert-Martin Greuel, Gerhard Pfister, and Hans Schönemann, SINGULAR *4-1-0 — A computer algebra system for polynomial computations*, `http://www.singular.uni-kl.de`, 2016.

[GAP17] The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.8.7*, 2017. 12

[GJ00] Ewgenij Gawrilow and Michael Joswig, *polymake: a framework for analyzing convex polytopes*, Polytopes—combinatorics and computation (Oberwolfach, 1997), DMV Sem., vol. 29, Birkhäuser, Basel, 2000, (`http://www.polymake.org`), pp. 43–73. MR 1785292 (2001f:52033)

[GSP17] Sebastian Gutsche, Øystein Skartsæterhagen, and Sebastian Posur, *The* `CAP` *project – Categories, Algorithms, and Programming*, (`http://homalg-project.github.io/CAP_project`), 2013–2017. 12

[hom17] homalg project authors, *The* `homalg` *project – Algorithmic Homological Algebra*, (`http://homalg-project.github.io`), 2003–2017. 169

[ML71] Saunders Mac Lane, *Categories for the working mathematician*, Graduate Texts in Mathematics, no. 5, Springer-Verlag, 1971. 24

[Pos17] Sebastian Posur, *Constructive category theory and applications to equivariant sheaves*, Ph.D. thesis, University of Siegen, 2017, `http://dokumentix.ub.uni-siegen.de/opus/volltexte/2017/1179/`. 47, 52, 55, 62, 63, 70, 72, 144, 145, 147

[Stu96] Bernd Sturmfels, *Gröbner bases and convex polytopes*, University Lecture Series, vol. 8, American Mathematical Society, Providence, RI, 1996. MR 1363949 (97b:13034) 109

[tt] 4ti2 team, *4ti2—a software package for algebraic, geometric and combinatorial problems on linear spaces*, Available at www.4ti2.de.

[Wei94] Charles A. Weibel, *An introduction to homological algebra*, Cambridge Studies in Advanced Mathematics, vol. 38, Cambridge University Press, Cambridge, 1994. MR MR1269324 (95f:18001) 134, 135, 137, 144

APPENDIX A

# Programming in Cap

## 1. An overview of installing categories

We give a short overview of the three steps of the initialization process of a category in Cap.

**1.a. Creating the data structures.** When implementing a category and its objects and morphisms, the first thing to do is to implement data structures for objects and morphisms.

**Example 1.1** (Category of finite sets)**.** Let $\mathcal{A}$ be the category of finite subsets of $\mathbb{N}$. A possible data structure for objects are unordered lists of integers. Consider the set $\{1, 2, 3\}$. Then $[1, 2, 3]$ and $[1, 3, 2]$ would be possible serializations of this set, but of course they differ on the machine level.

In this example the realization differs from the implemented category: We use ordered sets (i.e., lists) as data structures, instead of sets. In Chapters IV and V we came across more sophisticated examples of this difference between the data structure and the realized category and emphasized the importance of the equality notions.

**1.b. Implementing basic algorithms.** After implementing the data structures, the next step is to implement the categorical constructions as algorithms acting on the data structures. For every categorical construction there should be one algorithm implemented or a way to derive it from given algorithms. Note that the algorithms have to give equal output on equal input according to the implemented equality functions. A list of all possible basic algorithms which can be installed and derived in Cap can be found in Appendix C.

**1.c. Finalization of the category.** Once all necessary primitive operations are implemented the finalization tells Cap that the initialization process if the category is now completed. This will lead trigger the derivation mechanism (cf. Section VII.7) and derive all possible constructions for that type of category.

## 2. The category object

In the `GAP` implementation of Cap the category object itself is a large object. It contains several information about the category and can itself be used to create new categories. We are going to give a short description of the important components of the category object:

(1) **Name**: For every category the name of the category is stored. The name is used to display the category, but also to automatically create new names, e.g., names of

functors and natural transformations. Since functors and natural transformations can be installed as global functions, using their names as global names, the name of the category should be unique.

(2) **Filters**: Each category holds two unique filters (cf. Section A.5): One for objects and one for morphisms in the category. These filters are used to identify objects and morphisms that belong to the category.
(3) **Caches**: Each category stores all caches for categorical operations of that category (cf. Section VII.6).
(4) **Logical implications**: Each category stores applicable sequents (cf. Section VII.8) for this category.
(5) **Precondition check**: Most of the categorical operations have preconditions, e.g., the two arguments for PreCompose have to be composable. Checking these preconditions is useful when doing experiments, but slows down large computations for which all input data is correct. So each category holds a boolean value which indicates whether the preconditions should be checked for categorical operations or not.
(6) **Finalization indicator**: Each category stores a boolean value indicating if the category is finalized, i.e., all primitive operations for that category are already installed. Since the finalization of a category triggers the derivation process (cf. Section VII.7), a non-finalized category should not be used for computations.

## 3. Functors and natural transformations: The category of categories

CAP features an implementation of the category of categories, `CapCat`. Its objects are the CAP category objects themselfs, and its morphisms are the CAP functors. It is currently the only 2-category in CAP. Its 2-cells are natural transformations. The category `CapCat` is also a CAP category, which means the same categorical operations can be used to manipulate with objects and morphisms in `CapCat`.

### 3.a. Functors.

**Definition 3.1.** Let $\mathcal{A}, \mathcal{B}$ be categories. A **(covariant) functor** $F : \mathcal{A} \to \mathcal{B}$ consists of two maps

$$F_0 : \mathrm{Obj}_{\mathcal{A}} \to \mathrm{Obj}_{\mathcal{B}},\ A \mapsto F(A),$$
$$F_1 : \mathrm{Mor}_{\mathcal{A}} \to \mathrm{Mor}_{\mathcal{B}}, \varphi \mapsto F(\varphi),$$

such that

$$\begin{aligned}\mathrm{Source}(F(\varphi)) &= F(\mathrm{Source}(\varphi)),\\ \mathrm{Range}(F(\varphi)) &= F(\mathrm{Range}(\varphi)),\\ F(\varphi\psi) &\sim F(\varphi)F(\psi),\ \text{and}\\ F(\mathrm{id}_A) &\sim \mathrm{id}_{F(A)}.\end{aligned}$$

Two functors are composed by composing the maps $F_i$, and the identity functor is the functor where both $F_1$ and $F_2$ are the identity maps.

Covariant functors form the morphisms in the category of categories.

**Definition 3.2.** The **category of categories** in CAP, `CapCat`, is the category with categories as objects and functors as morphisms.

The main reason for the existence of `CapCat` in CAP is the implementation of functors and natural transformations.

**3.b. Implementation of functors.** The implementation of a functor $F$ consists of algorithms for the two functions $F_0$ and $F_1$ in Definition A.3.1.

**Data structure 3.3** (Functors)**.** Let $\mathcal{A}$ and $\mathcal{B}$ be categories and $F : \mathcal{A} \to \mathcal{B}$ a functor. The data structure of $F$ consists of the following two functions:

$$\begin{aligned}\mathrm{ObjectFunction}\,(F) &: \mathrm{Obj}_{\mathcal{A}} \to \mathrm{Obj}_{\mathcal{B}},\\ \mathrm{MorphismFunction}\,(F) &: \mathrm{Obj}_{\mathcal{B}} \times \mathrm{Mor}_{\mathcal{A}} \times \mathrm{Obj}_{\mathcal{B}} \to \mathrm{Mor}_{\mathcal{B}}.\end{aligned}$$

Both functions are cached independently.

The ObjectFunction has the role of $F_0$. The MorphismFunction has the role of $F_1$ and can be interpreted as a `WithGiven` function (see Section VII.5 for more details).

**Algorithm 3.4** (Functor evaluation)**.** Let $\mathcal{A}$ and $\mathcal{B}$ be categories, $F : \mathcal{A} \to \mathcal{B}$ a functor, and $A \in \mathrm{Obj}_{\mathcal{A}}$, $\varphi \in \mathrm{Mor}_{\mathcal{A}}$. The operation `ApplyFunctor` is used to evaluate the functor at objects or morphisms of $\mathcal{A}$ using the following algorithm:

(1)
$$\texttt{ApplyFunctor}\,(F, A) := \mathrm{ObjectFunction}\,(F)\,(A)\,.$$

(2) Let
$$\begin{aligned}A' &:= \mathrm{ObjectFunction}\,(F)\,(\mathrm{Source}\,(\varphi))\,,\\ B' &:= \mathrm{ObjectFunction}\,(F)\,(\mathrm{Range}\,(\varphi))\,.\end{aligned}$$

Then
$$\psi := \texttt{ApplyFunctor}\,(F, \varphi) := \mathrm{MorphismFunction}\,(F)\,(A', \varphi, B')\,,$$
such that $\mathrm{Source}\,(\psi) = A'$ and $\mathrm{Range}\,(\psi) = B'$.

This algorithm guarantees the compatibility of the result, since the result of the evaluation of a morphism $\varphi$ has the evaluations of the source and range of $\varphi$ as source and range of the result $\psi$, respectively.

REMARK 3.5. All functors in CAP are covariant. To implement contravariant or multivariate functors the opposite and the product category (cf. Section A.4) are used.

**3.c. Natural transformations.** Natural transformations are morphisms between functors. They are implemented as 2-cells in the category of categories. The corresponding categorical operations are:

(1) IdentityTwoCell,
(2) HorizontalPreCompose,
(3) and VerticalPreCompose.

The data structure contains two functors, serving as source and range, and a function defining how the natural transformation acts on objects.

**Definition 3.6.** Let $\mathcal{A}$ and $\mathcal{B}$ be categories and $F, G : \mathcal{A} \to \mathcal{B}$ functors. A **natural transformation** $\mathcal{N} : F \Rightarrow G$ consists of a map

$$\mathcal{N}_0 : \mathrm{Obj}_{\mathcal{A}} \to \mathrm{Mor}_{\mathcal{B}},\ A \mapsto (\mathcal{N}(A) : F(A) \to G(A))$$

such that for any morphism $\varphi : A \to B \in \mathrm{Mor}_{\mathcal{A}}$ the following diagram commutes up to congruence:

$$\begin{tikzcd}
F(A) \arrow[r, "\mathcal{N}(A)"] \arrow[d, "F(\varphi)"'] & G(A) \arrow[d, "G(\varphi)"] \\
F(B) \arrow[r, "\mathcal{N}(B)"] & G(B)
\end{tikzcd}$$

**Data structure 3.7.** Let $\mathcal{A}, \mathcal{B}$, $F, G$ and $\mathcal{N}$ be as in Definition A.3.6. The data structure for the natural transformation $\mathcal{N}$ consists of the functor $F$ as source, the functor $G$ as range, and a function

$$\begin{aligned}\mathrm{NaturalTransformationFunction}(\mathcal{N}) :& \mathrm{Obj}_{\mathcal{B}} \times \mathrm{Obj}_{\mathcal{A}} \times \mathrm{Obj}_{\mathcal{B}} \to \mathrm{Mor}_{\mathcal{B}},\\ & (B_1, A, B_2) \mapsto (\mathcal{N} : B_1 \to B_2).\end{aligned}$$

The function NaturalTransformationFunction is 3-ary for same reason as the function MorphismFunction for functors is 3-ary: When a natural transformation $\mathcal{N}$ is applied to an object $A \in \mathrm{Obj}_{\mathcal{A}}$, the resulting morphism should have the object $F(A)$ as source and $G(A)$ as range. Giving source and range of the resulting morphism as arguments to the natural transformation function guarantees this compatibility condition.

**Algorithm 3.8** (Natural transformation evaluation)**.** Let $\mathcal{A}, \mathcal{B}$, $F, G$ and $\mathcal{N}$ as in Definition A.3.6 and $A \in \mathrm{Obj}_{\mathcal{A}}$. A natural transformation is applied via

`ApplyNaturalTransformation`

using the following: Set

$$\begin{aligned} A_F &:= \texttt{ApplyFunctor}(F, A),\\ A_G &:= \texttt{ApplyFunctor}(G, A).\end{aligned}$$

Then it is

$$\begin{aligned}\psi := \texttt{ApplyNaturalTransformation}(\mathcal{N}, A)\\ := \mathrm{NaturalTransformationFunction}(\mathcal{N})(A_F, A, A_G),\end{aligned}$$

such that $\mathrm{Source}(\psi) = A_F$ and $\mathrm{Range}(\psi) = A_G$.

Again, natural transformations cache their output.

## 4. Special categories implemented in Cap

There are three special categories in CAP. We give a description of those categories and motivate their existence.

### 4.a. Opposite category.

**Definition 4.1.** Let $\mathcal{A}$ be a category. The **opposite category** $\mathcal{A}^{\mathrm{op}}$ is defined by

$$\begin{aligned} \mathrm{Obj}_{\mathcal{A}^{\mathrm{op}}} &:= \{A^{\mathrm{op}} \mid A \in \mathrm{Obj}_{\mathcal{A}}\}, \\ \mathrm{Mor}_{\mathcal{A}^{\mathrm{op}}} &:= \{\varphi^{\mathrm{op}} : B^{\mathrm{op}} \to A^{\mathrm{op}} \mid \varphi : A \to B \in \mathrm{Mor}_{\mathcal{A}}\}. \end{aligned}$$

In the `GAP` implementation of CAP the opposite category $\mathcal{A}^{\mathrm{op}}$ of a category $\mathcal{A}$ is constructed via `Opposite` $(\mathcal{A})$. The `Opposite` command also constructs objects and morphisms in the opposite category $\mathcal{A}^{\mathrm{op}}$ out of objects and morphisms of the underlying category $\mathcal{A}$.

Using the opposite category one can implement contravariant functors:

**Proposition 4.2.** *Let $\mathcal{A}$ and $\mathcal{B}$ be categories and $F : \mathcal{A} \to \mathcal{B}$ a contravariant functor. Then the functor $\widetilde{F} : A^{\mathrm{op}} \to B$ with*

$$\begin{aligned} \widetilde{F}\,(A^{\mathrm{op}}) &:= F\,(A) \\ \widetilde{F}\,(\varphi^{\mathrm{op}}) &:= F\,(\varphi) \end{aligned}$$

*is covariant.*

The implementation of the opposite category enables CAP to only use covariant functors as morphisms in `CapCat`. Contravariant functors are modeled as covariant functors having the opposite category as source.

### 4.b. Product category.

**Definition 4.3.** Let $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be categories. The **product category** $\mathcal{A} := \mathcal{A}_1 \times \cdots \times \mathcal{A}_n$ is defined by

$$\begin{aligned} \mathrm{Obj}_{\mathcal{A}} &:= \bigtimes_{i=1}^{n} \mathrm{Obj}_{\mathcal{A}_i}, \\ \mathrm{Mor}_{\mathcal{A}} &:= \bigtimes_{i=1}^{n} \mathrm{Mor}_{\mathcal{A}_i}. \end{aligned}$$

The product category is used to implement multivariate functors, i.e., functors that take more than one argument. The product of a list of categories can be created by the `Product` command. With the same command, objects and morphisms in the product category can be created.

### 4.c. Terminal category.

**Definition 4.4.** The **terminal category** $\mathcal{T}$ is a single object, single morphism category defined by

$$\begin{aligned}\mathrm{Obj}_{\mathcal{T}} &:= \{T\}\,,\\ \mathrm{Mor}_{\mathcal{T}} &:= \{\mathrm{id}_T : T \to T\}\,.\end{aligned}$$

It is an abelian category, where each categorical construction is either $T$ or $\mathrm{id}_T$.

**Proposition 4.5.** *The terminal category $\mathcal{T}$ is a terminal object in the category of categories,* `CapCat`*.*

In CAP, the terminal category can be accessed using `TerminalCategory`. The object $T$ and the morphism $\mathrm{id}_T$ can be accessed using `UniqueObject` and `UniqueMorphism`.

The category itself can be used to create single valued functors, which can again be used to evaluate multivariate functors at certain arguments.

**Example 4.6.** Let $\mathcal{A}$ be an abelian category and $A \in \mathrm{Obj}_{\mathcal{A}}$. Consider the functor

$$F : \mathcal{A} \to \mathcal{A},\ X \mapsto \mathrm{DirectSum}\,(A, X)$$

on objects, and with DirectSumFunctorial on morphisms. This functor can then be written as the sum of the functors

$$\begin{aligned}F_1 &: \mathcal{T} \to \mathcal{A},\, T \mapsto A\\ F_2 &: \mathcal{A} \to \mathcal{A},\, X \mapsto X.\end{aligned}$$

$F_1$ can be created using the command `FunctorFromTerminalCategory`, and $F_2$ using IdentityMorphism. Since functors are morphisms in the category of categories, the categorical construction DirectSumFunctorial can be used to produce the functor $F$.

## 5. Filters and Method Selection

We present the system that enables `GAP` to implement special methods for objects or morphisms with additional properties, e.g., a special method to compute the kernel object for monomorphisms.

### 5.a. Filters.

**Definition 5.1** (Filter)**.** Let $A$ be a `GAP` object.

(1) A **filter** $F$ is a boolean flag written as a function, i.e.,

$$F\,(A) \in \{\texttt{true}, \texttt{false}\}\,.$$

(2) A **property** $P$ is a boolean function together with a filter $HasP$. $P$ can only be evaluated once for any object $A$, and it is

$$P\,(A) \in \{\texttt{true}, \texttt{false}\}\,.$$

It is

$$HasP\,(A) = \texttt{true}$$

if and only if $P$ has already been evaluated.

Filters can be concatenated via the operation `and`. Such concatenated filters are evaluated as follows:

**Algorithm 5.2.** Let $A$ be a `GAP` object and $P$ and $Q$ filters. The filter $S := P$ `and` $Q$ is evaluated as follows:

- If $P\left(A\right) =$ `false`, then $S\left(A\right) :=$ `false` *without* looking at $Q$.
- If $P\left(A\right) =$ `true`, then $S\left(A\right) := Q\left(A\right)$.

This lazy evaluation practice makes it possible to use properties as filters.

**Definition 5.3** (Properties as Filters)**.** Let $A$ be a `GAP` object and $P$ be a property. Then $P$ can be used as a filter, evaluated like the filter

$$HasP \text{ and } P,$$

i.e., the filter can only be true if $P$ has been evaluated before.

This evaluation strategy for properties ensures that properties used as filters do not trigger expensive computations. Filters are used to decide which method to use for an operation, and triggering expensive computation when deciding which method to use can lead to more expensive computations than applying a generic function.

**5.b. Method selection.** `GAP` provides the possibility of gluing partial methods together to an operation, the so-called **Predicate Dispatch**.

A `GAP` **operation** $X$ is a callable object which reassembles a $n$-ary function for $0 \leqslant n \leqslant 6$. It is declared with a list of filters $F$ of length $n$ and equipped with **methods** $m_1, \ldots, m_k$. Each method $m_j$ is an $n$-ary function $h_j$ together with a filter list $G_j$ of length $n$, such that for each `GAP` object $A$ holds

$$G_{j,i}\left(A\right) \Rightarrow F_i\left(A\right),$$

$j = 1, \ldots, k$. If $X$ is called with $n$ arguments $A_1, \ldots, A_n$, the best matching function $h_j$ according to the filter list $G_j$ is called. The procedure of choosing the correct method for an operation is called **Method Selection**.

APPENDIX B

# Logical theorems in Cap

## 1. Logic for all categories

**Sequent 1.**

$$A : \mathrm{Obj} \mid \mathrm{IsZero}(A) \vdash \mathrm{IsTerminal}(A)$$

**Sequent 2.**

$$A : \mathrm{Obj} \mid \mathrm{IsZero}(A) \vdash \mathrm{IsInitial}(A)$$

**Sequent 3.**

$$A : \mathrm{Obj} \mid \mathrm{IsZero}(A) \vdash \mathrm{IsInjective}(A)$$

**Sequent 4.**

$$A : \mathrm{Obj} \mid \mathrm{IsZero}(A) \vdash \mathrm{IsProjective}(A)$$

**Sequent 5.**

$$\alpha : \mathrm{Mor} \mid \mathrm{IsIsomorphism}(\alpha) \vdash \mathrm{IsSplitMonomorphism}(\alpha)$$

**Sequent 6.**

$$\alpha : \mathrm{Mor} \mid \mathrm{IsIsomorphism}(\alpha) \vdash \mathrm{IsSplitEpimorphism}(\alpha)$$

**Sequent 7.**

$$\alpha : \mathrm{Mor} \mid \mathrm{IsOne}(\alpha) \vdash \mathrm{IsAutomorphism}(\alpha)$$

**Sequent 8.**

$$\alpha : \mathrm{Mor} \mid \mathrm{IsAutomorphism}(\alpha) \vdash \mathrm{IsIsomorphism}(\alpha)$$

**Sequent 9.**

$$\alpha : \mathrm{Mor} \mid \mathrm{IsAutomorphism}(\alpha) \vdash \mathrm{IsEndomorphism}(\alpha)$$

**Sequent 10.**

$$\begin{aligned}\alpha : \mathrm{Mor} \mid \mathrm{IsEndomorphism}(\alpha), \mathrm{IsIsomorphism}(\alpha)\\ \vdash \mathrm{IsAutomorphism}(\alpha)\end{aligned}$$

**Sequent 11.**

$$\alpha : \mathrm{Mor} \mid \mathrm{IsSplitMonomorphism}(\alpha) \vdash \mathrm{IsMonomorphism}(\alpha)$$

**Sequent 12.**

$$\alpha : \mathrm{Mor} \mid \mathrm{IsSplitEpimorphism}(\beta) \vdash \mathrm{IsEpimorphism}(\beta)$$

**Sequent 13.**

$$A : \mathrm{Obj} \mid () \vdash \mathrm{IsIdempotent}(\mathrm{IdentityMorphism}(A))$$

**Sequent 14.**

$$A : \mathrm{Obj} \mid () \vdash \mathrm{IsOne}\big(\mathrm{IdentityMorphism}(A)\big)$$

**Sequent 15.**

$$A : \mathrm{Obj} \mid () \vdash \mathrm{IsIdenticalToIdentityMorphism}\left(\mathrm{IdentityMorphism}(A)\right)$$

**Sequent 16.**

$$\mid () \vdash \mathrm{IsZero}\big(\,\mathrm{ZeroObject}()\big)$$

**Sequent 17.**

$$\begin{aligned}&\alpha : \mathrm{Mor}, \beta : \mathrm{Mor} \mid \mathrm{IsMonomorphism}(\beta)\\ &\qquad \vdash \mathrm{IsMonomorphism}\big(\mathrm{ProjectionInFactorOfFiberProduct}([\alpha,\beta],1)\big)\end{aligned}$$

**Sequent 18.**

$$\begin{aligned}&\alpha : \mathrm{Mor}, \beta : \mathrm{Mor} \mid \mathrm{IsMonomorphism}(\alpha)\\ &\qquad \vdash \mathrm{IsMonomorphism}\big(\mathrm{ProjectionInFactorOfFiberProduct}([\alpha,\beta],2)\big)\end{aligned}$$

**Sequent 19.**

$$\begin{aligned}&\alpha : \mathrm{Mor}, \beta : \mathrm{Mor} \mid \mathrm{IsEpimorphism}(\alpha)\\ &\qquad \vdash \mathrm{IsEpimorphism}\big(\mathrm{InjectionOfCofactorOfPushout}([\alpha,\beta],2)\big)\end{aligned}$$

**Sequent 20.**

$$\begin{aligned}&\alpha : \mathrm{Mor}, \beta : \mathrm{Mor} \mid \mathrm{IsEpimorphism}(\beta)\\ &\qquad \vdash \mathrm{IsEpimorphism}\big(\mathrm{InjectionOfCofactorOfPushout}([\alpha,\beta],1)\big)\end{aligned}$$

**Sequent 21.**

$$\begin{aligned}&\alpha : \mathrm{Mor} \mid ()\\ &\qquad \vdash \mathrm{IsMonomorphism}\big(\,\mathrm{KernelEmbedding}(\alpha)\big)\end{aligned}$$

**Sequent 22.**

$$\begin{aligned}&\alpha : \mathrm{Mor} \mid ()\\ &\qquad \vdash \mathrm{IsEpimorphism}\big(\,\mathrm{CokernelProjection}(\alpha)\big)\end{aligned}$$

**Sequent 23.**

$$\begin{aligned}&\alpha : \mathrm{Mor}, \beta : \mathrm{Mor} \mid ()\\ &\qquad \vdash \mathrm{IsMonomorphism}\big(\mathrm{Equalizer}(\alpha,\beta)\big)\end{aligned}$$

**Sequent 24.**

$$\begin{aligned}\alpha : \mathrm{Mor}, \beta : \mathrm{Mor} \mid ()\\ \vdash \mathrm{IsEpimorphism}\big(\mathrm{Coequalizer}(\alpha, \beta)\big)\end{aligned}$$

**Sequent 25.**

$$\begin{aligned}\alpha : \mathrm{Mor} \mid \mathrm{IsTerminal}\big(\mathrm{Source}(\alpha)\big)\\ \vdash \mathrm{IsSplitMonomorphism}(\alpha)\end{aligned}$$

**Sequent 26.**

$$\begin{aligned}\alpha : \mathrm{Mor} \mid \mathrm{IsInitial}\big(\mathrm{Range}(\alpha)\big)\\ \vdash \mathrm{IsSplitEpimorphism}(\alpha)\end{aligned}$$

**Sequent 27.**

$$L : \mathrm{ListObj} \mid \big(\forall x \in L : \mathrm{IsTerminal}(x)\big) \vdash \mathrm{IsTerminal}\big(\mathrm{DirectProduct}(L)\big)$$

**Sequent 28.**

$$L : \mathrm{ListObj} \mid \big(\forall x \in L : \mathrm{IsInitial}(x)\big) \vdash \mathrm{IsInitial}\big(\mathrm{Coproduct}(L)\big)$$

**Sequent 29.**

$$\begin{aligned}\alpha : \mathrm{Mor}, \beta : \mathrm{Mor} \mid \mathrm{IsMonomorphism}(\alpha), \mathrm{IsMonomorphism}(\beta)\\ \vdash \mathrm{IsMonomorphism}\big(\mathrm{PreCompose}(\alpha, \beta)\big)\end{aligned}$$

**Sequent 30.**

$$\begin{aligned}\alpha : \mathrm{Mor}, \beta : \mathrm{Mor} \mid \mathrm{IsEpimorphism}(\alpha), \mathrm{IsEpimorphism}(\beta)\\ \vdash \mathrm{IsEpimorphism}\big(\mathrm{PreCompose}(\alpha, \beta)\big)\end{aligned}$$

**Sequent 31.**

$$\alpha : \mathrm{Mor} \mid \mathrm{IsIsomorphism}(\alpha) \qquad \vdash \mathrm{IsIsomorphism}(\mathrm{InverseImmutable}(\alpha))$$

**Sequent 32.**

$$\begin{aligned}\alpha : \mathrm{Mor} \mid ()\\ \vdash \mathrm{IsMonomorphism}\big(\mathrm{ImageEmbedding}(\alpha)\big)\end{aligned}$$

## 2. Logic for preadditive categories

**Sequent 33.**

$$a : \mathrm{Obj}, b : \mathrm{Obj} \mid () \vdash \mathrm{IsZero}\big(\mathrm{ZeroMorphism}(a, b)\big)$$

**Sequent 34.**

$$a : \mathrm{Obj}, b : \mathrm{Obj} \mid () \vdash \mathrm{IsIdenticalToZeroMorphism}\ \big(\mathrm{ZeroMorphism}(a, b)\big)$$

**Sequent 35.**

$$a : \mathrm{Obj} \mid () \vdash \mathrm{IsZero}\big(\mathrm{UniversalMorphismIntoZeroObject}(a)\big)$$

**Sequent 36.**

$$a : \mathrm{Obj} \mid () \vdash \mathrm{IsZero}\big(\mathrm{UniversalMorphismFromZeroObject}(a)\big)$$

**Sequent 37.**

$$\alpha : \mathrm{Mor} \mid \mathrm{IsZero}(\mathrm{Source}(\alpha)) \vdash \mathrm{IsZero}(\alpha)$$

**Sequent 38.**

$$\alpha : \mathrm{Mor} \mid \mathrm{IsZero}(\mathrm{Range}(\alpha)) \vdash \mathrm{IsZero}(\alpha)$$

**Sequent 39.**

$$\alpha : \mathrm{Mor} \mid \mathrm{IsZero}(\alpha), \mathrm{IsMonomorphism}(\alpha) \vdash \mathrm{IsZero}(\mathrm{Source}(\alpha))$$

**Sequent 40.**

$$\alpha : \mathrm{Mor} \mid \mathrm{IsZero}(\alpha), \mathrm{IsEpimorphism}(\alpha) \vdash \mathrm{IsZero}(\mathrm{Range}(\alpha))$$

**Sequent 41.**

$$\alpha : \mathrm{Mor}, \beta : \mathrm{Mor} \mid \mathrm{IsZero}(\alpha) \vdash \mathrm{IsZero}(\mathrm{PreCompose}(\alpha, \beta))$$

**Sequent 42.**

$$\alpha : \mathrm{Mor}, \beta : \mathrm{Mor} \mid \mathrm{IsZero}(\beta) \vdash \mathrm{IsZero}(\mathrm{PreCompose}(\alpha, \beta))$$

**Sequent 43.**

$$\alpha : \mathrm{Mor} \mid \mathrm{IsInitial}\big(\mathrm{KernelObject}(\alpha)\big) \vdash \mathrm{IsMonomorphism}(\alpha)$$

**Sequent 44.**

$$\alpha : \mathrm{Mor} \mid \mathrm{IsMonomorphism}(\alpha) \vdash \mathrm{IsZero}(\mathrm{KernelObject}(\alpha))$$

**Sequent 45.**

$$\alpha : \mathrm{Mor} \mid \mathrm{IsZero}\big(\mathrm{KernelEmbedding}(\alpha)\big) \vdash \mathrm{IsMonomorphism}(\alpha)$$

**Sequent 46.**

$$\alpha : \mathrm{Mor} \mid \mathrm{IsMonomorphism}(\alpha) \vdash \mathrm{IsZero}(\mathrm{KernelEmbedding}(\alpha))$$

**Sequent 47.**

$$\alpha : \mathrm{Mor} \mid \mathrm{IsTerminal}\big(\mathrm{CokernelObject}(\alpha)\big) \vdash \mathrm{IsEpimorphism}(\alpha)$$

**Sequent 48.**

$$\alpha : \mathrm{Mor} \mid \mathrm{IsEpimorphism}(\alpha) \vdash \mathrm{IsZero}\big(\mathrm{CokernelObject}(\alpha)\big)$$

**Sequent 49.**

$$\alpha : \mathrm{Mor} \mid \mathrm{IsZero}\big(\mathrm{CokernelProjection}(\alpha)\big) \vdash \mathrm{IsEpimorphism}(\alpha)$$

**Sequent 50.**

$$\alpha : \mathrm{Mor} \mid \mathrm{IsEpimorphism}(\alpha) \vdash \mathrm{IsZero}\big(\mathrm{CokernelProjection}(\alpha)\big)$$

## 3. Logic for additive categories

**Sequent 51.**

$$A : \mathrm{Obj} \mid \mathrm{IsTerminal}(A) \vdash \mathrm{IsZero}(A)$$

**Sequent 52.**

$$A : \mathrm{Obj} \mid \mathrm{IsInitial}(A) \vdash \mathrm{IsZero}(A)$$

**Sequent 53.**

$$a : \mathrm{Obj} \mid \mathrm{IsZero}\big(a\big) \vdash \mathrm{IsZero}\big(\mathrm{IdentityMorphism}(a)\big)$$

**Sequent 54.**

$$\begin{aligned} a : \mathrm{Obj}, b : \mathrm{Obj} \mid \mathrm{IsZero}\big(a\big), \mathrm{IsZero}\big(b\big)& \\ \vdash \mathrm{IsZero}\big(\, \mathrm{DirectSum}([a, b])\big)& \end{aligned}$$

## 4. Logic for abelian categories

**Sequent 55.**

$$\alpha : \mathrm{Mor} \mid \mathrm{IsMonomorphism}(\alpha), \mathrm{IsEpimorphism}(\alpha) \vdash \mathrm{IsIsomorphism}(\alpha)$$

**Sequent 56.**

$$\begin{aligned} \alpha : \mathrm{Mor} \mid \mathrm{IsEpimorphism}\big(\alpha\big)& \\ \vdash \mathrm{IsIsomorphism}\big(\mathrm{ImageEmbedding}(\alpha)\big)& \end{aligned}$$

**Sequent 57.**

$$\begin{aligned} \alpha : \mathrm{Mor} \mid \mathrm{IsIsomorphism}\big(\mathrm{ImageEmbedding}(\alpha)\big)& \\ \vdash \mathrm{IsEpimorphism}\big(\alpha\big)& \end{aligned}$$

**Sequent 58.**

$$\begin{aligned} \alpha : \mathrm{Mor} \mid \mathrm{IsEpimorphism}\big(\alpha\big)& \\ \vdash \mathrm{IsIsomorphism}\big(\, \mathrm{AstrictionToCoimage}(\alpha)\big)& \end{aligned}$$

**Sequent 59.**

$$\begin{aligned} \alpha : \mathrm{Mor} \mid \mathrm{IsIsomorphism}\big(\, \mathrm{AstrictionToCoimage}(\alpha)\big)& \\ \vdash \mathrm{IsEpimorphism}\big(\alpha\big)& \end{aligned}$$

**Sequent 60.**

$$\begin{aligned} \alpha : \mathrm{Mor} \mid \mathrm{IsMonomorphism}\big(\alpha\big)& \\ \vdash \mathrm{IsIsomorphism}\big(\, \mathrm{CoimageProjection}(\alpha)\big)& \end{aligned}$$

**Sequent 61.**

$$\begin{aligned} \alpha : \mathrm{Mor} \mid \mathrm{IsIsomorphism}\big(\, \mathrm{CoimageProjection}(\alpha)\big)& \\ \vdash \mathrm{IsMonomorphism}\big(\alpha\big)& \end{aligned}$$

**Sequent 62.**

$$\begin{aligned}&\alpha : \mathrm{Mor} \mid \mathrm{IsMonomorphism}\big(\alpha\big)\\ &\qquad \vdash \mathrm{IsIsomorphism}\big(\,\mathrm{CoastrictionToImage}(\alpha)\big)\end{aligned}$$

**Sequent 63.**

$$\begin{aligned}&\alpha : \mathrm{Mor} \mid \mathrm{IsIsomorphism}\big(\,\mathrm{CoastrictionToImage}(\alpha)\big)\\ &\qquad \vdash \mathrm{IsMonomorphism}\big(\alpha\big)\end{aligned}$$

APPENDIX C

# All method names

- AdditionForMorphisms
- AdditiveInverseForMorphisms
- AssociatorLeftToRightWithGivenTensorProducts
- AssociatorRightToLeftWithGivenTensorProducts
- AstrictionToCoimage
- AstrictionToCoimageWithGivenCoimage
- BraidingInverseWithGivenTensorProducts
- BraidingWithGivenTensorProducts
- CoastrictionToImage
- CoastrictionToImageWithGivenImageObject
- CoevaluationForDualWithGivenTensorProduct
- CoevaluationMorphismWithGivenRange
- Coimage
- CoimageProjection
- CoimageProjectionWithGivenCoimage
- CokernelColift
- CokernelColiftWithGivenCokernelObject
- CokernelFunctorialWithGivenCokernelObjects
- CokernelObject
- CokernelProjection
- CokernelProjectionWithGivenCokernelObject
- Colift
- ColiftAlongEpimorphism
- Coproduct
- CoproductFunctorialWithGivenCoproducts
- DirectProduct
- DirectProductFunctorialWithGivenDirectProducts
- DirectSum
- DirectSumCodiagonalDifference
- DirectSumDiagonalDifference
- DirectSumFunctorialWithGivenDirectSums
- DirectSumProjectionInPushout
- DualOnMorphismsWithGivenDuals
- DualOnObjects

- EvaluationForDualWithGivenTensorProduct
- EvaluationMorphismWithGivenSource
- FiberProduct
- FiberProductEmbeddingInDirectSum
- FiberProductFunctorialWithGivenFiberProducts
- HorizontalPostCompose
- HorizontalPreCompose
- IdentityMorphism
- IdentityTwoCell
- ImageEmbedding
- ImageEmbeddingWithGivenImageObject
- ImageObject
- InitialObject
- InitialObjectFunctorial
- InjectionOfCofactorOfCoproduct
- InjectionOfCofactorOfCoproductWithGivenCoproduct
- InjectionOfCofactorOfDirectSum
- InjectionOfCofactorOfDirectSumWithGivenDirectSum
- InjectionOfCofactorOfPushout
- InjectionOfCofactorOfPushoutWithGivenPushout
- InternalHomOnMorphismsWithGivenInternalHoms
- InternalHomOnObjects
- InternalHomToTensorProductAdjunctionMap
- InverseImmutable
- InverseMorphismFromCoimageToImageWithGivenObjects
- IsAutomorphism
- IsCodominating
- IsCongruentForMorphisms
- IsDominating
- IsEndomorphism
- IsEpimorphism
- IsEqualAsFactorobjects
- IsEqualAsSubobjects
- IsEqualForCacheForMorphisms
- IsEqualForCacheForObjects
- IsEqualForMorphisms
- IsEqualForMorphismsOnMor
- IsEqualForObjects
- IsIdempotent
- IsIdenticalToIdentityMorphism
- IsIdenticalToZeroMorphism
- IsInitial
- IsInjective

- IsIsomorphism
- IsMonomorphism
- IsOne
- IsProjective
- IsSplitEpimorphism
- IsSplitMonomorphism
- IsTerminal
- IsWellDefinedForMorphisms
- IsWellDefinedForObjects
- IsWellDefinedForTwoCells
- IsZeroForMorphisms
- IsZeroForObjects
- IsomorphismFromCoimageToCokernelOfKernel
- IsomorphismFromCokernelOfDiagonalDifferenceToPushout
- IsomorphismFromCokernelOfKernelToCoimage
- IsomorphismFromCoproductToDirectSum
- IsomorphismFromDirectProductToDirectSum
- IsomorphismFromDirectSumToCoproduct
- IsomorphismFromDirectSumToDirectProduct
- IsomorphismFromDualToInternalHom
- IsomorphismFromFiberProductToKernelOfDiagonalDifference
- IsomorphismFromImageObjectToKernelOfCokernel
- IsomorphismFromInitialObjectToZeroObject
- IsomorphismFromInternalHomToDual
- IsomorphismFromInternalHomToObjectWithGivenInternalHom
- IsomorphismFromInternalHomToTensorProduct
- IsomorphismFromKernelOfCokernelToImageObject
- IsomorphismFromKernelOfDiagonalDifferenceToFiberProduct
- IsomorphismFromObjectToInternalHomWithGivenInternalHom
- IsomorphismFromPushoutToCokernelOfDiagonalDifference
- IsomorphismFromTensorProductToInternalHom
- IsomorphismFromTerminalObjectToZeroObject
- IsomorphismFromZeroObjectToInitialObject
- IsomorphismFromZeroObjectToTerminalObject
- KernelEmbedding
- KernelEmbeddingWithGivenKernelObject
- KernelLift
- KernelLiftWithGivenKernelObject
- KernelObject
- KernelObjectFunctorialWithGivenKernelObjects
- LambdaElimination
- LambdaIntroduction
- LeftDistributivityExpandingWithGivenObjects

- LeftDistributivityFactoringWithGivenObjects
- LeftUnitorInverseWithGivenTensorProduct
- LeftUnitorWithGivenTensorProduct
- Lift
- LiftAlongMonomorphism
- MonoidalPostComposeMorphismWithGivenObjects
- MonoidalPreComposeMorphismWithGivenObjects
- MorphismFromBidualWithGivenBidual
- MorphismFromCoimageToImageWithGivenObjects
- MorphismFromInternalHomToTensorProductWithGivenObjects
- MorphismFromTensorProductToInternalHomWithGivenObjects
- MorphismToBidualWithGivenBidual
- PostCompose
- PreCompose
- ProjectionInFactorOfDirectProduct
- ProjectionInFactorOfDirectProductWithGivenDirectProduct
- ProjectionInFactorOfDirectSum
- ProjectionInFactorOfDirectSumWithGivenDirectSum
- ProjectionInFactorOfFiberProduct
- ProjectionInFactorOfFiberProductWithGivenFiberProduct
- Pushout
- PushoutFunctorialWithGivenPushouts
- RankMorphism
- RightDistributivityExpandingWithGivenObjects
- RightDistributivityFactoringWithGivenObjects
- RightUnitorInverseWithGivenTensorProduct
- RightUnitorWithGivenTensorProduct
- TensorProductDualityCompatibilityMorphismWithGivenObjects
- TensorProductInternalHomCompatibilityMorphismInverseWithGivenObjects
- TensorProductInternalHomCompatibilityMorphismWithGivenObjects
- TensorProductOnMorphismsWithGivenTensorProducts
- TensorProductOnObjects
- TensorProductToInternalHomAdjunctionMap
- TensorUnit
- TerminalObject
- TerminalObjectFunctorial
- TraceMap
- UniversalMorphismFromCoproduct
- UniversalMorphismFromCoproductWithGivenCoproduct
- UniversalMorphismFromDirectSum
- UniversalMorphismFromDirectSumWithGivenDirectSum
- UniversalMorphismFromImage
- UniversalMorphismFromImageWithGivenImageObject

- UniversalMorphismFromInitialObject
- UniversalMorphismFromInitialObjectWithGivenInitialObject
- UniversalMorphismFromPushout
- UniversalMorphismFromPushoutWithGivenPushout
- UniversalMorphismFromZeroObject
- UniversalMorphismFromZeroObjectWithGivenZeroObject
- UniversalMorphismIntoCoimage
- UniversalMorphismIntoCoimageWithGivenCoimage
- UniversalMorphismIntoDirectProduct
- UniversalMorphismIntoDirectProductWithGivenDirectProduct
- UniversalMorphismIntoDirectSum
- UniversalMorphismIntoDirectSumWithGivenDirectSum
- UniversalMorphismIntoFiberProduct
- UniversalMorphismIntoFiberProductWithGivenFiberProduct
- UniversalMorphismIntoTerminalObject
- UniversalMorphismIntoTerminalObjectWithGivenTerminalObject
- UniversalMorphismIntoZeroObject
- UniversalMorphismIntoZeroObjectWithGivenZeroObject
- UniversalPropertyOfDual
- VerticalPostCompose
- VerticalPreCompose
- ZeroMorphism
- ZeroObject

# APPENDIX D

# Derivations

## Derivation index

- AdditionForMorphisms
- AssociatorLeftToRightWithGivenTensorProducts
- AssociatorRightToLeftWithGivenTensorProducts
- AstrictionToCoimage
- AstrictionToCoimageWithGivenCoimage
- BraidingInverseWithGivenTensorProducts
- BraidingWithGivenTensorProducts
- CoastrictionToImage
- CoastrictionToImageWithGivenImageObject
- CoevaluationForDualWithGivenTensorProduct
- CoevaluationMorphismWithGivenRange
- Coimage
- CoimageProjection
- CokernelColift
- CokernelColiftWithGivenCokernelObject
- CokernelFunctorialWithGivenCokernelObjects
- CokernelObject
- ColiftAlongEpimorphism
- Coproduct
- CoproductFunctorialWithGivenCoproducts
- DirectProduct
- DirectProductFunctorialWithGivenDirectProducts
- DirectSumCodiagonalDifference
- DirectSumDiagonalDifference
- DirectSumFunctorialWithGivenDirectSums
- DirectSumProjectionInPushout
- DualOnMorphismsWithGivenDuals
- DualOnObjects
- EvaluationForDualWithGivenTensorProduct
- EvaluationMorphismWithGivenSource
- FiberProduct
- FiberProductEmbeddingInDirectSum
- FiberProductFunctorialWithGivenFiberProducts

- HorizontalPostCompose
- HorizontalPreCompose
- ImageEmbedding
- ImageObject
- InitialObject
- InitialObjectFunctorial
- InjectionOfCofactorOfCoproduct
- InjectionOfCofactorOfDirectSum
- InjectionOfCofactorOfPushout
- InternalHomOnMorphismsWithGivenInternalHoms
- InternalHomOnObjects
- InternalHomToTensorProductAdjunctionMap
- InverseImmutable
- InverseMorphismFromCoimageToImageWithGivenObjects
- IsAutomorphism
- IsCodominating
- IsDominating
- IsEndomorphism
- IsEpimorphism
- IsEqualAsFactorobjects
- IsEqualAsSubobjects
- IsEqualForCacheForObjects
- IsEqualForMorphismsOnMor
- IsIdempotent
- IsIdenticalToIdentityMorphism
- IsIdenticalToZeroMorphism
- IsInitial
- IsIsomorphism
- IsMonomorphism
- IsOne
- IsTerminal
- IsZeroForMorphisms
- IsZeroForObjects
- IsomorphismFromCoimageToCokernelOfKernel
- IsomorphismFromCokernelOfDiagonalDifferenceToPushout
- IsomorphismFromCokernelOfKernelToCoimage
- IsomorphismFromCoproductToDirectSum
- IsomorphismFromDirectProductToDirectSum
- IsomorphismFromDirectSumToCoproduct
- IsomorphismFromDirectSumToDirectProduct
- IsomorphismFromFiberProductToKernelOfDiagonalDifference
- IsomorphismFromImageObjectToKernelOfCokernel
- IsomorphismFromInitialObjectToZeroObject

- IsomorphismFromInternalHomToObjectWithGivenInternalHom
- IsomorphismFromInternalHomToTensorProduct
- IsomorphismFromKernelOfCokernelToImageObject
- IsomorphismFromKernelOfDiagonalDifferenceToFiberProduct
- IsomorphismFromObjectToInternalHomWithGivenInternalHom
- IsomorphismFromPushoutToCokernelOfDiagonalDifference
- IsomorphismFromTensorProductToInternalHom
- IsomorphismFromTerminalObjectToZeroObject
- IsomorphismFromZeroObjectToInitialObject
- IsomorphismFromZeroObjectToTerminalObject
- KernelLift
- KernelLiftWithGivenKernelObject
- KernelObject
- KernelObjectFunctorialWithGivenKernelObjects
- LambdaElimination
- LambdaIntroduction
- LeftDistributivityExpandingWithGivenObjects
- LeftDistributivityFactoringWithGivenObjects
- LeftUnitorInverseWithGivenTensorProduct
- LeftUnitorWithGivenTensorProduct
- LiftAlongMonomorphism
- MonoidalPostComposeMorphismWithGivenObjects
- MonoidalPreComposeMorphismWithGivenObjects
- MorphismFromBidualWithGivenBidual
- MorphismFromCoimageToImageWithGivenObjects
- MorphismFromInternalHomToTensorProductWithGivenObjects
- MorphismFromTensorProductToInternalHomWithGivenObjects
- MorphismToBidualWithGivenBidual
- PostCompose
- PreCompose
- ProjectionInFactorOfDirectProduct
- ProjectionInFactorOfDirectSum
- ProjectionInFactorOfFiberProduct
- Pushout
- PushoutFunctorialWithGivenPushouts
- RankMorphism
- RightDistributivityExpandingWithGivenObjects
- RightDistributivityFactoringWithGivenObjects
- RightUnitorInverseWithGivenTensorProduct
- RightUnitorWithGivenTensorProduct
- TensorProductDualityCompatibilityMorphismWithGivenObjects
- TensorProductInternalHomCompatibilityMorphismInverseWithGivenObjects
- TensorProductInternalHomCompatibilityMorphismWithGivenObjects

- TensorProductToInternalHomAdjunctionMap
- TerminalObject
- TerminalObjectFunctorial
- TraceMap
- UniversalMorphismFromCoproduct
- UniversalMorphismFromDirectSum
- UniversalMorphismFromDirectSumWithGivenDirectSum
- UniversalMorphismFromImage
- UniversalMorphismFromImageWithGivenImageObject
- UniversalMorphismFromInitialObject
- UniversalMorphismFromInitialObjectWithGivenInitialObject
- UniversalMorphismFromPushout
- UniversalMorphismFromZeroObject
- UniversalMorphismIntoCoimage
- UniversalMorphismIntoCoimageWithGivenCoimage
- UniversalMorphismIntoDirectProduct
- UniversalMorphismIntoDirectSum
- UniversalMorphismIntoDirectSumWithGivenDirectSum
- UniversalMorphismIntoFiberProduct
- UniversalMorphismIntoTerminalObject
- UniversalMorphismIntoTerminalObjectWithGivenTerminalObject
- UniversalMorphismIntoZeroObject
- UniversalMorphismIntoZeroObjectWithGivenZeroObject
- UniversalPropertyOfDual
- VerticalPostCompose
- VerticalPreCompose
- ZeroMorphism

## Derivations for AdditionForMorphisms

**AdditionForMorphisms(mor1, mor2) as the composition of (mor1,mor2) with the codiagonal morphism**

This derivation is for additive categories. This derivation uses:

- UniversalMorphismIntoDirectSum $\times 1$
- IdentityMorphism $\times 1$
- UniversalMorphismFromDirectSum $\times 1$
- PreCompose $\times 1$

```
function ( mor1, mor2 )
    local  return_value, B, identity_morphism_B,
    componentwise_morphism, addition_morphism;
    B := Range( mor1 );
    componentwise_morphism := UniversalMorphismIntoDirectSum( mor1,
       mor2 );
    identity_morphism_B := IdentityMorphism( B );
    addition_morphism := UniversalMorphismFromDirectSum(
       identity_morphism_B, identity_morphism_B );
    return PreCompose( componentwise_morphism, addition_morphism );
end;
```

Back to index

### Derivations for AssociatorLeftToRightWithGivenTensorProducts

#### AssociatorLeftToRightWithGivenTensorProducts as the identity morphism

This derivation is for all categories. This derivation uses:

- IdentityMorphism × 1

```
function ( left_associated_object, object_1, object_2, object_3,
    right_associated_object )
    return IdentityMorphism( left_associated_object );
end;
```

Back to index

#### AssociatorLeftToRightWithGivenTensorProducts as the inverse of AssociatorRightToLeftWithGivenTensorProducts

This derivation is for all categories. This derivation uses:

- AssociatorRightToLeftWithGivenTensorProducts × 1

```
function ( left_associated_object, object_1, object_2, object_3,
    right_associated_object )
    return
     Inverse( AssociatorRightToLeftWithGivenTensorProducts(
         right_associated_object, object_1, object_2, object_3,
         left_associated_object ) );
end;
```

Back to index

### Derivations for AssociatorRightToLeftWithGivenTensorProducts

#### AssociatorRightToLeft as the identity morphism

This derivation is for all categories. This derivation uses:

- IdentityMorphism × 1

```
function ( right_associated_object, object_1, object_2, object_3,
    left_associated_object )
    return IdentityMorphism( right_associated_object );
end;
```

Back to index

**AssociatorRightToLeftWithGivenTensorProducts as the inverse of AssociatorLeftToRightWithGivenTensorProducts**

This derivation is for all categories. This derivation uses:

- AssociatorLeftToRightWithGivenTensorProducts $\times 1$

```
function ( right_associated_object, object_1, object_2, object_3,
    left_associated_object )
    return
     Inverse( AssociatorLeftToRightWithGivenTensorProducts(
         left_associated_object, object_1, object_2, object_3,
         right_associated_object ) );
end;
```

Back to index

## Derivations for AstrictionToCoimage

**AstrictionToCoimage using that coimage projection can be seen as a cokernel**

This derivation is for all categories. This derivation uses:

- ColiftAlongEpimorphism $\times 1$
- CoimageProjectionWithGivenCoimage $\times 1$
- CoimageProjection $\times 1$

```
function ( morphism )
    local  coimage_projection;
    coimage_projection := CoimageProjection( morphism );
    return ColiftAlongEpimorphism( coimage_projection, morphism );
end;
```

Back to index

## Derivations for AstrictionToCoimageWithGivenCoimage

**AstrictionToCoimage using that coimage projection can be seen as a cokernel**

This derivation is for all categories. This derivation uses:

- ColiftAlongEpimorphism $\times 1$
- CoimageProjectionWithGivenCoimage $\times 1$
- CoimageProjection $\times 1$

```
function ( morphism, coimage )
    local  coimage_projection;
    coimage_projection := CoimageProjectionWithGivenCoimage(
       morphism, coimage );
    return ColiftAlongEpimorphism( coimage_projection, morphism );
end;
```

Back to index

## Derivations for BraidingInverseWithGivenTensorProducts

### BraidingInverseWithGivenTensorProducts using BraidingWithGivenTensorProducts

This derivation is for symmetric monoidal categories. This derivation uses:

- BraidingWithGivenTensorProducts $\times 1$

```
function ( object_2_tensored_object_1, object_1, object_2,
    object_1_tensored_object_2 )
    return BraidingWithGivenTensorProducts(
       object_2_tensored_object_1, object_2, object_1,
       object_1_tensored_object_2 );
end;
```

Back to index

### BraidingInverseWithGivenTensorProducts as the inverse of the braiding

This derivation is for braided monoidal categories. This derivation uses:

- BraidingWithGivenTensorProducts $\times 1$
- TensorProductOnObjects $\times 2$

```
function ( object_2_tensored_object_1, object_1, object_2,
    object_1_tensored_object_2 )
    return Inverse( Braiding( object_1, object_2 ) );
end;
```

Back to index

## Derivations for BraidingWithGivenTensorProducts

### BraidingWithGivenTensorProducts using BraidingInverseWithGivenTensorProducts

This derivation is for symmetric monoidal categories. This derivation uses:

- BraidingInverseWithGivenTensorProducts $\times 1$

```
function ( object_1_tensored_object_2, object_1, object_2,
    object_2_tensored_object_1 )
    return BraidingInverseWithGivenTensorProducts(
       object_1_tensored_object_2, object_2, object_1,
       object_2_tensored_object_1 );
end;
```

Back to index

**BraidingWithGivenTensorProducts as the inverse of BraidingInverse**

This derivation is for braided monoidal categories. This derivation uses:

- BraidingInverseWithGivenTensorProducts × 1
- TensorProductOnObjects × 2

```
function ( object_1_tensored_object_2, object_1, object_2,
    object_2_tensored_object_1 )
    return Inverse( BraidingInverse( object_1, object_2 ) );
end;
```

Back to index

## Derivations for CoastrictionToImage

**CoastrictionToImage using that image embedding can be seen as a kernel**

This derivation is for all categories. This derivation uses:

- LiftAlongMonomorphism × 1
- ImageEmbeddingWithGivenImageObject × 1
- ImageEmbedding × 1

```
function ( morphism )
    local  image_embedding;
    image_embedding := ImageEmbedding( morphism );
    return LiftAlongMonomorphism( image_embedding, morphism );
end;
```

Back to index

## Derivations for CoastrictionToImageWithGivenImageObject

**CoastrictionToImage using that image embedding can be seen as a kernel**

This derivation is for all categories. This derivation uses:

- LiftAlongMonomorphism × 1
- ImageEmbeddingWithGivenImageObject × 1
- ImageEmbedding × 1

```
function ( morphism, image )
    local  image_embedding;
    image_embedding := ImageEmbeddingWithGivenImageObject(
       morphism, image );
    return LiftAlongMonomorphism( image_embedding, morphism );
end;
```

Back to index

## Derivations for CoevaluationForDualWithGivenTensorProduct

**CoevaluationForDualWithGivenTensorProduct using LambdaIntroduction on the identity and IsomorphismFromInternalHomToTensorProduct**

This derivation is for rigid symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism $\times 1$
- DualOnObjects $\times 1$
- LambdaIntroduction $\times 1$
- PreCompose $\times 2$
- IsomorphismFromInternalHomToTensorProduct $\times 1$
- BraidingWithGivenTensorProducts $\times 1$
- TensorProductOnObjects $\times 2$

```
function ( unit, object, tensor_object )
    local  morphism;
    morphism := IdentityMorphism( object );
    morphism := LambdaIntroduction( morphism );
    morphism
     := PreCompose( morphism,
       IsomorphismFromInternalHomToTensorProduct( object, object ) );
    morphism
     := PreCompose( morphism, Braiding( DualOnObjects( object ),
         object ) );
    return morphism;
end;
```

Back to index

## Derivations for CoevaluationMorphismWithGivenRange

**CoevaluationMorphismWithGivenRange using the rigidity of the monoidal category**

This derivation is for rigid symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism $\times 2$
- DualOnObjects $\times 1$
- PreCompose $\times 5$
- TensorProductOnObjects $\times 6$
- IsomorphismFromTensorProductToInternalHom $\times 1$
- LeftUnitorInverseWithGivenTensorProduct $\times 1$
- TensorUnit $\times 1$
- CoevaluationForDualWithGivenTensorProduct $\times 1$
- TensorProductOnMorphismsWithGivenTensorProducts $\times 3$
- AssociatorLeftToRightWithGivenTensorProducts $\times 1$
- BraidingWithGivenTensorProducts $\times 2$

```
function ( object_1, object_2, internal_hom )
    local  morphism, dual_2, id_1;
    dual_2 := DualOnObjects( object_2 );
    id_1 := IdentityMorphism( object_1 );
    morphism := LeftUnitorInverse( object_1 );
    morphism
     := PreCompose( morphism,
       TensorProductOnMorphisms( CoevaluationForDual( object_2 ),
         id_1 ) );
    morphism
     := PreCompose( morphism,
       TensorProductOnMorphisms( Braiding( object_2, dual_2 ), id_1
         ) );
    morphism
     := PreCompose( morphism,
       AssociatorLeftToRight( dual_2, object_2, object_1 ) );
    morphism
     := PreCompose( morphism,
       TensorProductOnMorphisms( IdentityMorphism( dual_2 ),
         Braiding( object_2, object_1 ) ) );
    morphism
     := PreCompose( morphism,
       IsomorphismFromTensorProductToInternalHom( object_2,
         TensorProductOnObjects( object_1, object_2 ) ) );
    return morphism;
end;
```

Back to index

**CoevaluationMorphismWithGivenRange using the rigidity of the monoidal category**

This derivation is for rigid symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism $\times$ 2
- DualOnObjects $\times$ 1
- PreCompose $\times$ 3
- TensorProductOnObjects $\times$ 6
- IsomorphismFromTensorProductToInternalHom $\times$ 1
- CoevaluationForDualWithGivenTensorProduct $\times$ 1
- TensorUnit $\times$ 1
- TensorProductOnMorphismsWithGivenTensorProducts $\times$ 3
- BraidingWithGivenTensorProducts $\times$ 2

```
function ( object_1, object_2, internal_hom )
    local  morphism, dual_2, id_1;
    dual_2 := DualOnObjects( object_2 );
    id_1 := IdentityMorphism( object_1 );
    morphism
     := TensorProductOnMorphisms( CoevaluationForDual( object_2 ),
       id_1 );
    morphism
     := PreCompose( morphism,
       TensorProductOnMorphisms( Braiding( object_2, dual_2 ), id_1
         ) );
    morphism
     := PreCompose( morphism,
       TensorProductOnMorphisms( IdentityMorphism( dual_2 ),
         Braiding( object_2, object_1 ) ) );
    morphism
     := PreCompose( morphism,
       IsomorphismFromTensorProductToInternalHom( object_2,
         TensorProductOnObjects( object_1, object_2 ) ) );
    return morphism;
end;
```

Back to index

**CoevaluationMorphismWithGivenRange using the tensor hom adjunction on the identity**

This derivation is for symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism $\times 1$
- TensorProductOnObjects $\times 1$
- TensorProductToInternalHomAdjunctionMap $\times 1$

```
function ( object_1, object_2, internal_hom )
    return TensorProductToInternalHomAdjunctionMap( object_1,
       object_2,
       IdentityMorphism( TensorProductOnObjects( object_1, object_2
           ) ) );
end;
```

Back to index

## Derivations for Coimage

**Coimage as the range of CoimageProjection**

This derivation is for all categories. This derivation uses:

- CoimageProjection × 1

```
function ( morphism )
    return Range( CoimageProjection( morphism ) );
end;
```

Back to index

**Coimage as the range of IsomorphismFromCokernelOfKernelToCoimage**
This derivation is for all categories. This derivation uses:

- IsomorphismFromCokernelOfKernelToCoimage × 1

```
function ( morphism )
    return
     Range( IsomorphismFromCokernelOfKernelToCoimage( morphism ) );
end;
```

Back to index

**Coimage as the source of IsomorphismFromCoimageToCokernelOfKernel**
This derivation is for all categories. This derivation uses:

- IsomorphismFromCoimageToCokernelOfKernel × 1

```
function ( morphism )
    return
     Source( IsomorphismFromCoimageToCokernelOfKernel( morphism ) );
end;
```

Back to index

## Derivations for CoimageProjection

**CoimageProjection as the cokernel projection of the kernel embedding**
This derivation is for Abelian categories. This derivation uses:

- AdditionForMorphisms × 1
- AdditiveInverseForMorphisms × 1
- KernelEmbedding × 1
- CokernelProjection × 1
- IsomorphismFromCokernelOfKernelToCoimage × 1
- PreCompose × 1

```
function ( mor )
    local  coimage_projection;
    coimage_projection
     := CokernelProjection( KernelEmbedding( mor ) );
    return PreCompose( coimage_projection,
       IsomorphismFromCokernelOfKernelToCoimage( mor ) );
end;
```

Back to index

## Derivations for CokernelColift

### CokernelColift using ColiftAlongEpimorphism and CokernelProjection

This derivation is for all categories. This derivation uses:

- ColiftAlongEpimorphism $\times 1$
- CokernelProjectionWithGivenCokernelObject $\times 1$
- CokernelProjection $\times 1$

```
function ( mor, test_morphism )
    return ColiftAlongEpimorphism( CokernelProjection( mor ),
       test_morphism );
end;
```

Back to index

## Derivations for CokernelColiftWithGivenCokernelObject

### CokernelColift using ColiftAlongEpimorphism and CokernelProjection

This derivation is for all categories. This derivation uses:

- ColiftAlongEpimorphism $\times 1$
- CokernelProjectionWithGivenCokernelObject $\times 1$
- CokernelProjection $\times 1$

```
function ( mor, test_morphism, cokernel )
    return
     ColiftAlongEpimorphism(
       CokernelProjectionWithGivenCokernelObject( mor, cokernel ),
       test_morphism );
end;
```

Back to index

## Derivations for CokernelFunctorialWithGivenCokernelObjects

### CokernelFunctorialWithGivenCokernelObjects using the universality of the cokernel

This derivation is for all categories. This derivation uses:

- CokernelColift $\times 1$
- PreCompose $\times 1$
- CokernelProjection $\times 1$

```
function ( cokernel_alpha, alpha, nu, alpha_p, cokernel_alpha_p )
    return
     CokernelColift( alpha,
       PreCompose( nu, CokernelProjection( alpha_p ) ) );
end;
```

Back to index

## Derivations for CokernelObject

### CokernelObject as the range of CokernelProjection

This derivation is for all categories. This derivation uses:

- CokernelProjection $\times 1$

```
function ( mor )
    return Range( CokernelProjection( mor ) );
end;
```

Back to index

## Derivations for ColiftAlongEpimorphism

### ColiftAlongEpimorphism using Colift

This derivation is for all categories. This derivation uses:

- Colift $\times 1$

```
function ( alpha, beta )
    return Colift( alpha, beta );
end;
```

Back to index

This derivation is for all categories. This derivation uses:

- KernelEmbedding $\times 1$
- CokernelColift $\times 2$
- PreCompose $\times 1$
- InverseImmutable $\times 1$

```
function ( epimorphism, test_morphism )
    local  kernel_emb, cokernel_colift_to_range_of_epimorphism,
    cokernel_colift_to_range_of_test_morphism, inverse;
    kernel_emb := KernelEmbedding( epimorphism );
    cokernel_colift_to_range_of_epimorphism
     := CokernelColift( kernel_emb, epimorphism );
    cokernel_colift_to_range_of_test_morphism
     := CokernelColift( kernel_emb, test_morphism );
    return
     PreCompose( Inverse( cokernel_colift_to_range_of_epimorphism )
        , cokernel_colift_to_range_of_test_morphism );
end;
```

Back to index

## Derivations for Coproduct

### Coproduct as the range of the first injection

This derivation is for all categories. This derivation uses:

- InjectionOfCofactorOfCoproduct $\times 1$

```
function ( object_product_list )
    return
     Range( InjectionOfCofactorOfCoproduct( object_product_list, 1
         ) );
end;
```

Back to index

### Coproduct as the range of IsomorphismFromDirectSumToCoproduct

This derivation is for all categories. This derivation uses:

- IsomorphismFromDirectSumToCoproduct $\times 1$

```
function ( object_product_list )
    return
     Range( IsomorphismFromDirectSumToCoproduct(
         object_product_list ) );
end;
```

Back to index

## Derivations for CoproductFunctorialWithGivenCoproducts

### CoproductFunctorialWithGivenCoproducts using the universality of the coproduct

This derivation is for all categories. This derivation uses:

- PreCompose $\times 2$
- InjectionOfCofactorOfCoproduct $\times 2$
- UniversalMorphismFromCoproduct $\times 1$

```
function ( coproduct_source, morphism_list, coproduct_range )
    local  coproduct_diagram, sink, diagram;
    coproduct_diagram := List( morphism_list, function ( mor )
            return Range( mor );
        end );
    sink := List( [ 1 .. Length( morphism_list ) ], function ( i )
            return
             PreCompose( morphism_list[i],
               InjectionOfCofactorOfCoproduct( coproduct_diagram, i
                 ) );
        end );
    diagram := List( morphism_list, function ( mor )
            return Source( mor );
        end );
    return UniversalMorphismFromCoproduct( diagram, sink );
end;
```

Back to index

## Derivations for DirectProduct

### DirectProduct as Source of ProjectionInFactorOfDirectProduct

This derivation is for all categories. This derivation uses:

- ProjectionInFactorOfDirectProduct $\times 1$

```
function ( object_product_list )
    return
     Source( ProjectionInFactorOfDirectProduct( object_product_list
          , 1 ) );
end;
```

Back to index

### DirectProduct as the source of IsomorphismFromDirectProductToDirectSum

This derivation is for all categories. This derivation uses:

- IsomorphismFromDirectProductToDirectSum $\times 1$

```
function ( object_product_list )
    return
     Source( IsomorphismFromDirectProductToDirectSum(
         object_product_list ) );
end;
```

Back to index

## Derivations for DirectProductFunctorialWithGivenDirectProducts

### DirectProductFunctorialWithGivenDirectProducts using universality of direct product

This derivation is for all categories. This derivation uses:

- PreCompose $\times 2$
- ProjectionInFactorOfDirectProduct $\times 2$
- UniversalMorphismIntoDirectProduct $\times 1$

```
function ( direct_product_source, morphism_list,
    direct_product_range )
    local  direct_product_diagram, source, diagram;
    direct_product_diagram := List( morphism_list, function ( mor )
            return Source( mor );
        end );
    source := List( [ 1 .. Length( morphism_list ) ], function ( i )
            return
             PreCompose( ProjectionInFactorOfDirectProduct(
                 direct_product_diagram, i ), morphism_list[i] );
        end );
    diagram := List( morphism_list, function ( mor )
            return Range( mor );
        end );
    return UniversalMorphismIntoDirectProduct( diagram, source );
end;
```

Back to index

## Derivations for DirectSumCodiagonalDifference

**DirectSumCodiagonalDifference using the operations defining this morphism**

This derivation is for all categories. This derivation uses:

- InjectionOfCofactorOfDirectSum $\times 2$
- PreCompose $\times 2$
- UniversalMorphismFromDirectSum $\times 2$
- AdditiveInverseForMorphisms $\times 2$
- AdditionForMorphisms $\times 2$
- UniversalMorphismFromZeroObject $\times 1$

```
function ( diagram )
    local  cobase, direct_sum_diagram, number_of_morphisms,
    list_of_morphisms, mor1, mor2;
    direct_sum_diagram := List( diagram, Range );
    number_of_morphisms := Length( diagram );
    list_of_morphisms := List( [ 1 .. number_of_morphisms ],
       function ( i )
            return
             PreCompose( diagram[i],
               InjectionOfCofactorOfDirectSum( direct_sum_diagram,
                 i ) );
        end );
    if number_of_morphisms = 1  then
        return UniversalMorphismFromZeroObject(
           Range( list_of_morphisms[1] ) );
    fi;
    mor1 := CallFuncList( UniversalMorphismFromDirectSum,
       list_of_morphisms{[ 1 .. number_of_morphisms - 1 ]} );
    mor2 := CallFuncList( UniversalMorphismFromDirectSum,
       list_of_morphisms{[ 2 .. number_of_morphisms ]} );
    return mor1 - mor2;
end;
```

Back to index

## Derivations for DirectSumDiagonalDifference

### DirectSumDiagonalDifference using the operations defining this morphism

This derivation is for all categories. This derivation uses:

- PreCompose $\times$ 2
- ProjectionInFactorOfDirectSum $\times$ 2
- UniversalMorphismIntoDirectSum $\times$ 2
- AdditiveInverseForMorphisms $\times$ 2
- AdditionForMorphisms $\times$ 2
- UniversalMorphismIntoZeroObject $\times$ 1

```
function ( diagram )
    local  direct_sum_diagram, number_of_morphisms,
    list_of_morphisms, mor1, mor2;
    direct_sum_diagram := List( diagram, Source );
    number_of_morphisms := Length( diagram );
    list_of_morphisms := List( [ 1 .. number_of_morphisms ],
       function ( i )
            return
             PreCompose(
               ProjectionInFactorOfDirectSum( direct_sum_diagram, i
                 ), diagram[i] );
        end );
    if number_of_morphisms = 1  then
        return UniversalMorphismIntoZeroObject(
           Source( list_of_morphisms[1] ) );
    fi;
    mor1 := CallFuncList( UniversalMorphismIntoDirectSum,
       list_of_morphisms{[ 1 .. number_of_morphisms - 1 ]} );
    mor2 := CallFuncList( UniversalMorphismIntoDirectSum,
       list_of_morphisms{[ 2 .. number_of_morphisms ]} );
    return mor1 - mor2;
end;
```

Back to index

### Derivations for DirectSumFunctorialWithGivenDirectSums

**DirectSumFunctorialWithGivenDirectSums using the universal morphism into direct sum**

This derivation is for additive categories. This derivation uses:

- PreCompose $\times 2$
- ProjectionInFactorOfDirectSum $\times 2$
- UniversalMorphismIntoDirectSum $\times 1$

```
function ( direct_sum_source, morphism_list, direct_sum_range )
    local  direct_sum_diagram, source, diagram;
    direct_sum_diagram := List( morphism_list, function ( mor )
            return Source( mor );
        end );
    source := List( [ 1 .. Length( morphism_list ) ], function ( i )
            return
             PreCompose(
               ProjectionInFactorOfDirectSum( direct_sum_diagram, i
                 ), morphism_list[i] );
        end );
    diagram := List( morphism_list, function ( mor )
            return Range( mor );
        end );
    return UniversalMorphismIntoDirectSum( diagram, source );
end;
```

Back to index

**DirectSumFunctorialWithGivenDirectSums using the universal morphism from direct sum**

This derivation is for additive categories. This derivation uses:

- PreCompose $\times 2$
- InjectionOfCofactorOfDirectSum $\times 2$
- UniversalMorphismFromDirectSum $\times 1$

```
function ( direct_sum_source, morphism_list, direct_sum_range )
    local  direct_sum_diagram, sink, diagram;
    direct_sum_diagram := List( morphism_list, function ( mor )
            return Range( mor );
        end );
    sink := List( [ 1 .. Length( morphism_list ) ], function ( i )
            return
             PreCompose( morphism_list[i],
               InjectionOfCofactorOfDirectSum( direct_sum_diagram,
                 i ) );
        end );
    diagram := List( morphism_list, function ( mor )
            return Source( mor );
        end );
    return UniversalMorphismFromDirectSum( diagram, sink );
end;
```

Back to index

## Derivations for DirectSumProjectionInPushout

**DirectSumProjectionInPushout as the cokernel projection of DirectSumCodiagonalDifference**

This derivation is for all categories. This derivation uses:

- CokernelProjection $\times 1$
- DirectSumCodiagonalDifference $\times 1$
- IsomorphismFromCokernelOfDiagonalDifferenceToPushout $\times 1$
- PreCompose $\times 1$

```
function ( diagram )
    local  cokernel_proj_of_diagonal_difference;
    cokernel_proj_of_diagonal_difference
     := CokernelProjection( DirectSumCodiagonalDifference( diagram
         ) );
    return PreCompose( cokernel_proj_of_diagonal_difference,
       IsomorphismFromCokernelOfDiagonalDifferenceToPushout(
         diagram ) );
end;
```

Back to index

**DirectSumProjectionInPushout using the universal property of the direct sum**

This derivation is for all categories. This derivation uses:

- UniversalMorphismFromDirectSum $\times 1$
- InjectionOfCofactorOfPushout $\times 2$

```
function ( diagram )
    local  ranges_of_diagram, test_sink;
    ranges_of_diagram := List( diagram, Range );
    test_sink := List( [ 1 .. Length( diagram ) ], function ( i )
            return InjectionOfCofactorOfPushout( diagram, i );
```

```
        end );
    return UniversalMorphismFromDirectSum( ranges_of_diagram,
       test_sink );
end;
```

Back to index

### Derivations for DualOnMorphismsWithGivenDuals

#### DualOnMorphismsWithGivenDuals using InternalHomOnMorphisms and IsomorphismFromDualToInternalHom

This derivation is for symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism $\times 1$
- TensorUnit $\times 1$
- IsomorphismFromDualToInternalHom $\times 1$
- IsomorphismFromInternalHomToDual $\times 1$
- PreCompose $\times 2$
- InternalHomOnMorphismsWithGivenInternalHoms $\times 1$
- InternalHomOnObjects $\times 2$

```
function ( new_source, morphism, new_range )
    local  category, result_morphism;
    category := CapCategory( morphism );
    result_morphism := InternalHomOnMorphisms( morphism,
       IdentityMorphism( TensorUnit( category ) ) );
    result_morphism
     :=
      PreCompose( IsomorphismFromDualToInternalHom(
         Range( morphism ) ), result_morphism );
    result_morphism := PreCompose( result_morphism,
       IsomorphismFromInternalHomToDual( Source( morphism ) ) );
    return result_morphism;
end;
```

Back to index

### Derivations for DualOnObjects

#### DualOnObjects as the source of IsomorphismFromDualToInternalHom

This derivation is for symmetric closed monoidal categories. This derivation uses:

- IsomorphismFromDualToInternalHom $\times 1$

```
function ( object )
    return Source( IsomorphismFromDualToInternalHom( object ) );
end;
```

Back to index

**DualOnObjects as the range of IsomorphismFromInternalHomToDual**

This derivation is for symmetric closed monoidal categories. This derivation uses:

- IsomorphismFromInternalHomToDual $\times 1$

```
function ( object )
    return Range( IsomorphismFromInternalHomToDual( object ) );
end;
```

Back to index

## Derivations for EvaluationForDualWithGivenTensorProduct

**EvaluationForDualWithGivenTensorProduct using the tensor hom adjunction and IsomorphismFromDualToInternalHom**

This derivation is for symmetric closed monoidal categories. This derivation uses:

- IsomorphismFromDualToInternalHom $\times 1$
- InternalHomToTensorProductAdjunctionMap $\times 1$

```
function ( tensor_object, object, unit )
    return InternalHomToTensorProductAdjunctionMap( object, unit,
       IsomorphismFromDualToInternalHom( object ) );
end;
```

Back to index

## Derivations for EvaluationMorphismWithGivenSource

**EvaluationMorphismWithGivenSource using the rigidity of the monoidal category**

This derivation is for rigid symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism $\times 3$
- DualOnObjects $\times 2$
- PreCompose $\times 4$
- IsomorphismFromInternalHomToTensorProduct $\times 1$
- RightUnitorWithGivenTensorProduct $\times 1$
- TensorProductOnObjects $\times 6$
- TensorUnit $\times 1$
- EvaluationForDualWithGivenTensorProduct $\times 1$
- TensorProductOnMorphismsWithGivenTensorProducts $\times 3$
- AssociatorLeftToRightWithGivenTensorProducts $\times 1$
- BraidingWithGivenTensorProducts $\times 1$

```
function ( object_1, object_2, internal_hom_tensored_object_1 )
    local  morphism;
    morphism
```

```
      := TensorProductOnMorphisms(
        IsomorphismFromInternalHomToTensorProduct( object_1,
          object_2 ), IdentityMorphism( object_1 ) );
    morphism
      := PreCompose( morphism,
        TensorProductOnMorphisms(
          Braiding( DualOnObjects( object_1 ), object_2 ),
          IdentityMorphism( object_1 ) ) );
    morphism
      := PreCompose( morphism, AssociatorLeftToRight( object_2,
          DualOnObjects( object_1 ), object_1 ) );
    morphism
      := PreCompose( morphism,
        TensorProductOnMorphisms( IdentityMorphism( object_2 ),
          EvaluationForDual( object_1 ) ) );
    morphism := PreCompose( morphism, RightUnitor( object_2 ) );
    return morphism;
end;
```

Back to index

**EvaluationMorphismWithGivenSource using the rigidity and strictness of the monoidal category**

This derivation is for rigid symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism × 3
- DualOnObjects × 1
- PreCompose × 2
- IsomorphismFromInternalHomToTensorProduct × 1
- EvaluationForDualWithGivenTensorProduct × 1
- TensorProductOnObjects × 6
- TensorUnit × 1
- TensorProductOnMorphismsWithGivenTensorProducts × 3
- BraidingWithGivenTensorProducts × 1

```
function ( object_1, object_2, internal_hom_tensored_object_1 )
    local  morphism;
    morphism
      := TensorProductOnMorphisms(
        IsomorphismFromInternalHomToTensorProduct( object_1,
          object_2 ), IdentityMorphism( object_1 ) );
    morphism
      := PreCompose( morphism,
        TensorProductOnMorphisms(
          Braiding( DualOnObjects( object_1 ), object_2 ),
        IdentityMorphism( object_1 ) ) );
    morphism
     := PreCompose( morphism,
       TensorProductOnMorphisms( IdentityMorphism( object_2 ),
         EvaluationForDual( object_1 ) ) );
    return morphism;
end;
```

Back to index

**EvaluationMorphismWithGivenSource using the tenor hom adjunction on the identity**

This derivation is for symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism $\times 1$
- InternalHomOnObjects $\times 1$
- InternalHomToTensorProductAdjunctionMap $\times 1$

```
function ( object_1, object_2, tensor_object )
    return InternalHomToTensorProductAdjunctionMap( object_1,
       object_2,
       IdentityMorphism( InternalHomOnObjects( object_1, object_2 )
         ) );
end;
```

Back to index

## Derivations for FiberProduct

**FiberProduct as the source of FiberProductEmbeddingInDirectSum**

This derivation is for all categories. This derivation uses:

- FiberProductEmbeddingInDirectSum $\times 1$

```
function ( diagram )
    return Source( FiberProductEmbeddingInDirectSum( diagram ) );
end;
```

Back to index

## Derivations for FiberProductEmbeddingInDirectSum

**FiberProductEmbeddingInDirectSum as the kernel embedding of DirectSumDiagonalDifference**

This derivation is for all categories. This derivation uses:

- KernelEmbedding $\times 1$
- DirectSumDiagonalDifference $\times 1$
- IsomorphismFromFiberProductToKernelOfDiagonalDifference $\times 1$
- PreCompose $\times 1$

```
function ( diagram )
    local  kernel_of_diagonal_difference;
    kernel_of_diagonal_difference
     := KernelEmbedding( DirectSumDiagonalDifference( diagram ) );
    return
     PreCompose(
       IsomorphismFromFiberProductToKernelOfDiagonalDifference(
         diagram ), kernel_of_diagonal_difference );
end;
```

Back to index

**FiberProductEmbeddingInDirectSum using the universal property of the direct sum**

This derivation is for all categories. This derivation uses:

- UniversalMorphismIntoDirectSum $\times 1$
- ProjectionInFactorOfFiberProduct $\times 2$

```
function ( diagram )
    local  sources_of_diagram, test_source;
    sources_of_diagram := List( diagram, Source );
    test_source := List( [ 1 .. Length( diagram ) ], function ( i )
            return ProjectionInFactorOfFiberProduct( diagram, i );
        end );
    return UniversalMorphismIntoDirectSum( sources_of_diagram,
       test_source );
end;
```

Back to index

## Derivations for FiberProductFunctorialWithGivenFiberProducts

**FiberProductFunctorialWithGivenFiberProducts using the universality of the fiber product**

This derivation is for all categories. This derivation uses:

- PreCompose $\times 2$
- ProjectionInFactorOfFiberProduct $\times 2$
- UniversalMorphismIntoFiberProduct $\times 1$

```
function ( fiber_product_source, morphism_of_morphisms,
    fiber_product_range )
    local  pullback_diagram, source, diagram;
    pullback_diagram := List( morphism_of_morphisms,
       function ( mor )
            return mor[1];
```

```
        end );
    source := List( [ 1 .. Length( morphism_of_morphisms ) ],
       function ( i )
            return
             PreCompose( ProjectionInFactorOfFiberProduct(
                 pullback_diagram, i ), morphism_of_morphisms[i][2]
               );
        end );
    diagram := List( morphism_of_morphisms, function ( mor )
            return mor[3];
        end );
    return UniversalMorphismIntoFiberProduct( diagram, source );
end;
```

Back to index

## Derivations for HorizontalPostCompose

### HorizontalPostCompose using HorizontalPreCompose

This derivation is for all categories. This derivation uses:

- HorizontalPreCompose $\times 1$

```
function ( twocell_right, twocell_left )
    return HorizontalPreCompose( twocell_left, twocell_right );
end;
```

Back to index

## Derivations for HorizontalPreCompose

### HorizontalPreCompose using HorizontalPostCompose

This derivation is for all categories. This derivation uses:

- HorizontalPostCompose $\times 1$

```
function ( twocell_left, twocell_right )
    return HorizontalPostCompose( twocell_right, twocell_left );
end;
```

Back to index

## Derivations for ImageEmbedding

### ImageEmbedding as the kernel embedding of the cokernel projection

This derivation is for Abelian categories. This derivation uses:

- AdditionForMorphisms $\times 1$
- AdditiveInverseForMorphisms $\times 1$
- KernelEmbedding $\times 1$

- CokernelProjection  × 1
- IsomorphismFromImageObjectToKernelOfCokernel  × 1
- PreCompose  × 1

```
function ( mor )
    local  image_embedding;
    image_embedding := KernelEmbedding( CokernelProjection( mor ) );
    return
     PreCompose( IsomorphismFromImageObjectToKernelOfCokernel( mor )
        , image_embedding );
end;
```

Back to index

## Derivations for ImageObject

### ImageObject as the source of ImageEmbedding

This derivation is for all categories. This derivation uses:

- ImageEmbedding  × 1

```
function ( mor )
    return Source( ImageEmbedding( mor ) );
end;
```

Back to index

### ImageObject as the source of IsomorphismFromImageObjectToKernelOfCokernel

This derivation is for all categories. This derivation uses:

- IsomorphismFromImageObjectToKernelOfCokernel  × 1

```
function ( morphism )
    return
     Source( IsomorphismFromImageObjectToKernelOfCokernel( morphism
         ) );
end;
```

Back to index

### ImageObject as the range of IsomorphismFromKernelOfCokernelToImageObject

This derivation is for all categories. This derivation uses:

- IsomorphismFromKernelOfCokernelToImageObject  × 1

```
function ( morphism )
    return
     Range( IsomorphismFromKernelOfCokernelToImageObject( morphism
         ) );
end;
```

Back to index

## Derivations for InitialObject

**InitialObject as the source of IsomorphismFromInitialObjectToZeroObject**
This derivation is for all categories. This derivation uses:

- IsomorphismFromInitialObjectToZeroObject $\times 1$

```
function ( category )
    return
     Source( IsomorphismFromInitialObjectToZeroObject( category ) );
end;
```

Back to index

**InitialObject as the range of IsomorphismFromZeroObjectToInitialObject**
This derivation is for all categories. This derivation uses:

- IsomorphismFromZeroObjectToInitialObject $\times 1$

```
function ( category )
    return
     Range( IsomorphismFromZeroObjectToInitialObject( category ) );
end;
```

Back to index

## Derivations for InitialObjectFunctorial

**InitialObjectFunctorial using the identity morphism of initial object**
This derivation is for all categories. This derivation uses:

- InitialObject $\times 1$
- IdentityMorphism $\times 1$

```
function ( category )
    local  initial_object;
    initial_object := InitialObject( category );
    return IdentityMorphism( initial_object );
end;
```

Back to index

**InitialObjectFunctorial using the universality of the initial object**
This derivation is for all categories. This derivation uses:

- InitialObject $\times 1$
- UniversalMorphismFromInitialObject $\times 1$

```
function ( category )
    local  initial_object;
    initial_object := InitialObject( category );
    return UniversalMorphismFromInitialObject( initial_object );
end;
```

Back to index

## Derivations for InjectionOfCofactorOfCoproduct

### InjectionOfCofactorOfCoproduct using InjectionOfCofactorOfDirectSum

This derivation is for all categories. This derivation uses:

- AdditionForMorphisms $\times 1$
- AdditiveInverseForMorphisms $\times 1$
- PreCompose $\times 1$
- InjectionOfCofactorOfDirectSum $\times 1$
- IsomorphismFromDirectSumToCoproduct $\times 1$

```
function ( diagram, injection_number )
    return
     PreCompose( InjectionOfCofactorOfDirectSum( diagram,
         injection_number ), IsomorphismFromDirectSumToCoproduct(
         diagram ) );
end;
```

Back to index

## Derivations for InjectionOfCofactorOfDirectSum

### InjectionOfCofactorOfDirectSum using InjectionOfCofactorOfCoproduct

This derivation is for all categories. This derivation uses:

- AdditionForMorphisms $\times 1$
- AdditiveInverseForMorphisms $\times 1$
- PreCompose $\times 1$
- InjectionOfCofactorOfCoproduct $\times 1$
- IsomorphismFromCoproductToDirectSum $\times 1$

```
function ( diagram, injection_number )
    return
     PreCompose( InjectionOfCofactorOfCoproduct( diagram,
         injection_number ), IsomorphismFromCoproductToDirectSum(
         diagram ) );
end;
```

Back to index

## Derivations for InjectionOfCofactorOfPushout

### InjectionOfCofactorOfPushout by composing the direct sum injection with the direct sum projection to the pushout

This derivation is for all categories. This derivation uses:

- AdditionForMorphisms × 1
- AdditiveInverseForMorphisms × 1
- PreCompose × 1
- InjectionOfCofactorOfDirectSum × 1
- DirectSumProjectionInPushout × 1

```
function ( diagram, injection_number )
    local  projection_from_direct_sum, direct_sum_diagram,
    injection;
    projection_from_direct_sum := DirectSumProjectionInPushout(
       diagram );
    direct_sum_diagram := List( diagram, Range );
    injection := InjectionOfCofactorOfDirectSum( direct_sum_diagram
        , injection_number );
    return PreCompose( injection, projection_from_direct_sum );
end;
```

Back to index

## Derivations for InternalHomOnMorphismsWithGivenInternalHoms

### InternalHomOnMorphismsWithGivenInternalHoms using functorality of Dual and TensorProduct

This derivation is for rigid symmetric closed monoidal categories. This derivation uses:

- PreCompose × 2
- IsomorphismFromTensorProductToInternalHom × 1
- IsomorphismFromInternalHomToTensorProduct × 1
- DualOnMorphismsWithGivenDuals × 1
- DualOnObjects × 2
- TensorProductOnMorphismsWithGivenTensorProducts × 1
- TensorProductOnObjects × 2

```
function ( internal_hom_source, morphism_1, morphism_2,
    internal_hom_range )
    local  dual_morphism;
    dual_morphism := DualOnMorphisms( morphism_1 );
    return
     PreCompose(
       PreCompose( IsomorphismFromInternalHomToTensorProduct(
           Range( morphism_1 ), Source( morphism_2 ) ),
         TensorProductOnMorphisms( dual_morphism, morphism_2 ) ),
       IsomorphismFromTensorProductToInternalHom(
         Source( morphism_1 ), Range( morphism_2 ) ) );
end;
```

Back to index

## Derivations for InternalHomOnObjects

### InternalHomOnObjects as the source of IsomorphismFromInternalHomToTensorProduct

This derivation is for rigid symmetric closed monoidal categories. This derivation uses:

- IsomorphismFromInternalHomToTensorProduct $\times 1$

```
function ( object_1, object_2 )
    return
     Source( IsomorphismFromInternalHomToTensorProduct( object_1,
         object_2 ) );
end;
```

Back to index

### InternalHomOnObjects as the range of IsomorphismFromTensorProductToInternalHom

This derivation is for symmetric closed monoidal categories. This derivation uses:

- IsomorphismFromTensorProductToInternalHom $\times 1$

```
function ( object_1, object_2 )
    return
     Range( IsomorphismFromTensorProductToInternalHom( object_1,
         object_2 ) );
end;
```

Back to index

## Derivations for InternalHomToTensorProductAdjunctionMap

### InternalHomToTensorProductAdjunctionMap using TensorProductOnMorphisms and EvaluationMorphism

This derivation is for symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism $\times 1$
- PreCompose $\times 1$
- TensorProductOnMorphismsWithGivenTensorProducts $\times 1$
- TensorProductOnObjects $\times 2$
- EvaluationMorphismWithGivenSource $\times 1$
- InternalHomOnObjects $\times 1$

```
function ( object_1, object_2, morphism )
    local  evaluation, tensor_product_on_morphisms;
    tensor_product_on_morphisms
     := TensorProductOnMorphisms( morphism,
       IdentityMorphism( object_1 ) );
    evaluation := EvaluationMorphism( object_1, object_2 );
    return PreCompose( tensor_product_on_morphisms, evaluation );
end;
```

Back to index

## Derivations for InverseImmutable

### Inverse using LiftAlongMonomorphism of an identity morphism

This derivation is for all categories. This derivation uses:

- IdentityMorphism $\times 1$
- LiftAlongMonomorphism $\times 1$

```
function ( mor )
    local  identity_of_range;
    identity_of_range := IdentityMorphism( Range( mor ) );
    return LiftAlongMonomorphism( mor, identity_of_range );
end;
```

Back to index

### Inverse using ColiftAlongEpimorphism of an identity morphism

This derivation is for all categories. This derivation uses:

- IdentityMorphism $\times 1$
- ColiftAlongEpimorphism $\times 1$

```
function ( mor )
    local  identity_of_source;
    identity_of_source := IdentityMorphism( Source( mor ) );
    return ColiftAlongEpimorphism( mor, identity_of_source );
end;
```

Back to index

## Derivations for InverseMorphismFromCoimageToImageWithGivenObjects

### InverseMorphismFromCoimageToImageWithGivenObjects as the inverse of MorphismFromCoimageToImage

This derivation is for Abelian categories.

```
function ( coimage, morphism, image )
    return Inverse( MorphismFromCoimageToImage( morphism ) );
end;
```

Back to index

## Derivations for IsAutomorphism

### IsAutomorphism by checking IsIsomorphism and IsEndomorphism

This derivation is for all categories. This derivation uses:

- IsIsomorphism × 1
- IsEndomorphism × 1

```
function ( morphism )
    return IsIsomorphism( morphism ) and IsEndomorphism( morphism );
end;
```

Back to index

## Derivations for IsCodominating

### IsCodominating using IsDominating and duality by kernel

This derivation is for all categories. This derivation uses:

- KernelEmbedding × 2
- IsDominating × 1

```
function ( factor1, factor2 )
    local  kernel_embedding_1, kernel_embedding_2;
    kernel_embedding_1 := KernelEmbedding( factor1 );
    kernel_embedding_2 := KernelEmbedding( factor2 );
    return IsDominating( kernel_embedding_2, kernel_embedding_1 );
end;
```

Back to index

### IsCodominating(factor1, factor2) by deciding if KernelEmbedding(factor2) composed with factor1 is zero

This derivation is for all categories. This derivation uses:

- KernelEmbedding × 1
- PreCompose × 1
- IsZeroForMorphisms × 1

```
function ( factor1, factor2 )
    local  kernel_embedding, composition;
    kernel_embedding := KernelEmbedding( factor2 );
    composition := PreCompose( kernel_embedding, factor1 );
    return IsZero( composition );
end;
```

Back to index

## Derivations for IsDominating

### IsDominating using IsCodominating and duality by cokernel

This derivation is for all categories. This derivation uses:

- CokernelProjection $\times 2$
- IsCodominating $\times 1$

```
function ( sub1, sub2 )
    local  cokernel_projection_1, cokernel_projection_2;
    cokernel_projection_1 := CokernelProjection( sub1 );
    cokernel_projection_2 := CokernelProjection( sub2 );
    return IsCodominating( cokernel_projection_1,
       cokernel_projection_2 );
end;
```

Back to index

### IsDominating(sub1, sub2) by deciding if sub1 composed with CokernelProjection(sub2) is zero

This derivation is for all categories. This derivation uses:

- CokernelProjection $\times 1$
- PreCompose $\times 1$
- IsZeroForMorphisms $\times 1$

```
function ( sub1, sub2 )
    local  cokernel_projection, composition;
    cokernel_projection := CokernelProjection( sub2 );
    composition := PreCompose( sub1, cokernel_projection );
    return IsZero( composition );
end;
```

Back to index

## Derivations for IsEndomorphism

### IsEndomorphism by deciding whether source and range are equal as objects

This derivation is for all categories. This derivation uses:

- IsEqualForObjects $\times 1$

```
function ( morphism )
    return IsEqualForObjects( Source( morphism ), Range( morphism )
      );
end;
```

Back to index

## Derivations for IsEpimorphism

**IsEpimorphism by deciding if the cokernel is a zero object**

This derivation is for additive categories. This derivation uses:

- IsZeroForObjects $\times 1$
- CokernelObject $\times 1$

```
function ( morphism )
    return IsZero( CokernelObject( morphism ) );
end;
```

Back to index

**IsEpimorphism by deciding if the codiagonal morphism is an isomorphism**

This derivation is for all categories. This derivation uses:

- IdentityMorphism $\times 1$
- UniversalMorphismFromPushout $\times 1$
- IsIsomorphism $\times 1$

```
function ( morphism )
    local  pushout_diagram, identity, codiagonal_morphism;
    pushout_diagram := [ morphism, morphism ];
    identity := IdentityMorphism( Range( morphism ) );
    codiagonal_morphism := UniversalMorphismFromPushout(
       pushout_diagram, identity, identity );
    return IsIsomorphism( codiagonal_morphism );
end;
```

Back to index

## Derivations for IsEqualAsFactorobjects

**IsEqualAsFactorobjects(factor1, factor2) if factor1 dominates factor2 and vice versa**

This derivation is for all categories. This derivation uses:

- IsCodominating $\times 2$

```
function ( factor1, factor2 )
    return IsCodominating( factor1, factor2 )
      and IsCodominating( factor1, factor2 );
end;
```

Back to index

## Derivations for IsEqualAsSubobjects

### IsEqualAsSubobjects(sub1, sub2) if sub1 dominates sub2 and vice versa

This derivation is for all categories. This derivation uses:

- IsDominating × 2

```
function ( sub1, sub2 );
    return IsDominating( sub1, sub2 ) and IsDominating( sub2, sub1 );
end;
```

Back to index

## Derivations for IsEqualForCacheForObjects

This derivation is for all categories. This derivation uses:

- IsEqualForObjects × 1

```
function ( object_1, object_2 )
    local  ret_value;
    return IsEqualForObjects( object_1, object_2 ) = true;
end;
```

Back to index

## Derivations for IsEqualForMorphismsOnMor

### IsEqualForMorphismsOnMor using IsEqualForMorphisms

This derivation is for all categories. This derivation uses:

- IsEqualForMorphisms × 1
- IsEqualForObjects × 2

```
function ( morphism_1, morphism_2 )
    local  value_1, value_2;
    value_1 := IsEqualForObjects( Source( morphism_1 ),
       Source( morphism_2 ) );
    if value_1 = fail  then
        return fail;
    fi;
    value_2 := IsEqualForObjects( Range( morphism_1 ),
       Range( morphism_2 ) );
    if value_2 = fail  then
        return fail;
    fi;
    if value_1 = false or value_2 = false  then
        return false;
    fi;
    return IsEqualForMorphisms( morphism_1, morphism_2 );
end;
```

Back to index

## Derivations for IsIdempotent

### IsIdempotent by comparing the square of the morphism with itself

This derivation is for all categories. This derivation uses:

- PreCompose $\times 1$
- IsCongruentForMorphisms $\times 1$

```
function ( morphism )
    return IsCongruentForMorphisms( PreCompose( morphism, morphism )
        , morphism );
end;
```

Back to index

## Derivations for IsIdenticalToIdentityMorphism

### IsIdenticalToIdentityMorphism using IsEqualForMorphismsOnMor and I-dentityMorphism

This derivation is for all categories. This derivation uses:

- IsEqualForMorphismsOnMor $\times 1$
- IdentityMorphism $\times 1$

```
function ( morphism )
    return IsEqualForMorphismsOnMor( morphism,
       IdentityMorphism( Source( morphism ) ) );
end;
```

Back to index

## Derivations for IsIdenticalToZeroMorphism

### IsIdenticalToZeroMorphism using IsEqualForMorphismsOnMor and Zero-Morphism

This derivation is for all categories. This derivation uses:

- ZeroMorphism $\times 1$
- IsEqualForMorphismsOnMor $\times 1$

```
function ( morphism )
    return IsEqualForMorphismsOnMor( morphism,
       ZeroMorphism( Source( morphism ), Range( morphism ) ) );
end;
```

Back to index

## Derivations for IsInitial

### IsInitial using IsZeroForObjects

This derivation is for additive categories. This derivation uses:

- IsZeroForObjects $\times 1$

```
function ( object )
    return IsZeroForObjects( object );
end;
```

Back to index

## Derivations for IsIsomorphism

### IsIsomorphism by deciding if it is a mono and an epi

This derivation is for Abelian categories. This derivation uses:

- IsMonomorphism $\times 1$
- IsEpimorphism $\times 1$

```
function ( morphism )
    return IsMonomorphism( morphism ) and IsEpimorphism( morphism );
end;
```

Back to index

## Derivations for IsMonomorphism

### IsMonomorphism by deciding if the kernel is a zero object

This derivation is for additive categories. This derivation uses:

- IsZeroForObjects $\times 1$
- KernelObject $\times 1$

```
function ( morphism )
    return IsZero( KernelObject( morphism ) );
end;
```

Back to index

### IsMonomorphism by deciding if the diagonal morphism is an isomorphism

This derivation is for all categories. This derivation uses:

- IsIsomorphism $\times 1$
- IdentityMorphism $\times 1$
- UniversalMorphismIntoFiberProduct $\times 1$

```
function ( morphism )
    local  pullback_diagram, pullback_projection_1,
    pullback_projection_2, identity, diagonal_morphism;
    pullback_diagram := [ morphism, morphism ];
    identity := IdentityMorphism( Source( morphism ) );
    diagonal_morphism := UniversalMorphismIntoFiberProduct(
       pullback_diagram, identity, identity );
    return IsIsomorphism( diagonal_morphism );
end;
```

Back to index

### Derivations for IsOne

#### IsOne by comparing with the identity morphism

This derivation is for all categories. This derivation uses:

- IdentityMorphism $\times 1$
- IsCongruentForMorphisms $\times 1$

```
function ( morphism )
    local  object;
    object := Source( morphism );
    return IsCongruentForMorphisms( IdentityMorphism( object ),
       morphism );
end;
```

Back to index

### Derivations for IsTerminal

#### IsTerminal using IsZeroForObjects

This derivation is for additive categories. This derivation uses:

- IsZeroForObjects $\times 1$

```
function ( object )
    return IsZeroForObjects( object );
end;
```

Back to index

### Derivations for IsZeroForMorphisms

#### IsZeroForMorphisms by deciding whether the given morphism is congruent to the zero morphism

This derivation is for all categories. This derivation uses:

- ZeroMorphism $\times 1$
- IsCongruentForMorphisms $\times 1$

```
function ( morphism )
    local  zero_morphism;
    zero_morphism := ZeroMorphism( Source( morphism ),
       Range( morphism ) );
    return IsCongruentForMorphisms( zero_morphism, morphism );
end;
```

Back to index

## Derivations for IsZeroForObjects

### IsZeroForObjects by comparing identity morphism with zero morphism

This derivation is for all categories. This derivation uses:

- IsCongruentForMorphisms $\times 1$
- IdentityMorphism $\times 1$
- ZeroMorphism $\times 1$

```
function ( object )
    return IsCongruentForMorphisms( IdentityMorphism( object ),
       ZeroMorphism( object, object ) );
end;
```

Back to index

## Derivations for IsomorphismFromCoimageToCokernelOfKernel

### IsomorphismFromCoimageToCokernelOfKernel as the inverse of IsomorphismFromCokernelOfKernelToCoimage

This derivation is for all categories. This derivation uses:

- IsomorphismFromCokernelOfKernelToCoimage $\times 1$

```
function ( morphism )
    return
     Inverse( IsomorphismFromCokernelOfKernelToCoimage( morphism ) );
end;
```

Back to index

## Derivations for IsomorphismFromCokernelOfDiagonalDifferenceToPushout

### IsomorphismFromCokernelOfDiagonalDifferenceToPushout using the universal property of the cokernel

This derivation is for all categories. This derivation uses:

- CokernelColift $\times 1$
- DirectSumCodiagonalDifference $\times 1$
- DirectSumProjectionInPushout $\times 1$

```
function ( diagram )
    local  direct_sum_codiagonal_difference,
    direct_sum_projection_in_pushout;
    direct_sum_codiagonal_difference
```

```
     := DirectSumCodiagonalDifference( diagram );
    direct_sum_projection_in_pushout
     := DirectSumProjectionInPushout( diagram );
    return CokernelColift( direct_sum_codiagonal_difference,
       direct_sum_projection_in_pushout );
end;
```

Back to index

**IsomorphismFromCokernelOfDiagonalDifferenceToPushout as the inverse of IsomorphismFromPushoutToCokernelOfDiagonalDifference**

This derivation is for all categories. This derivation uses:

- IsomorphismFromPushoutToCokernelOfDiagonalDifference $\times 1$
- InverseImmutable $\times 1$

```
function ( diagram )
    return
     Inverse( IsomorphismFromPushoutToCokernelOfDiagonalDifference(
         diagram ) );
end;
```

Back to index

## Derivations for IsomorphismFromCokernelOfKernelToCoimage

**IsomorphismFromCokernelOfKernelToCoimage as the inverse of IsomorphismFromCoimageToCokernelOfKernel**

This derivation is for all categories. This derivation uses:

- IsomorphismFromCoimageToCokernelOfKernel $\times 1$

```
function ( morphism )
    return
     Inverse( IsomorphismFromCoimageToCokernelOfKernel( morphism ) );
end;
```

Back to index

**IsomorphismFromCokernelOfKernelToCoimage using the universal property of the coimage**

This derivation is for all categories. This derivation uses:

- KernelEmbedding $\times 1$
- CokernelProjection $\times 1$
- ColiftAlongEpimorphism $\times 1$
- UniversalMorphismIntoCoimage $\times 1$

```
function ( morphism )
    local  cokernel_proj, morphism_from_cokernel;
    cokernel_proj
```

```
      := CokernelProjection( KernelEmbedding( morphism ) );
    morphism_from_cokernel
      := ColiftAlongEpimorphism( cokernel_proj, morphism );
    return UniversalMorphismIntoCoimage( morphism,
       [ cokernel_proj, morphism_from_cokernel ] );
end;
```

Back to index

## Derivations for IsomorphismFromCoproductToDirectSum

**IsomorphismFromCoproductToDirectSum using cofactor injections and the universal property of the coproduct**

This derivation is for all categories. This derivation uses:

- InjectionOfCofactorOfDirectSum × 2
- UniversalMorphismFromCoproduct × 1

```
function ( diagram )
    local  sink;
    sink := List( [ 1 .. Length( diagram ) ], function ( i )
            return InjectionOfCofactorOfDirectSum( diagram, i );
        end );
    return UniversalMorphismFromCoproduct( diagram, sink );
end;
```

Back to index

**IsomorphismFromCoproductToDirectSum as the inverse of IsomorphismFromDirectSumToCoproduct**

This derivation is for all categories. This derivation uses:

- InverseImmutable × 1
- IsomorphismFromDirectSumToCoproduct × 1

```
function ( diagram )
    return Inverse( IsomorphismFromDirectSumToCoproduct( diagram ) );
end;
```

Back to index

## Derivations for IsomorphismFromDirectProductToDirectSum

**IsomorphismFromDirectProductToDirectSum using direct product projections and universal property of direct sum**

This derivation is for all categories. This derivation uses:

- ProjectionInFactorOfDirectProduct × 2
- UniversalMorphismIntoDirectSum × 1

```
function ( diagram )
    local  source;
    source := List( [ 1 .. Length( diagram ) ], function ( i )
            return ProjectionInFactorOfDirectProduct( diagram, i );
        end );
    return UniversalMorphismIntoDirectSum( diagram, source );
end;
```

Back to index

**IsomorphismFromDirectProductToDirectSum as the inverse of IsomorphismFromDirectSumToDirectProduct**

This derivation is for all categories. This derivation uses:

- IsomorphismFromDirectSumToDirectProduct $\times 1$
- InverseImmutable $\times 1$

```
function ( diagram )
    return
     Inverse( IsomorphismFromDirectSumToDirectProduct( diagram ) );
end;
```

Back to index

## Derivations for IsomorphismFromDirectSumToCoproduct

**IsomorphismFromDirectSumToCoproduct using cofactor injections and the universal property of the direct sum**

This derivation is for all categories. This derivation uses:

- InjectionOfCofactorOfCoproduct $\times 2$
- UniversalMorphismFromDirectSum $\times 1$

```
function ( diagram )
    local  sink;
    sink := List( [ 1 .. Length( diagram ) ], function ( i )
            return InjectionOfCofactorOfCoproduct( diagram, i );
        end );
    return UniversalMorphismFromDirectSum( diagram, sink );
end;
```

Back to index

**IsomorphismFromDirectSumToCoproduct as the inverse of IsomorphismFromCoproductToDirectSum**

This derivation is for all categories. This derivation uses:

- InverseImmutable $\times 1$
- IsomorphismFromCoproductToDirectSum $\times 1$

```
function ( diagram )
    return Inverse( IsomorphismFromCoproductToDirectSum( diagram ) );
end;
```

Back to index

## Derivations for IsomorphismFromDirectSumToDirectProduct

### IsomorphismFromDirectSumToDirectProduct using direct sum projections and universal property of direct product

This derivation is for all categories. This derivation uses:

- ProjectionInFactorOfDirectSum × 2
- UniversalMorphismIntoDirectProduct × 1

```
function ( diagram )
    local  source;
    source := List( [ 1 .. Length( diagram ) ], function ( i )
            return ProjectionInFactorOfDirectSum( diagram, i );
        end );
    return UniversalMorphismIntoDirectProduct( diagram, source );
end;
```

Back to index

### IsomorphismFromDirectSumToDirectProduct as the inverse of IsomorphismFromDirectProductToDirectSum

This derivation is for all categories. This derivation uses:

- InverseImmutable × 1
- IsomorphismFromDirectProductToDirectSum × 1

```
function ( diagram );
    return
     Inverse( IsomorphismFromDirectProductToDirectSum( diagram ) );
end;
```

Back to index

## Derivations for IsomorphismFromFiberProductToKernelOfDiagonalDifference

### IsomorphismFromFiberProductToKernelOfDiagonalDifference as the inverse of IsomorphismFromKernelOfDiagonalDifferenceTo FiberProduct

This derivation is for all categories. This derivation uses:

- IsomorphismFromKernelOfDiagonalDifferenceToFiberProduct × 1
- InverseImmutable × 1

```
function ( diagram )
    return
     Inverse(
       IsomorphismFromKernelOfDiagonalDifferenceToFiberProduct(
         diagram ) );
end;
```

Back to index

**IsomorphismFromFiberProductToKernelOfDiagonalDifference using the universal property of the kernel**

This derivation is for all categories. This derivation uses:

- KernelLift × 1
- DirectSumDiagonalDifference × 1
- FiberProductEmbeddingInDirectSum × 1

```
function ( diagram )
    local  direct_sum_diagonal_difference,
    fiber_product_embedding_in_direct_sum;
    direct_sum_diagonal_difference := DirectSumDiagonalDifference(
       diagram );
    fiber_product_embedding_in_direct_sum
     := FiberProductEmbeddingInDirectSum( diagram );
    return KernelLift( direct_sum_diagonal_difference,
       fiber_product_embedding_in_direct_sum );
end;
```

Back to index

## Derivations for IsomorphismFromImageObjectToKernelOfCokernel

**IsomorphismFromImageObjectToKernelOfCokernel as the inverse of IsomorphismFromKernelOfCokernelToImageObject**

This derivation is for all categories. This derivation uses:

- IsomorphismFromKernelOfCokernelToImageObject × 1

```
function ( morphism )
    return
     Inverse( IsomorphismFromKernelOfCokernelToImageObject(
         morphism ) );
end;
```

Back to index

**IsomorphismFromImageObjectToKernelOfCokernel using the universal property of the image**

This derivation is for all categories. This derivation uses:

- KernelEmbedding × 1
- CokernelProjection × 1
- LiftAlongMonomorphism × 1
- UniversalMorphismFromImage × 1

```
function ( morphism )
    local  kernel_emb, morphism_to_kernel;
    kernel_emb := KernelEmbedding( CokernelProjection( morphism ) );
    morphism_to_kernel := LiftAlongMonomorphism( kernel_emb,
       morphism );
    return UniversalMorphismFromImage( morphism,
       [ morphism_to_kernel, kernel_emb ] );
end;
```

Back to index

## Derivations for IsomorphismFromInitialObjectToZeroObject

**IsomorphismFromInitialObjectToZeroObject as the unique morphism from initial object to zero object**

This derivation is for additive categories. This derivation uses:

- UniversalMorphismFromInitialObject × 1
- ZeroObject × 1

```
function ( category )
    return UniversalMorphismFromInitialObject(
       ZeroObject( category ) );
end;
```

Back to index

**IsomorphismFromInitialObjectToZeroObject using the universal property of the zero object**

This derivation is for all categories. This derivation uses:

- UniversalMorphismIntoZeroObject × 1
- InitialObject × 1

```
function ( category )
    return UniversalMorphismIntoZeroObject(
       InitialObject( category ) );
end;
```

Back to index

**IsomorphismFromInitialObjectToZeroObject as the inverse of Isomorphism-FromZeroObjectToInitialObject**

This derivation is for all categories. This derivation uses:

- InverseImmutable × 1
- IsomorphismFromZeroObjectToInitialObject × 1

```
function ( category )
    return
     Inverse( IsomorphismFromZeroObjectToInitialObject( category ) );
end;
```

Back to index

## Derivations for IsomorphismFromInternalHomToObjectWithGivenInternalHom

### IsomorphismFromInternalHomToObjectWithGivenInternalHom using the coevaluation morphism

This derivation is for symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism × 1
- TensorUnit × 1
- PreCompose × 1
- RightUnitorWithGivenTensorProduct × 1
- TensorProductOnObjects × 1
- InternalHomOnMorphismsWithGivenInternalHoms × 1
- InternalHomOnObjects × 2
- CoevaluationMorphismWithGivenRange × 1

```
function ( object, internal_hom )
    local  unit;
    unit := TensorUnit( CapCategory( object ) );
    return PreCompose( CoevaluationMorphism( object, unit ),
       InternalHomOnMorphisms( IdentityMorphism( unit ),
         RightUnitor( object ) ) );
end;
```

Back to index

## Derivations for IsomorphismFromInternalHomToTensorProduct

### IsomorphismFromInternalHomToTensorProduct using MorphismFromInternalHomToTensorProduct

This derivation is for rigid symmetric closed monoidal categories. This derivation uses:

- MorphismFromInternalHomToTensorProductWithGivenObjects × 1
- DualOnObjects × 1
- TensorProductOnObjects × 1
- InternalHomOnObjects × 1

```
function ( object_1, object_2 )
    return MorphismFromInternalHomToTensorProduct( object_1,
       object_2 );
end;
```

Back to index

## Derivations for IsomorphismFromKernelOfCokernelToImageObject

**IsomorphismFromKernelOfCokernelToImageObject as the inverse of IsomorphismFromImageObjectToKernelOfCokernel**

This derivation is for all categories. This derivation uses:

- IsomorphismFromImageObjectToKernelOfCokernel $\times 1$

```
function ( morphism )
    return
     Inverse( IsomorphismFromImageObjectToKernelOfCokernel(
         morphism ) );
end;
```

Back to index

## Derivations for IsomorphismFromKernelOfDiagonalDifferenceToFiberProduct

**IsomorphismFromKernelOfDiagonalDifferenceToFiberProduct using the universal property of the fiber product**

This derivation is for all categories. This derivation uses:

- KernelEmbedding $\times 1$
- PreCompose $\times 2$
- ProjectionInFactorOfDirectSum $\times 2$
- UniversalMorphismIntoFiberProduct $\times 1$
- DirectSumDiagonalDifference $\times 1$

```
function ( diagram )
    local  kernel_emb, sources_of_diagram, test_source;
    kernel_emb
     := KernelEmbedding( DirectSumDiagonalDifference( diagram ) );
    sources_of_diagram := List( diagram, Source );
    test_source := List( [ 1 .. Length( diagram ) ], function ( i )
            return
             PreCompose( kernel_emb,
               ProjectionInFactorOfDirectSum( sources_of_diagram, i
                 ) );
        end );
    return UniversalMorphismIntoFiberProduct( diagram, test_source );
end;
```

Back to index

**IsomorphismFromKernelOfDiagonalDifferenceToFiberProduct as the inverse of IsomorphismFromFiberProductToKernelOfDiagona lDifference**

This derivation is for all categories. This derivation uses:

- IsomorphismFromFiberProductToKernelOfDiagonalDifference × 1
- InverseImmutable × 1

```
function ( diagram )
    return
     Inverse(
       IsomorphismFromFiberProductToKernelOfDiagonalDifference(
         diagram ) );
end;
```

Back to index

## Derivations for IsomorphismFromObjectToInternalHomWithGivenInternalHom

**IsomorphismFromObjectToInternalHomWithGivenInternalHom using the evaluation morphism**

This derivation is for symmetric closed monoidal categories. This derivation uses:

- TensorUnit × 1
- PreCompose × 1
- RightUnitorInverseWithGivenTensorProduct × 1
- TensorProductOnObjects × 1
- EvaluationMorphismWithGivenSource × 1
- InternalHomOnObjects × 1

```
function ( object, internal_hom )
    local  unit, morphism;
    unit := TensorUnit( CapCategory( object ) );
    morphism := EvaluationMorphism( unit, object );
    return PreCompose( RightUnitorInverse( internal_hom ), morphism
       );
end;
```

Back to index

## Derivations for IsomorphismFromPushoutToCokernelOfDiagonalDifference

**IsomorphismFromPushoutToCokernelOfDiagonalDifference using the universal property of the pushout**

This derivation is for all categories. This derivation uses:

- CokernelProjection × 1
- PreCompose × 2
- InjectionOfCofactorOfDirectSum × 2
- UniversalMorphismFromPushout × 1
- DirectSumCodiagonalDifference × 1

```
function ( diagram )
    local  cokernel_proj, ranges_of_diagram, test_sink;
    cokernel_proj
     := CokernelProjection( DirectSumCodiagonalDifference( diagram
         ) );
    ranges_of_diagram := List( diagram, Range );
    test_sink := List( [ 1 .. Length( diagram ) ], function ( i )
            return
             PreCompose( InjectionOfCofactorOfDirectSum(
                 ranges_of_diagram, i ), cokernel_proj );
        end );
    return UniversalMorphismFromPushout( diagram, test_sink );
end;
```

Back to index

**IsomorphismFromPushoutToCokernelOfDiagonalDifference as the inverse of IsomorphismFromCokernelOfDiagonalDifferenceToPushout**

This derivation is for all categories. This derivation uses:

- IsomorphismFromCokernelOfDiagonalDifferenceToPushout × 1
- InverseImmutable × 1

```
function ( diagram )
    return
     Inverse( IsomorphismFromCokernelOfDiagonalDifferenceToPushout(
         diagram ) );
end;
```

Back to index

## Derivations for IsomorphismFromTensorProductToInternalHom

**IsomorphismFromTensorProductToInternalHom using MorphismFromTensorProductToInternalHom**

This derivation is for rigid symmetric closed monoidal categories. This derivation uses:

- MorphismFromTensorProductToInternalHomWithGivenObjects × 1
- DualOnObjects × 1
- TensorProductOnObjects × 1
- InternalHomOnObjects × 1

```
function ( object_1, object_2 )
    return MorphismFromTensorProductToInternalHom( object_1,
       object_2 );
end;
```

Back to index

## Derivations for IsomorphismFromTerminalObjectToZeroObject

**IsomorphismFromTerminalObjectToZeroObject as the inverse of IsomorphismFromZeroObjectToTerminalObject**

This derivation is for all categories. This derivation uses:

- InverseImmutable $\times 1$
- IsomorphismFromZeroObjectToTerminalObject $\times 1$

```
function ( category )
    return
     Inverse( IsomorphismFromZeroObjectToTerminalObject( category )
       );
end;
```

Back to index

**IsomorphismFromTerminalObjectToZeroObject using the universal property of the zero object**

This derivation is for all categories. This derivation uses:

- UniversalMorphismIntoZeroObject $\times 1$
- TerminalObject $\times 1$

```
function ( category )
    return UniversalMorphismIntoZeroObject(
       TerminalObject( category ) );
end;
```

Back to index

## Derivations for IsomorphismFromZeroObjectToInitialObject

**IsomorphismFromZeroObjectToInitialObject as the inverse of IsomorphismFromInitialObjectToZeroObject**

This derivation is for all categories. This derivation uses:

- InverseImmutable $\times 1$
- IsomorphismFromInitialObjectToZeroObject $\times 1$

```
function ( category )
    return
    Inverse( IsomorphismFromInitialObjectToZeroObject( category ) );
end;
```

Back to index

**IsomorphismFromZeroObjectToInitialObject using the universal property of the zero object**

This derivation is for all categories. This derivation uses:

- UniversalMorphismFromZeroObject $\times 1$
- InitialObject $\times 1$

```
function ( category )
    return UniversalMorphismFromZeroObject(
       InitialObject( category ) );
end;
```

Back to index

## Derivations for IsomorphismFromZeroObjectToTerminalObject

**IsomorphismFromZeroObjectToTerminalObject as the unique morphism from zero object to terminal object**

This derivation is for additive categories. This derivation uses:

- UniversalMorphismIntoTerminalObject $\times 1$
- ZeroObject $\times 1$

```
function ( category )
    return UniversalMorphismIntoTerminalObject(
       ZeroObject( category ) );
end;
```

Back to index

**IsomorphismFromZeroObjectToTerminalObject using the universal property of the zero object**

This derivation is for all categories. This derivation uses:

- UniversalMorphismFromZeroObject $\times 1$
- TerminalObject $\times 1$

```
function ( category )
    return UniversalMorphismFromZeroObject(
       TerminalObject( category ) );
end;
```

Back to index

**IsomorphismFromZeroObjectToTerminalObject as the inverse of IsomorphismFromTerminalObjectToZeroObject**

This derivation is for all categories. This derivation uses:

- InverseImmutable × 1
- IsomorphismFromTerminalObjectToZeroObject × 1

```
function ( category )
    return
     Inverse( IsomorphismFromTerminalObjectToZeroObject( category )
       );
end;
```

Back to index

### Derivations for KernelLift

**KernelLift using LiftAlongMonomorphism and KernelEmbedding**

This derivation is for all categories. This derivation uses:

- LiftAlongMonomorphism × 1
- KernelEmbeddingWithGivenKernelObject × 1
- KernelEmbedding × 1

```
function ( mor, test_morphism )
    return LiftAlongMonomorphism( KernelEmbedding( mor ),
       test_morphism );
end;
```

Back to index

### Derivations for KernelLiftWithGivenKernelObject

**KernelLift using LiftAlongMonomorphism and KernelEmbedding**

This derivation is for all categories. This derivation uses:

- LiftAlongMonomorphism × 1
- KernelEmbeddingWithGivenKernelObject × 1
- KernelEmbedding × 1

```
function ( mor, test_morphism, kernel )
    return
     LiftAlongMonomorphism( KernelEmbeddingWithGivenKernelObject(
         mor, kernel ), test_morphism );
end;
```

Back to index

### Derivations for KernelObject

**KernelObject as the source of KernelEmbedding**

This derivation is for all categories. This derivation uses:

- KernelEmbedding × 1

```
function ( mor )
    return Source( KernelEmbedding( mor ) );
end;
```

Back to index

### Derivations for KernelObjectFunctorialWithGivenKernelObjects

**KernelObjectFunctorialWithGivenKernelObjects using the universality of the kernel**

This derivation is for all categories. This derivation uses:

- KernelLift $\times 1$
- PreCompose $\times 1$
- KernelEmbedding $\times 1$

```
function ( kernel_alpha, alpha, mu, alpha_p, kernel_alpha_p )
    return
     KernelLift( alpha_p, PreCompose( KernelEmbedding( alpha ), mu
         ) );
end;
```

Back to index

### Derivations for LambdaElimination

**LambdaElimination using the tensor hom adjunction and left unitor**

This derivation is for symmetric closed monoidal categories. This derivation uses:

- PreCompose $\times 1$
- InternalHomToTensorProductAdjunctionMap $\times 1$
- LeftUnitorInverseWithGivenTensorProduct $\times 1$
- TensorProductOnObjects $\times 1$
- TensorUnit $\times 1$

```
function ( object_1, object_2, morphism )
    local  result_morphism;
    result_morphism := InternalHomToTensorProductAdjunctionMap(
       object_1, object_2, morphism );
    return PreCompose( LeftUnitorInverse( object_1 ),
       result_morphism );
end;
```

Back to index

## Derivations for LambdaIntroduction

### LambdaIntroduction using the tensor hom adjunction and left unitor

This derivation is for symmetric closed monoidal categories. This derivation uses:

- TensorUnit × 1
- PreCompose × 1
- TensorProductToInternalHomAdjunctionMap × 1
- LeftUnitorWithGivenTensorProduct × 1
- TensorProductOnObjects × 1

```
function ( morphism )
    local  result_morphism, category, source;
    category := CapCategory( morphism );
    source := Source( morphism );
    result_morphism := PreCompose( LeftUnitor( source ), morphism );
    return TensorProductToInternalHomAdjunctionMap(
       TensorUnit( category ), source, result_morphism );
end;
```

Back to index

## Derivations for LeftDistributivityExpandingWithGivenObjects

### LeftDistributivityExpandingWithGivenObjects using the universal property of the direct sum

This derivation is for additive categories. This derivation uses:

- IdentityMorphism × 1
- ProjectionInFactorOfDirectSum × 2
- UniversalMorphismIntoDirectSum × 1
- TensorProductOnObjects × 4
- TensorProductOnMorphismsWithGivenTensorProducts × 2

```
function ( factored_object, object, summands, expanded_object )
    local  nr_summands, projection_list, id, diagram;
    nr_summands := Size( summands );
    projection_list := List( [ 1 .. nr_summands ], function ( i )
            return ProjectionInFactorOfDirectSum( summands, i );
        end );
    id := IdentityMorphism( object );
    projection_list := List( projection_list, function ( mor )
            return TensorProductOnMorphisms( id, mor );
        end );
    diagram := List( summands, function ( summand )
            return TensorProductOnObjects( object, summand );
        end );
    return UniversalMorphismIntoDirectSum( diagram, projection_list
        );
end;
```

Back to index

## Derivations for LeftDistributivityFactoringWithGivenObjects

**LeftDistributivityFactoringWithGivenObjects using the universal property of the direct sum**

This derivation is for additive categories. This derivation uses:

- IdentityMorphism × 1
- InjectionOfCofactorOfDirectSum × 2
- UniversalMorphismFromDirectSum × 1
- TensorProductOnObjects × 4
- TensorProductOnMorphismsWithGivenTensorProducts × 2

```
function ( expanded_object, object, summands, factored_object )
    local  nr_summands, injection_list, id, diagram;
    nr_summands := Size( summands );
    injection_list := List( [ 1 .. nr_summands ], function ( i )
            return InjectionOfCofactorOfDirectSum( summands, i );
        end );
    id := IdentityMorphism( object );
    injection_list := List( injection_list, function ( mor )
            return TensorProductOnMorphisms( id, mor );
        end );
    diagram := List( summands, function ( summand )
            return TensorProductOnObjects( object, summand );
        end );
    return UniversalMorphismFromDirectSum( diagram, injection_list );
end;
```

Back to index

## Derivations for LeftUnitorInverseWithGivenTensorProduct

**LeftUnitorInverseWithGivenTensorProduct as the identity morphism**

This derivation is for all categories. This derivation uses:

- IdentityMorphism × 1

```
function ( object, unit_tensored_object )
    return IdentityMorphism( object );
end;
```

Back to index

**LeftUnitorInverseWithGivenTensorProduct as the inverse of LeftUnitor-WithGivenTensorProduct**

This derivation is for all categories. This derivation uses:

- LeftUnitorWithGivenTensorProduct × 1

```
function ( object, unit_tensored_object )
    return
     Inverse( LeftUnitorWithGivenTensorProduct( object,
         unit_tensored_object ) );
end;
```

Back to index

### Derivations for LeftUnitorWithGivenTensorProduct

**LeftUnitorWithGivenTensorProduct as the identity morphism**

This derivation is for all categories. This derivation uses:

- IdentityMorphism × 1

```
function ( object, unit_tensored_object )
    return IdentityMorphism( object );
end;
```

Back to index

**LeftUnitorWithGivenTensorProduct as the inverse of LeftUnitorInverse-WithGivenTensorProduct**

This derivation is for all categories. This derivation uses:

- LeftUnitorInverseWithGivenTensorProduct × 1

```
function ( object, unit_tensored_object )
    return
     Inverse( LeftUnitorInverseWithGivenTensorProduct( object,
         unit_tensored_object ) );
end;
```

Back to index

### Derivations for LiftAlongMonomorphism

**LiftAlongMonomorphism using Lift**

This derivation is for all categories. This derivation uses:

- Lift × 1

```
function ( alpha, beta )
    return Lift( beta, alpha );
end;
```

Back to index

### Derivations for MonoidalPostComposeMorphismWithGivenObjects

**MonoidalPostComposeMorphismWithGivenObjects using associator, evaluation, and tensor hom adjunction**

This derivation is for symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism × 1
- PreCompose × 1
- TensorProductOnObjects × 4
- InternalHomOnObjects × 2
- TensorProductToInternalHomAdjunctionMap × 1
- TensorProductOnMorphismsWithGivenTensorProducts × 1
- AssociatorLeftToRightWithGivenTensorProducts × 1
- EvaluationMorphismWithGivenSource × 2

```
function ( new_source, x, y, z, new_range )
    local  hom_x_y, hom_y_z, morphism;
    hom_x_y := InternalHomOnObjects( x, y );
    hom_y_z := InternalHomOnObjects( y, z );
    morphism
     :=
      PreCompose( [ AssociatorLeftToRight( hom_y_z, hom_x_y, x ),
          TensorProductOnMorphisms( IdentityMorphism( hom_y_z ),
             EvaluationMorphism( x, y ) ),
          EvaluationMorphism( y, z ) ] );
    return TensorProductToInternalHomAdjunctionMap(
       TensorProductOnObjects( hom_y_z, hom_x_y ), x, morphism );
end;
```

Back to index

**MonoidalPostComposeMorphismWithGivenObjects using evaluation, and tensor hom adjunction**

This derivation is for symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism × 1
- PreCompose × 1
- TensorProductOnObjects × 2
- InternalHomOnObjects × 2
- TensorProductToInternalHomAdjunctionMap × 1
- TensorProductOnMorphismsWithGivenTensorProducts × 1
- EvaluationMorphismWithGivenSource × 2

```
function ( new_source, x, y, z, new_range )
    local  hom_x_y, hom_y_z, morphism;
    hom_x_y := InternalHomOnObjects( x, y );
    hom_y_z := InternalHomOnObjects( y, z );
    morphism
     :=
      PreCompose(
       [ TensorProductOnMorphisms( IdentityMorphism( hom_y_z ),
             EvaluationMorphism( x, y ) ),
          EvaluationMorphism( y, z ) ] );
    return TensorProductToInternalHomAdjunctionMap(
       TensorProductOnObjects( hom_y_z, hom_x_y ), x, morphism );
end;
```

Back to index

**MonoidalPostComposeMorphismWithGivenObjects using MonoidalPreComposeMorphism and braiding**

This derivation is for symmetric closed monoidal categories. This derivation uses:

- PreCompose $\times 1$
- InternalHomOnObjects $\times 3$
- BraidingWithGivenTensorProducts $\times 1$
- TensorProductOnObjects $\times 2$
- MonoidalPreComposeMorphismWithGivenObjects $\times 1$

```
function ( new_source, x, y, z, new_range )
    local  braiding;
    braiding := Braiding( InternalHomOnObjects( y, z ),
       InternalHomOnObjects( x, y ) );
    return
     PreCompose( braiding, MonoidalPreComposeMorphism( x, y, z ) );
end;
```

Back to index

## Derivations for MonoidalPreComposeMorphismWithGivenObjects

**MonoidalPreComposeMorphismWithGivenObjects using associator, braiding, evaluation, and tensor hom adjunction**

This derivation is for symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism $\times 2$
- PreCompose $\times 1$
- TensorProductOnObjects $\times 4$
- InternalHomOnObjects $\times 2$
- TensorProductToInternalHomAdjunctionMap $\times 1$
- TensorProductOnMorphismsWithGivenTensorProducts $\times 2$
- AssociatorRightToLeftWithGivenTensorProducts $\times 1$
- AssociatorLeftToRightWithGivenTensorProducts $\times 1$

- BraidingWithGivenTensorProducts × 2
- EvaluationMorphismWithGivenSource × 2

```
function ( new_source, x, y, z, new_range )
    local  hom_x_y, hom_y_z, morphism;
    hom_x_y := InternalHomOnObjects( x, y );
    hom_y_z := InternalHomOnObjects( y, z );
    morphism
     :=
      PreCompose( [ AssociatorLeftToRight( hom_x_y, hom_y_z, x ),
          TensorProductOnMorphisms( IdentityMorphism( hom_x_y ),
             Braiding( hom_y_z, x ) ),
          AssociatorRightToLeft( hom_x_y, x, hom_y_z ),
          TensorProductOnMorphisms( EvaluationMorphism( x, y ),
             IdentityMorphism( hom_y_z ) ), Braiding( y, hom_y_z ),
          EvaluationMorphism( y, z ) ] );
    return TensorProductToInternalHomAdjunctionMap(
       TensorProductOnObjects( hom_x_y, hom_y_z ), x, morphism );
end;
```

Back to index

**MonoidalPreComposeMorphismWithGivenObjects using, braiding, evaluation, and tensor hom adjunction**

This derivation is for symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism × 2
- PreCompose × 1
- TensorProductOnObjects × 4
- InternalHomOnObjects × 2
- TensorProductToInternalHomAdjunctionMap × 1
- TensorProductOnMorphismsWithGivenTensorProducts × 2
- BraidingWithGivenTensorProducts × 2
- EvaluationMorphismWithGivenSource × 2

```
function ( new_source, x, y, z, new_range )
    local  hom_x_y, hom_y_z, morphism;
    hom_x_y := InternalHomOnObjects( x, y );
    hom_y_z := InternalHomOnObjects( y, z );
    morphism
     :=
      PreCompose(
       [ TensorProductOnMorphisms( IdentityMorphism( hom_x_y ),
             Braiding( hom_y_z, x ) ),
          TensorProductOnMorphisms( EvaluationMorphism( x, y ),
             IdentityMorphism( hom_y_z ) ), Braiding( y, hom_y_z ),
          EvaluationMorphism( y, z ) ] );
    return TensorProductToInternalHomAdjunctionMap(
       TensorProductOnObjects( hom_x_y, hom_y_z ), x, morphism );
end;
```

Back to index

**MonoidalPreComposeMorphismWithGivenObjects using MonoidalPostComposeMorphism and braiding**

This derivation is for symmetric closed monoidal categories. This derivation uses:

- PreCompose $\times 1$
- InternalHomOnObjects $\times 3$
- BraidingWithGivenTensorProducts $\times 1$
- TensorProductOnObjects $\times 2$
- MonoidalPostComposeMorphismWithGivenObjects $\times 1$

```
function ( new_source, x, y, z, new_range )
    local  braiding;
    braiding := Braiding( InternalHomOnObjects( x, y ),
       InternalHomOnObjects( y, z ) );
    return
     PreCompose( braiding, MonoidalPostComposeMorphism( x, y, z ) );
end;
```

Back to index

## Derivations for MorphismFromBidualWithGivenBidual

**MorphismFromBidualWithGivenBidual as the inverse of MorphismToBidualWithGivenBidual**

This derivation is for rigid symmetric closed monoidal categories. This derivation uses:

- MorphismToBidualWithGivenBidual $\times 1$

```
function ( object, bidual )
    return
     Inverse( MorphismToBidualWithGivenBidual( object, bidual ) );
end;
```

Back to index

## Derivations for MorphismFromCoimageToImageWithGivenObjects

**MorphismFromCoimageToImageWithGivenObjects using that images are given by kernels of cokernels**

This derivation is for pre Abelian categories. This derivation uses:

- CokernelProjection $\times 1$
- IsomorphismFromKernelOfCokernelToImageObject $\times 1$

- CoimageProjection × 1
- AstrictionToCoimage × 1
- KernelLift × 1
- PreCompose × 1

```
function ( coimage, morphism, image )
    local  coimage_projection, cokernel_projection, kernel_lift;
    cokernel_projection := CokernelProjection( morphism );
    coimage_projection := CoimageProjection( morphism );
    kernel_lift := KernelLift( cokernel_projection,
       AstrictionToCoimage( morphism ) );
    return
     PreCompose( kernel_lift,
       IsomorphismFromKernelOfCokernelToImageObject( morphism ) );
end;
```

Back to index

## Derivations for MorphismFromInternalHomToTensorProductWithGivenObjects

### MorphismFromInternalHomToTensorProductWithGivenObjects using IsomorphismFromInternalHomToTensorProduct

This derivation is for rigid symmetric closed monoidal categories. This derivation uses:

- IsomorphismFromInternalHomToTensorProduct × 1

```
function ( tensor_object, object_1, object_2, internal_hom )
    return IsomorphismFromInternalHomToTensorProduct( object_1,
       object_2 );
end;
```

Back to index

## Derivations for MorphismFromTensorProductToInternalHomWithGivenObjects

### MorphismFromTensorProductToInternalHomWithGivenObjects using IsomorphismFromTensorProductToInternalHom

This derivation is for rigid symmetric closed monoidal categories. This derivation uses:

- IsomorphismFromTensorProductToInternalHom × 1

```
function ( tensor_object, object_1, object_2, internal_hom )
    return IsomorphismFromTensorProductToInternalHom( object_1,
       object_2 );
end;
```

Back to index

**MorphismFromTensorProductToInternalHomWithGivenObjects using TensorProductInternalHomCompatibilityMorphism**

This derivation is for symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism $\times 1$
- TensorUnit $\times 1$
- IsomorphismFromDualToInternalHom $\times 1$
- PreCompose $\times 1$
- RightUnitorWithGivenTensorProduct $\times 1$
- TensorProductOnObjects $\times 4$
- IsomorphismFromObjectToInternalHomWithGivenInternalHom $\times 1$
- InternalHomOnObjects $\times 3$
- IsomorphismFromInternalHomToObjectWithGivenInternalHom $\times 1$
- TensorProductOnMorphismsWithGivenTensorProducts $\times 2$
- TensorProductInternalHomCompatibilityMorphismWithGivenObjects $\times 1$

```
function ( tensor_object, object_1, object_2, internal_hom )
    local  unit, morphism;
    unit := TensorUnit( CapCategory( object_1 ) );
    morphism
     :=
      PreCompose(
       [
          TensorProductOnMorphisms( IsomorphismFromDualToInternalHom(
               object_1 ), IsomorphismFromObjectToInternalHom(
               object_2 ) ),
          TensorProductInternalHomCompatibilityMorphism( object_1,
             unit, unit, object_2 ),
          TensorProductOnMorphisms( IdentityMorphism( internal_hom )
              , IsomorphismFromInternalHomToObject( unit ) ),
          RightUnitor( internal_hom ) ] );
    return morphism;
end;
```

Back to index

## Derivations for MorphismToBidualWithGivenBidual

**MorphismToBidualWithGivenBidual as the inverse of MorphismFromBidualWithGivenBidual**

This derivation is for rigid symmetric closed monoidal categories. This derivation uses:

- MorphismFromBidualWithGivenBidual $\times 1$

```
function ( object, bidual )
    return
    Inverse( MorphismFromBidualWithGivenBidual( object, bidual ) );
end;
```

Back to index

**MorphismToBidualWithGivenBidual using the braiding and the universal property of the dual**

This derivation is for symmetric closed monoidal categories. This derivation uses:

- DualOnObjects × 2
- PreCompose × 1
- UniversalPropertyOfDual × 1
- EvaluationForDualWithGivenTensorProduct × 1
- TensorProductOnObjects × 2
- TensorUnit × 1
- BraidingWithGivenTensorProducts × 1

```
function ( object, bidual )
    local  morphism;
    morphism := Braiding( object, DualOnObjects( object ) );
    morphism := PreCompose( morphism, EvaluationForDual( object ) );
    return UniversalPropertyOfDual( object, DualOnObjects( object )
        , morphism );
end;
```

Back to index

**MorphismToBidualWithGivenBidual using Coevaluation, InternalHom, and Evaluation**

This derivation is for symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism × 2
- TensorUnit × 1
- DualOnObjects × 1
- PreCompose × 1
- BraidingWithGivenTensorProducts × 1
- TensorProductOnObjects × 2
- InternalHomOnMorphismsWithGivenInternalHoms × 2
- InternalHomOnObjects × 4
- EvaluationMorphismWithGivenSource × 1
- CoevaluationMorphismWithGivenRange × 1

```
function ( object, bidual )
    local  morphism, dual_object, tensor_unit;
    dual_object := DualOnObjects( object );
    tensor_unit := TensorUnit( CapCategory( object ) );
    morphism
     :=
      PreCompose(
        [ CoevaluationMorphism( object, dual_object ),
            InternalHomOnMorphisms( IdentityMorphism( dual_object ),
               Braiding( object, dual_object ) ),
            InternalHomOnMorphisms( IdentityMorphism( dual_object ),
               EvaluationMorphism( object, tensor_unit ) ) ] );
    return morphism;
end;
```

Back to index

## Derivations for PostCompose

### PostCompose using PreCompose and swapping arguments

This derivation is for all categories. This derivation uses:

- PreCompose × 1

```
function ( right_mor, left_mor )
    return PreCompose( left_mor, right_mor );
end;
```

Back to index

## Derivations for PreCompose

### PreCompose using PostCompose and swapping arguments

This derivation is for all categories. This derivation uses:

- PostCompose × 1

```
function ( left_mor, right_mor )
    return PostCompose( right_mor, left_mor );
end;
```

Back to index

## Derivations for ProjectionInFactorOfDirectProduct

### ProjectionInFactorOfDirectProduct using ProjectionInFactorOfDirectSum

This derivation is for all categories. This derivation uses:

- AdditionForMorphisms × 1
- AdditiveInverseForMorphisms × 1
- PreCompose × 1
- ProjectionInFactorOfDirectSum × 1
- IsomorphismFromDirectProductToDirectSum × 1

```
function ( diagram, projection_number )
    return
     PreCompose( IsomorphismFromDirectProductToDirectSum( diagram )
        , ProjectionInFactorOfDirectSum( diagram, projection_number
         ) );
end;
```

Back to index

### Derivations for ProjectionInFactorOfDirectSum

**ProjectionInFactorOfDirectSum using ProjectionInFactorOfDirectProduct**
This derivation is for all categories. This derivation uses:

- AdditionForMorphisms $\times 1$
- AdditiveInverseForMorphisms $\times 1$
- PreCompose $\times 1$
- ProjectionInFactorOfDirectProduct $\times 1$
- IsomorphismFromDirectSumToDirectProduct $\times 1$

```
function ( diagram, projection_number )
    return
     PreCompose( IsomorphismFromDirectSumToDirectProduct( diagram )
        , ProjectionInFactorOfDirectProduct( diagram,
         projection_number ) );
end;
```

Back to index

### Derivations for ProjectionInFactorOfFiberProduct

**ProjectionInFactorOfFiberProduct by composing the direct sum embedding with the direct sum projection**
This derivation is for all categories. This derivation uses:

- AdditionForMorphisms $\times 1$
- AdditiveInverseForMorphisms $\times 1$
- PreCompose $\times 1$
- ProjectionInFactorOfDirectSum $\times 1$
- FiberProductEmbeddingInDirectSum $\times 1$

```
function ( diagram, projection_number )
    local  embedding_in_direct_sum, direct_sum_diagram, projection;
    embedding_in_direct_sum := FiberProductEmbeddingInDirectSum(
       diagram );
    direct_sum_diagram := List( diagram, Source );
    projection := ProjectionInFactorOfDirectSum( direct_sum_diagram
        , projection_number );
    return PreCompose( embedding_in_direct_sum, projection );
end;
```

Back to index

## Derivations for Pushout

### Pushout as the range of DirectSumProjectionInPushout

This derivation is for all categories. This derivation uses:

- DirectSumProjectionInPushout $\times 1$

```
function ( diagram )
    return Range( DirectSumProjectionInPushout( diagram ) );
end;
```

Back to index

## Derivations for PushoutFunctorialWithGivenPushouts

### PushoutFunctorialWithGivenPushouts using the universality of the pushout

This derivation is for all categories. This derivation uses:

- PreCompose $\times 2$
- InjectionOfCofactorOfPushout $\times 2$
- UniversalMorphismFromPushout $\times 1$

```
function ( pushout_source, morphism_of_morphisms, pushout_range )
    local  pushout_diagram, sink, diagram;
    pushout_diagram := List( morphism_of_morphisms, function ( mor )
            return mor[3];
        end );
    sink := List( [ 1 .. Length( morphism_of_morphisms ) ],
       function ( i )
            return PreCompose( morphism_of_morphisms[i][2],
               InjectionOfCofactorOfPushout( pushout_diagram, i ) );
        end );
    diagram := List( morphism_of_morphisms, function ( mor )
            return mor[1];
        end );
    return UniversalMorphismFromPushout( diagram, sink );
end;
```

Back to index

## Derivations for RankMorphism

### Rank of an object as the trace of its identity

This derivation is for rigid symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism $\times 1$
- TraceMap $\times 1$

```
function ( object )
    return TraceMap( IdentityMorphism( object ) );
end;
```

Back to index

## Derivations for RightDistributivityExpandingWithGivenObjects

### RightDistributivityExpandingWithGivenObjects using the universal property of the direct sum

This derivation is for additive categories. This derivation uses:

- IdentityMorphism $\times 1$
- ProjectionInFactorOfDirectSum $\times 2$
- UniversalMorphismIntoDirectSum $\times 1$
- TensorProductOnObjects $\times 4$
- TensorProductOnMorphismsWithGivenTensorProducts $\times 2$

```
function ( factored_object, summands, object, expanded_object )
    local  nr_summands, projection_list, id, diagram;
    nr_summands := Size( summands );
    projection_list := List( [ 1 .. nr_summands ], function ( i )
            return ProjectionInFactorOfDirectSum( summands, i );
        end );
    id := IdentityMorphism( object );
    projection_list := List( projection_list, function ( mor )
            return TensorProductOnMorphisms( mor, id );
        end );
    diagram := List( summands, function ( summand )
            return TensorProductOnObjects( summand, object );
        end );
    return UniversalMorphismIntoDirectSum( diagram, projection_list
       );
end;
```

Back to index

## Derivations for RightDistributivityFactoringWithGivenObjects

**RightDistributivityFactoringWithGivenObjects using the universal property of the direct sum**

This derivation is for additive categories. This derivation uses:

- IdentityMorphism × 1
- InjectionOfCofactorOfDirectSum × 2
- UniversalMorphismFromDirectSum × 1
- TensorProductOnObjects × 4
- TensorProductOnMorphismsWithGivenTensorProducts × 2

```
function ( expanded_object, summands, object, factored_object )
    local  nr_summands, injection_list, id, diagram;
    nr_summands := Size( summands );
    injection_list := List( [ 1 .. nr_summands ], function ( i )
            return InjectionOfCofactorOfDirectSum( summands, i );
        end );
    id := IdentityMorphism( object );
    injection_list := List( injection_list, function ( mor )
            return TensorProductOnMorphisms( mor, id );
        end );
    diagram := List( summands, function ( summand )
            return TensorProductOnObjects( summand, object );
        end );
    return UniversalMorphismFromDirectSum( diagram, injection_list );
end;
```

Back to index

## Derivations for RightUnitorInverseWithGivenTensorProduct

**RightUnitorInverseWithGivenTensorProduct as the identity morphism**

This derivation is for all categories. This derivation uses:

- IdentityMorphism × 1

```
function ( object, object_tensored_unit )
    return IdentityMorphism( object );
end;
```

Back to index

**RightUnitorInverseWithGivenTensorProduct as the inverse of RightUnitorWithGivenTensorProduct**

This derivation is for all categories. This derivation uses:

- RightUnitorWithGivenTensorProduct × 1

```
function ( object, object_tensored_unit )
    return
     Inverse( RightUnitorWithGivenTensorProduct( object,
         object_tensored_unit ) );
end;
```

Back to index

## Derivations for RightUnitorWithGivenTensorProduct

### RightUnitorWithGivenTensorProduct as the identity morphism

This derivation is for all categories. This derivation uses:

- IdentityMorphism × 1

```
function ( object, object_tensored_unit )
    return IdentityMorphism( object );
end;
```

Back to index

### RightUnitorWithGivenTensorProduct as the inverse of RightUnitorInverse-WithGivenTensorProduct

This derivation is for all categories. This derivation uses:

- RightUnitorInverseWithGivenTensorProduct × 1

```
function ( object, object_tensored_unit )
    return
     Inverse( RightUnitorInverseWithGivenTensorProduct( object,
         object_tensored_unit ) );
end;
```

Back to index

## Derivations for TensorProductDualityCompatibilityMorphismWithGivenObjects

### TensorProductDualityCompatibilityMorphismWithGivenObjects using left unitoar, and compatibility of tensor product and internal hom

This derivation is for symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism × 1
- TensorUnit × 1
- IsomorphismFromDualToInternalHom × 2
- IsomorphismFromInternalHomToDual × 1
- PreCompose × 1
- TensorProductOnObjects × 3
- LeftUnitorWithGivenTensorProduct × 1

- TensorProductOnMorphismsWithGivenTensorProducts × 1
- InternalHomOnMorphismsWithGivenInternalHoms × 1
- InternalHomOnObjects × 3
- TensorProductInternalHomCompatibilityMorphismWithGivenObjects × 1

```
function ( new_source, object_1, object_2, new_range )
    local  morphism, unit, tensor_product_on_object_1_and_object_2;
    unit := TensorUnit( CapCategory( object_1 ) );
    tensor_product_on_object_1_and_object_2
     := TensorProductOnObjects( object_1, object_2 );
    morphism
     :=
      PreCompose(
       [
          TensorProductOnMorphisms( IsomorphismFromDualToInternalHom(
               object_1 ), IsomorphismFromDualToInternalHom(
               object_2 ) ),
          TensorProductInternalHomCompatibilityMorphism( object_1,
             unit, object_2, unit ),
          InternalHomOnMorphisms(
             IdentityMorphism(
               tensor_product_on_object_1_and_object_2 ),
             LeftUnitor( unit ) ),
          IsomorphismFromInternalHomToDual(
             tensor_product_on_object_1_and_object_2 ) ] );
    return morphism;
end;
```

Back to index

**TensorProductDualityCompatibilityMorphismWithGivenObjects using compatibility of tensor product and internal hom**

This derivation is for symmetric closed monoidal categories. This derivation uses:

- TensorUnit × 1
- IsomorphismFromDualToInternalHom × 2
- IsomorphismFromInternalHomToDual × 1
- PreCompose × 1
- TensorProductOnObjects × 3
- TensorProductOnMorphismsWithGivenTensorProducts × 1
- TensorProductInternalHomCompatibilityMorphismWithGivenObjects × 1
- InternalHomOnObjects × 3

```
function ( new_source, object_1, object_2, new_range )
    local  morphism, unit, tensor_product_on_object_1_and_object_2;
    unit := TensorUnit( CapCategory( object_1 ) );
    tensor_product_on_object_1_and_object_2
     := TensorProductOnObjects( object_1, object_2 );
    morphism
     :=
      PreCompose(
       [
          TensorProductOnMorphisms( IsomorphismFromDualToInternalHom(
               object_1 ), IsomorphismFromDualToInternalHom(
               object_2 ) ),
          TensorProductInternalHomCompatibilityMorphism( object_1,
             unit, object_2, unit ),
          IsomorphismFromInternalHomToDual(
             tensor_product_on_object_1_and_object_2 ) ] );
    return morphism;
end;
```

Back to index

## Derivations for TensorProductInternalHomCompatibilityMorphismInverseWithGivenObjects

**TensorProductInternalHomCompatibilityMorphismInverseWithGivenObjects as the inverse of TensorProductInternalHomCompatibilityMorphi smWithGivenObjects**

This derivation is for rigid symmetric closed monoidal categories. This derivation uses:

- TensorProductInternalHomCompatibilityMorphismWithGivenObjects × 1

```
function ( a1, b1, a2, b2, new_source_and_range_list )
    return
     Inverse(
       TensorProductInternalHomCompatibilityMorphismWithGivenObjects
          ( a1, b1, a2, b2, new_source_and_range_list ) );
end;
```

Back to index

## Derivations for TensorProductInternalHomCompatibilityMorphismWithGivenObjects

**TensorProductInternalHomCompatibilityMorphismWithGivenObjects using associator, braiding an the evaluation morphism**

This derivation is for symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism × 4
- PreCompose × 1
- TensorProductOnObjects × 14

- InternalHomOnObjects × 2
- TensorProductToInternalHomAdjunctionMap × 1
- TensorProductOnMorphismsWithGivenTensorProducts × 7
- AssociatorRightToLeftWithGivenTensorProducts × 2
- AssociatorLeftToRightWithGivenTensorProducts × 2
- BraidingWithGivenTensorProducts × 1
- EvaluationMorphismWithGivenSource × 2

```
function ( a1, b1, a2, b2, new_source_and_range_list )
    local  morphism, int_hom_a1_b1, int_hom_a2_b2, id_a2,
    tensor_product_on_objects_int_hom_a1_b1_int_hom_a2_b2;
    int_hom_a1_b1 := InternalHomOnObjects( a1, b1 );
    int_hom_a2_b2 := InternalHomOnObjects( a2, b2 );
    id_a2 := IdentityMorphism( a2 );
    tensor_product_on_objects_int_hom_a1_b1_int_hom_a2_b2
     := TensorProductOnObjects( int_hom_a1_b1, int_hom_a2_b2 );
    morphism
     :=
      PreCompose(
       [
          AssociatorRightToLeft(
             tensor_product_on_objects_int_hom_a1_b1_int_hom_a2_b2,
             a1, a2 ),
          TensorProductOnMorphisms(
             AssociatorLeftToRight( int_hom_a1_b1, int_hom_a2_b2,
               a1 ), id_a2 ),
          TensorProductOnMorphisms(
             TensorProductOnMorphisms(
               IdentityMorphism( int_hom_a1_b1 ),
               Braiding( int_hom_a2_b2, a1 ) ), id_a2 ),
          TensorProductOnMorphisms(
             AssociatorRightToLeft( int_hom_a1_b1, a1,
               int_hom_a2_b2 ), id_a2 ),
          TensorProductOnMorphisms(
             TensorProductOnMorphisms( EvaluationMorphism( a1, b1 )
                , IdentityMorphism( int_hom_a2_b2 ) ), id_a2 ),
          AssociatorLeftToRight( b1, int_hom_a2_b2, a2 ),
          TensorProductOnMorphisms( IdentityMorphism( b1 ),
             EvaluationMorphism( a2, b2 ) ) ] );
    return TensorProductToInternalHomAdjunctionMap(
       tensor_product_on_objects_int_hom_a1_b1_int_hom_a2_b2,
       TensorProductOnObjects( a1, a2 ), morphism );
end;
```

Back to index

**TensorProductInternalHomCompatibilityMorphismWithGivenObjects using braiding an the evaluation morphism**

This derivation is for symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism $\times 4$
- PreCompose $\times 1$
- TensorProductOnObjects $\times 10$
- InternalHomOnObjects $\times 2$
- TensorProductToInternalHomAdjunctionMap $\times 1$
- TensorProductOnMorphismsWithGivenTensorProducts $\times 5$
- BraidingWithGivenTensorProducts $\times 1$
- EvaluationMorphismWithGivenSource $\times 2$

```
function ( a1, b1, a2, b2, new_source_and_range_list )
    local  morphism, int_hom_a1_b1, int_hom_a2_b2, id_a2,
    tensor_product_on_objects_int_hom_a1_b1_int_hom_a2_b2;
    int_hom_a1_b1 := InternalHomOnObjects( a1, b1 );
    int_hom_a2_b2 := InternalHomOnObjects( a2, b2 );
    id_a2 := IdentityMorphism( a2 );
    tensor_product_on_objects_int_hom_a1_b1_int_hom_a2_b2
     := TensorProductOnObjects( int_hom_a1_b1, int_hom_a2_b2 );
    morphism
     :=
      PreCompose(
       [
          TensorProductOnMorphisms(
             TensorProductOnMorphisms(
               IdentityMorphism( int_hom_a1_b1 ),
               Braiding( int_hom_a2_b2, a1 ) ), id_a2 ),
          TensorProductOnMorphisms(
             TensorProductOnMorphisms( EvaluationMorphism( a1, b1 )
                , IdentityMorphism( int_hom_a2_b2 ) ), id_a2 ),
          TensorProductOnMorphisms( IdentityMorphism( b1 ),
             EvaluationMorphism( a2, b2 ) ) ] );
    return TensorProductToInternalHomAdjunctionMap(
       tensor_product_on_objects_int_hom_a1_b1_int_hom_a2_b2,
       TensorProductOnObjects( a1, a2 ), morphism );
end;
```

Back to index

## Derivations for TensorProductToInternalHomAdjunctionMap

### TensorProductToInternalHomAdjunctionMap using CoevaluationMorphism and InternalHom

This derivation is for symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism $\times 1$
- PreCompose $\times 1$
- InternalHomOnMorphismsWithGivenInternalHoms $\times 1$
- InternalHomOnObjects $\times 2$
- CoevaluationMorphismWithGivenRange $\times 1$
- TensorProductOnObjects $\times 1$

```
function ( object_1, object_2, morphism )
    local  coevaluation, internal_hom_on_morphisms;
    coevaluation := CoevaluationMorphism( object_1, object_2 );
    internal_hom_on_morphisms
     := InternalHomOnMorphisms( IdentityMorphism( object_2 ),
       morphism );
    return PreCompose( coevaluation, internal_hom_on_morphisms );
end;
```

Back to index

## Derivations for TerminalObject

### TerminalObject as the source of IsomorphismFromTerminalObjectToZeroObject

This derivation is for all categories. This derivation uses:

- IsomorphismFromTerminalObjectToZeroObject $\times 1$

```
function ( category )
    return
     Source( IsomorphismFromTerminalObjectToZeroObject( category ) );
end;
```

Back to index

### TerminalObject as the range of IsomorphismFromZeroObjectToTerminalObject

This derivation is for all categories. This derivation uses:

- IsomorphismFromZeroObjectToTerminalObject $\times 1$

```
function ( category )
    return
     Range( IsomorphismFromZeroObjectToTerminalObject( category ) );
end;
```

Back to index

## Derivations for TerminalObjectFunctorial

### TerminalObjectFunctorial using the identity morphism of terminal object

This derivation is for all categories. This derivation uses:

- TerminalObject × 1
- IdentityMorphism × 1

```
function ( category )
    local  terminal_object;
    terminal_object := TerminalObject( category );
    return IdentityMorphism( terminal_object );
end;
```

Back to index

### TerminalObjectFunctorial using the universality of terminal object

This derivation is for all categories. This derivation uses:

- TerminalObject × 1
- UniversalMorphismIntoTerminalObject × 1

```
function ( category )
    local  terminal_object;
    terminal_object := TerminalObject( category );
    return UniversalMorphismIntoTerminalObject( terminal_object );
end;
```

Back to index

## Derivations for TraceMap

### TraceMap composing the lambda abstraction with the evaluation

This derivation is for rigid symmetric closed monoidal categories. This derivation uses:

- LambdaIntroduction × 1
- PreCompose × 2
- IsomorphismFromInternalHomToTensorProduct × 1
- EvaluationForDualWithGivenTensorProduct × 1
- TensorProductOnObjects × 1
- DualOnObjects × 1
- TensorUnit × 1

```
function ( morphism )
    local  result_morphism, object;
    result_morphism := LambdaIntroduction( morphism );
    object := Source( morphism );
    result_morphism := PreCompose( result_morphism,
      IsomorphismFromInternalHomToTensorProduct( object, object ) );
    return PreCompose( result_morphism, EvaluationForDual( object )
      );
end;
```

Back to index

## Derivations for UniversalMorphismFromCoproduct

**UniversalMorphismFromCoproduct using UniversalMorphismFromDirect-Sum**

This derivation is for all categories. This derivation uses:

- AdditionForMorphisms × 1
- AdditiveInverseForMorphisms × 1
- PreCompose × 1
- UniversalMorphismFromDirectSum × 1
- IsomorphismFromCoproductToDirectSum × 1

```
function ( diagram, sink )
    return
     PreCompose( IsomorphismFromCoproductToDirectSum( diagram ),
       UniversalMorphismFromDirectSum( diagram, sink ) );
end;
```

Back to index

## Derivations for UniversalMorphismFromDirectSum

**UniversalMorphismFromDirectSum using projections of the direct sum**

This derivation is for additive categories. This derivation uses:

- PreCompose × 2
- ProjectionInFactorOfDirectSumWithGivenDirectSum × 2
- AdditionForMorphisms × 1
- ProjectionInFactorOfDirectSum × 2

```
function ( diagram, sink )
    local  nr_components;
    nr_components := Length( sink );
    return Sum( List( [ 1 .. nr_components ], function ( i )
              return
               PreCompose( ProjectionInFactorOfDirectSum( diagram,
                   i ), sink[i] );
          end ) );
end;
```

Back to index

**UniversalMorphismFromDirectSum using UniversalMorphismFromCoproduct**

This derivation is for all categories. This derivation uses:

- AdditionForMorphisms × 1
- AdditiveInverseForMorphisms × 1
- PreCompose × 1
- UniversalMorphismFromCoproduct × 1
- IsomorphismFromDirectSumToCoproduct × 1

```
function ( diagram, sink )
    return
     PreCompose( IsomorphismFromDirectSumToCoproduct( diagram ),
       UniversalMorphismFromCoproduct( diagram, sink ) );
end;
```

Back to index

## Derivations for UniversalMorphismFromDirectSumWithGivenDirectSum

**UniversalMorphismFromDirectSum using projections of the direct sum**

This derivation is for additive categories. This derivation uses:

- PreCompose × 2
- ProjectionInFactorOfDirectSumWithGivenDirectSum × 2
- AdditionForMorphisms × 1
- ProjectionInFactorOfDirectSum × 2

```
function ( diagram, sink, direct_sum )
    local  nr_components;
    nr_components := Length( sink );
    return Sum( List( [ 1 .. nr_components ], function ( i )
              return
               PreCompose(
                 ProjectionInFactorOfDirectSumWithGivenDirectSum(
                   diagram, i, direct_sum ), sink[i] );
          end ) );
end;
```

Back to index

## Derivations for UniversalMorphismFromImage

**UniversalMorphismFromImage using ImageEmbedding and LiftAlongMonomorphism**

This derivation is for all categories. This derivation uses:

- LiftAlongMonomorphism × 1

- ImageEmbeddingWithGivenImageObject × 1
- ImageEmbedding × 1

```
function ( morphism, test_factorization )
    local  image_embedding;
    image_embedding := ImageEmbedding( morphism );
    return LiftAlongMonomorphism( test_factorization[2],
       image_embedding );
end;
```

Back to index

### Derivations for UniversalMorphismFromImageWithGivenImageObject

#### UniversalMorphismFromImage using ImageEmbedding and LiftAlongMonomorphism

This derivation is for all categories. This derivation uses:

- LiftAlongMonomorphism × 1
- ImageEmbeddingWithGivenImageObject × 1
- ImageEmbedding × 1

```
function ( morphism, test_factorization, image )
    local  image_embedding;
    image_embedding := ImageEmbeddingWithGivenImageObject(
       morphism, image );
    return LiftAlongMonomorphism( test_factorization[2],
       image_embedding );
end;
```

Back to index

### Derivations for UniversalMorphismFromInitialObject

#### UniversalMorphismFromInitialObject computing the zero morphism

This derivation is for additive categories. This derivation uses:

- ZeroMorphism × 1
- InitialObject × 1

```
function ( test_sink )
    local  initial_object;
    initial_object := InitialObject( CapCategory( test_sink ) );
    return ZeroMorphism( initial_object, test_sink );
end;
```

Back to index

#### UniversalMorphismFromInitialObject using UniversalMorphismFromZeroObject

This derivation is for all categories. This derivation uses:

- AdditionForMorphisms × 1
- AdditiveInverseForMorphisms × 1
- UniversalMorphismFromZeroObject × 1
- IsomorphismFromInitialObjectToZeroObject × 1
- PreCompose × 1

```
function ( obj )
    local  category;
    category := CapCategory( obj );
    return
     PreCompose( IsomorphismFromInitialObjectToZeroObject( category
         ), UniversalMorphismFromZeroObject( obj ) );
end;
```

Back to index

## Derivations for UniversalMorphismFromInitialObjectWithGivenInitialObject

### UniversalMorphismFromInitialObject computing the zero morphism

This derivation is for additive categories. This derivation uses:

- ZeroMorphism × 1
- InitialObject × 1

```
function ( test_sink, initial_object )
    return ZeroMorphism( initial_object, test_sink );
end;
```

Back to index

## Derivations for UniversalMorphismFromPushout

### UniversalMorphismFromPushout using the universality of the cokernel representation of the pushout

This derivation is for all categories. This derivation uses:

- AdditionForMorphisms × 1
- AdditiveInverseForMorphisms × 1
- CokernelColift × 1
- PreCompose × 1
- UniversalMorphismFromDirectSum × 1
- IsomorphismFromPushoutToCokernelOfDiagonalDifference × 1
- DirectSumCodiagonalDifference × 1

```
function ( diagram, sink )
    local  test_function, direct_sum_codiagonal_difference,
    cokernel_colift;
    test_function := CallFuncList( UniversalMorphismFromDirectSum,
       sink );
    direct_sum_codiagonal_difference
     := DirectSumCodiagonalDifference( diagram );
    cokernel_colift
     := CokernelColift( direct_sum_codiagonal_difference,
       test_function );
    return
     PreCompose(
       IsomorphismFromPushoutToCokernelOfDiagonalDifference(
         diagram ), cokernel_colift );
end;
```

Back to index

## Derivations for UniversalMorphismFromZeroObject

**UniversalMorphismFromZeroObject computing the zero morphism**
This derivation is for additive categories. This derivation uses:

- ZeroMorphism $\times 1$
- ZeroObject $\times 1$

```
function ( test_sink )
    local  zero_object;
    zero_object := ZeroObject( CapCategory( test_sink ) );
    return ZeroMorphism( zero_object, test_sink );
end;
```

Back to index

**UniversalMorphismFromZeroObject using UniversalMorphismFromInitial-Object**
This derivation is for all categories. This derivation uses:

- AdditionForMorphisms $\times 1$
- AdditiveInverseForMorphisms $\times 1$
- IsomorphismFromZeroObjectToInitialObject $\times 1$
- UniversalMorphismFromInitialObject $\times 1$
- PreCompose $\times 1$

```
function ( obj )
    local  category;
    category := CapCategory( obj );
    return
     PreCompose( IsomorphismFromZeroObjectToInitialObject( category
        ), UniversalMorphismFromInitialObject( obj ) );
end;
```

Back to index

### Derivations for UniversalMorphismFromZeroObjectWithGivenZeroObject

#### UniversalMorphismFromZeroObject computing the zero morphism

This derivation is for additive categories. This derivation uses:

- ZeroMorphism × 1
- ZeroObject × 1

```
function ( test_sink, zero_object )
    return ZeroMorphism( zero_object, test_sink );
end;
```

Back to index

### Derivations for UniversalMorphismIntoCoimage

#### UniversalMorphismIntoCoimage using CoimageProjection and ColiftAlong-Epimorphism

This derivation is for all categories. This derivation uses:

- ColiftAlongEpimorphism × 1
- CoimageProjectionWithGivenCoimage × 1
- CoimageProjection × 1

```
function ( morphism, test_factorization )
    local  coimage_projection;
    coimage_projection := CoimageProjection( morphism );
    return ColiftAlongEpimorphism( test_factorization[1],
       coimage_projection );
end;
```

Back to index

### Derivations for UniversalMorphismIntoCoimageWithGivenCoimage

#### UniversalMorphismIntoCoimage using CoimageProjection and ColiftAlong-Epimorphism

This derivation is for all categories. This derivation uses:

- ColiftAlongEpimorphism × 1
- CoimageProjectionWithGivenCoimage × 1
- CoimageProjection × 1

```
function ( morphism, test_factorization, coimage )
    local  coimage_projection;
    coimage_projection := CoimageProjectionWithGivenCoimage(
       morphism, coimage );
    return ColiftAlongEpimorphism( test_factorization[1],
       coimage_projection );
end;
```

Back to index

## Derivations for UniversalMorphismIntoDirectProduct

### UniversalMorphismIntoDirectProduct using UniversalMorphismIntoDirect-Sum

This derivation is for all categories. This derivation uses:

- AdditionForMorphisms $\times 1$
- AdditiveInverseForMorphisms $\times 1$
- PreCompose $\times 1$
- UniversalMorphismIntoDirectSum $\times 1$
- IsomorphismFromDirectSumToDirectProduct $\times 1$

```
function ( diagram, source )
    return
     PreCompose( UniversalMorphismIntoDirectSum( diagram, source ),
       IsomorphismFromDirectSumToDirectProduct( diagram ) );
end;
```

Back to index

## Derivations for UniversalMorphismIntoDirectSum

### UniversalMorphismIntoDirectSum using the injections of the direct sum

This derivation is for additive categories. This derivation uses:

- PreCompose $\times 2$
- InjectionOfCofactorOfDirectSumWithGivenDirectSum $\times 2$
- AdditionForMorphisms $\times 1$
- InjectionOfCofactorOfDirectSum $\times 2$

```
function ( diagram, source )
    local  nr_components;
    nr_components := Length( source );
    return Sum( List( [ 1 .. nr_components ], function ( i )
              return
               PreCompose( source[i],
                 InjectionOfCofactorOfDirectSum( diagram, i ) );
```

```
          end ) );
end;
```

Back to index

**UniversalMorphismIntoDirectSum using UniversalMorphismIntoDirectProduct**

This derivation is for all categories. This derivation uses:

- AdditionForMorphisms $\times 1$
- AdditiveInverseForMorphisms $\times 1$
- PreCompose $\times 1$
- UniversalMorphismIntoDirectProduct $\times 1$
- IsomorphismFromDirectProductToDirectSum $\times 1$

```
function ( diagram, source )
    return
     PreCompose( UniversalMorphismIntoDirectProduct( diagram,
         source ), IsomorphismFromDirectProductToDirectSum( diagram
         ) );
end;
```

Back to index

**Derivations for UniversalMorphismIntoDirectSumWithGivenDirectSum**

**UniversalMorphismIntoDirectSum using the injections of the direct sum**

This derivation is for additive categories. This derivation uses:

- PreCompose $\times 2$
- InjectionOfCofactorOfDirectSumWithGivenDirectSum $\times 2$
- AdditionForMorphisms $\times 1$
- InjectionOfCofactorOfDirectSum $\times 2$

```
function ( diagram, source, direct_sum )
    local  nr_components;
    nr_components := Length( source );
    return Sum( List( [ 1 .. nr_components ], function ( i )
              return
               PreCompose( source[i],
                 InjectionOfCofactorOfDirectSumWithGivenDirectSum(
                   diagram, i, direct_sum ) );
          end ) );
end;
```

Back to index

## Derivations for UniversalMorphismIntoFiberProduct

**UniversalMorphismIntoFiberProduct using the universality of the kernel representation of the pullback**

This derivation is for all categories. This derivation uses:

- AdditionForMorphisms × 1
- AdditiveInverseForMorphisms × 1
- KernelLift × 1
- PreCompose × 1
- UniversalMorphismIntoDirectSum × 1
- DirectSumDiagonalDifference × 1
- IsomorphismFromKernelOfDiagonalDifferenceToFiberProduct × 1

```
function ( diagram, source )
    local  test_function, direct_sum_diagonal_difference,
    kernel_lift;
    test_function := CallFuncList( UniversalMorphismIntoDirectSum,
       source );
    direct_sum_diagonal_difference := DirectSumDiagonalDifference(
       diagram );
    kernel_lift := KernelLift( direct_sum_diagonal_difference,
       test_function );
    return
     PreCompose( kernel_lift,
       IsomorphismFromKernelOfDiagonalDifferenceToFiberProduct(
         diagram ) );
end;
```

Back to index

## Derivations for UniversalMorphismIntoTerminalObject

**UniversalMorphismIntoTerminalObject computing the zero morphism**

This derivation is for additive categories. This derivation uses:

- ZeroMorphism × 1
- TerminalObject × 1

```
function ( test_source )
    local  terminal_object;
    terminal_object := TerminalObject( CapCategory( test_source ) );
    return ZeroMorphism( test_source, terminal_object );
end;
```

Back to index

**UniversalMorphismFromInitialObject using UniversalMorphismFromZeroObject**

This derivation is for all categories. This derivation uses:

- AdditionForMorphisms × 1
- AdditiveInverseForMorphisms × 1
- UniversalMorphismIntoZeroObject × 1
- IsomorphismFromZeroObjectToTerminalObject × 1
- PreCompose × 1

```
function ( obj )
    local  category;
    category := CapCategory( obj );
    return PreCompose( UniversalMorphismIntoZeroObject( obj ),
       IsomorphismFromZeroObjectToTerminalObject( category ) );
end;
```

Back to index

## Derivations for UniversalMorphismIntoTerminalObjectWithGivenTerminalObject

**UniversalMorphismIntoTerminalObject computing the zero morphism**

This derivation is for additive categories. This derivation uses:

- ZeroMorphism × 1
- TerminalObject × 1

```
function ( test_source, terminal_object )
    return ZeroMorphism( test_source, terminal_object );
end;
```

Back to index

## Derivations for UniversalMorphismIntoZeroObject

**UniversalMorphismIntoZeroObject computing the zero morphism**

This derivation is for additive categories. This derivation uses:

- ZeroMorphism × 1
- ZeroObject × 1

```
function ( test_source )
    local  zero_object;
    zero_object := ZeroObject( CapCategory( test_source ) );
    return ZeroMorphism( test_source, zero_object );
end;
```

Back to index

**UniversalMorphismIntoZeroObject using UniversalMorphismIntoTerminal-Object**

This derivation is for all categories. This derivation uses:

- AdditionForMorphisms × 1
- AdditiveInverseForMorphisms × 1
- IsomorphismFromTerminalObjectToZeroObject × 1
- UniversalMorphismIntoTerminalObject × 1
- PreCompose × 1

```
function ( obj )
    local  category;
    category := CapCategory( obj );
    return PreCompose( UniversalMorphismIntoTerminalObject( obj ),
       IsomorphismFromTerminalObjectToZeroObject( category ) );
end;
```

Back to index

## Derivations for UniversalMorphismIntoZeroObjectWithGivenZeroObject

**UniversalMorphismIntoZeroObject computing the zero morphism**

This derivation is for additive categories. This derivation uses:

- ZeroMorphism × 1
- ZeroObject × 1

```
function ( test_source, zero_object )
    return ZeroMorphism( test_source, zero_object );
end;
```

Back to index

## Derivations for UniversalPropertyOfDual

**UniversalPropertyOfDual using the hom tensor adjunction**

This derivation is for symmetric closed monoidal categories. This derivation uses:

- IsomorphismFromInternalHomToDual × 1
- PreCompose × 1
- TensorProductToInternalHomAdjunctionMap × 1

```
function ( object_1, object_2, test_morphism )
    local  adjoint_morphism;
    adjoint_morphism := TensorProductToInternalHomAdjunctionMap(
       object_1, object_2, test_morphism );
    return
     PreCompose( adjoint_morphism,
        IsomorphismFromInternalHomToDual( object_2 ) );
end;
```

Back to index

### Derivations for VerticalPostCompose

#### VerticalPostCompose using VerticalPreCompose

This derivation is for all categories. This derivation uses:

- VerticalPreCompose $\times 1$

```
function ( twocell_below, twocell_above )
    return VerticalPreCompose( twocell_above, twocell_below );
end;
```

Back to index

### Derivations for VerticalPreCompose

#### VerticalPreCompose using VerticalPostCompose

This derivation is for all categories. This derivation uses:

- VerticalPostCompose $\times 1$

```
function ( twocell_above, twocell_below )
    return VerticalPostCompose( twocell_below, twocell_above );
end;
```

Back to index

### Derivations for ZeroMorphism

#### Zero morphism by composition of morphism into and from zero object

This derivation is for additive categories. This derivation uses:

- PreCompose $\times 1$
- UniversalMorphismIntoZeroObject $\times 1$
- UniversalMorphismFromZeroObject $\times 1$

```
function ( obj_source, obj_range )
    return PreCompose( UniversalMorphismIntoZeroObject( obj_source )
        , UniversalMorphismFromZeroObject( obj_range ) );
end;
```

Back to index

APPENDIX E

# Final Derivations

## Final derivation index

- IsomorphismFromFiberProductToKernelOfDiagonalDifference
- IsomorphismFromKernelOfDiagonalDifferenceToFiberProduct
- IsomorphismFromPushoutToCokernelOfDiagonalDifference
- IsomorphismFromCokernelOfDiagonalDifferenceToPushout
- IsomorphismFromImageObjectToKernelOfCokernel
- IsomorphismFromKernelOfCokernelToImageObject
- IsomorphismFromCoimageToCokernelOfKernel
- IsomorphismFromCokernelOfKernelToCoimage
- IsomorphismFromInitialObjectToZeroObject
- IsomorphismFromZeroObjectToInitialObject
- IsomorphismFromTerminalObjectToZeroObject
- IsomorphismFromZeroObjectToTerminalObject
- IsomorphismFromDirectSumToDirectProduct
- IsomorphismFromDirectProductToDirectSum
- IsomorphismFromCoproductToDirectSum
- IsomorphismFromDirectSumToCoproduct
- IsEqualForObjects
- IsCongruentForMorphisms0
- IsEqualForMorphisms
- IsEqualForCacheForMorphisms
- IsomorphismFromTensorProductToInternalHom
- IsomorphismFromInternalHomToTensorProduct
- IsomorphismFromInternalHomToDual
- IsomorphismFromDualToInternalHom

## Final derivation for IsomorphismFromFiberProductToKernelOfDiagonalDifference

This final derivation is for all categories. This derivation uses:

- DirectSumDiagonalDifference $\times 1$
- KernelObject $\times 1$
- IdentityMorphism $\times 1$

This derivation can only be triggered if the following operations are not installed:

- ProjectionInFactorOfFiberProduct
- ProjectionInFactorOfFiberProductWithGivenFiberProduct
- UniversalMorphismIntoFiberProductWithGivenFiberProduct
- FiberProductEmbeddingInDirectSum
- IsomorphismFromFiberProductToKernelOfDiagonalDifference
- IsomorphismFromKernelOfDiagonalDifferenceToFiberProduct

```
function ( diagram )
    local  kernel_of_diagonal_difference;
    kernel_of_diagonal_difference
     := KernelObject( DirectSumDiagonalDifference( diagram ) );
    return IdentityMorphism( kernel_of_diagonal_difference );
end;
```

Back to index

## Final derivation for IsomorphismFromKernelOfDiagonalDifferenceToFiberProduct

This final derivation is for all categories. This derivation uses:

- DirectSumDiagonalDifference $\times 1$
- KernelObject $\times 1$
- IdentityMorphism $\times 1$

This derivation can only be triggered if the following operations are not installed:

- ProjectionInFactorOfFiberProduct
- ProjectionInFactorOfFiberProductWithGivenFiberProduct
- UniversalMorphismIntoFiberProductWithGivenFiberProduct
- FiberProductEmbeddingInDirectSum
- IsomorphismFromFiberProductToKernelOfDiagonalDifference
- IsomorphismFromKernelOfDiagonalDifferenceToFiberProduct

```
function ( diagram )
    local  kernel_of_diagonal_difference;
    kernel_of_diagonal_difference
     := KernelObject( DirectSumDiagonalDifference( diagram ) );
    return IdentityMorphism( kernel_of_diagonal_difference );
end;
```

Back to index

## Final derivation for IsomorphismFromPushoutToCokernelOfDiagonalDifference

This final derivation is for all categories. This derivation uses:

- CokernelObject $\times 1$

- DirectSumCodiagonalDifference × 1
- IdentityMorphism × 1

This derivation can only be triggered if the following operations are not installed:

- Pushout
- InjectionOfCofactorOfPushout
- InjectionOfCofactorOfPushoutWithGivenPushout
- UniversalMorphismFromPushoutWithGivenPushout
- DirectSumProjectionInPushout
- IsomorphismFromPushoutToCokernelOfDiagonalDifference
- IsomorphismFromCokernelOfDiagonalDifferenceToPushout

```
function ( diagram )
    local  cokernel_of_diagonal_difference;
    cokernel_of_diagonal_difference
     := CokernelObject( DirectSumCodiagonalDifference( diagram ) );
    return IdentityMorphism( cokernel_of_diagonal_difference );
end;
```

Back to index

## Final derivation for IsomorphismFromCokernelOfDiagonalDifferenceToPushout

This final derivation is for all categories. This derivation uses:

- CokernelObject × 1
- DirectSumCodiagonalDifference × 1
- IdentityMorphism × 1

This derivation can only be triggered if the following operations are not installed:

- Pushout
- InjectionOfCofactorOfPushout
- InjectionOfCofactorOfPushoutWithGivenPushout
- UniversalMorphismFromPushoutWithGivenPushout
- DirectSumProjectionInPushout
- IsomorphismFromPushoutToCokernelOfDiagonalDifference
- IsomorphismFromCokernelOfDiagonalDifferenceToPushout

```
function ( diagram )
    local  cokernel_of_diagonal_difference;
    cokernel_of_diagonal_difference
     := CokernelObject( DirectSumCodiagonalDifference( diagram ) );
    return IdentityMorphism( cokernel_of_diagonal_difference );
end;
```

Back to index

### Final derivation for IsomorphismFromImageObjectToKernelOfCokernel

This final derivation is for all categories. This derivation uses:

- KernelObject × 1
- CokernelProjection × 1
- IdentityMorphism × 1

This derivation can only be triggered if the following operations are not installed:

- ImageObject
- ImageEmbedding
- ImageEmbeddingWithGivenImageObject
- CoastrictionToImage
- CoastrictionToImageWithGivenImageObject
- UniversalMorphismFromImage
- UniversalMorphismFromImageWithGivenImageObject
- IsomorphismFromImageObjectToKernelOfCokernel
- IsomorphismFromKernelOfCokernelToImageObject

```
function ( mor )
    local  kernel_of_cokernel;
    kernel_of_cokernel := KernelObject( CokernelProjection( mor ) );
    return IdentityMorphism( kernel_of_cokernel );
end;
```

Back to index

### Final derivation for IsomorphismFromKernelOfCokernelToImageObject

This final derivation is for all categories. This derivation uses:

- KernelObject × 1
- CokernelProjection × 1
- IdentityMorphism × 1

This derivation can only be triggered if the following operations are not installed:

- ImageObject
- ImageEmbedding
- ImageEmbeddingWithGivenImageObject
- CoastrictionToImage
- CoastrictionToImageWithGivenImageObject
- UniversalMorphismFromImage
- UniversalMorphismFromImageWithGivenImageObject
- IsomorphismFromImageObjectToKernelOfCokernel
- IsomorphismFromKernelOfCokernelToImageObject

```
function ( mor )
    local  kernel_of_cokernel;
    kernel_of_cokernel := KernelObject( CokernelProjection( mor ) );
    return IdentityMorphism( kernel_of_cokernel );
end;
```

Back to index

### Final derivation for IsomorphismFromCoimageToCokernelOfKernel

This final derivation is for all categories. This derivation uses:

- CokernelObject $\times 1$
- KernelEmbedding $\times 1$
- IdentityMorphism $\times 1$

This derivation can only be triggered if the following operations are not installed:

- Coimage
- CoimageProjection
- CoimageProjectionWithGivenCoimage
- AstrictionToCoimage
- AstrictionToCoimageWithGivenCoimage
- UniversalMorphismIntoCoimage
- UniversalMorphismIntoCoimageWithGivenCoimage
- IsomorphismFromCoimageToCokernelOfKernel
- IsomorphismFromCokernelOfKernelToCoimage

```
function ( mor )
    local  cokernel_of_kernel;
    cokernel_of_kernel := CokernelObject( KernelEmbedding( mor ) );
    return IdentityMorphism( cokernel_of_kernel );
end;
```

Back to index

### Final derivation for IsomorphismFromCokernelOfKernelToCoimage

This final derivation is for all categories. This derivation uses:

- CokernelObject $\times 1$
- KernelEmbedding $\times 1$
- IdentityMorphism $\times 1$

This derivation can only be triggered if the following operations are not installed:

- Coimage
- CoimageProjection
- CoimageProjectionWithGivenCoimage
- AstrictionToCoimage
- AstrictionToCoimageWithGivenCoimage
- UniversalMorphismIntoCoimage

- UniversalMorphismIntoCoimageWithGivenCoimage
- IsomorphismFromCoimageToCokernelOfKernel
- IsomorphismFromCokernelOfKernelToCoimage

```
function ( mor )
    local  cokernel_of_kernel;
    cokernel_of_kernel := CokernelObject( KernelEmbedding( mor ) );
    return IdentityMorphism( cokernel_of_kernel );
end;
```

Back to index

### Final derivation for IsomorphismFromInitialObjectToZeroObject

This final derivation is for all categories. This derivation uses:

- ZeroObject $\times 1$
- IdentityMorphism $\times 1$

This derivation can only be triggered if the following operations are not installed:

- InitialObject
- UniversalMorphismFromInitialObject

```
function ( category )
    return IdentityMorphism( ZeroObject( category ) );
end;
```

Back to index

### Final derivation for IsomorphismFromZeroObjectToInitialObject

This final derivation is for all categories. This derivation uses:

- ZeroObject $\times 1$
- IdentityMorphism $\times 1$

This derivation can only be triggered if the following operations are not installed:

- InitialObject
- UniversalMorphismFromInitialObject

```
function ( category )
    return IdentityMorphism( ZeroObject( category ) );
end;
```

Back to index

### Final derivation for IsomorphismFromTerminalObjectToZeroObject

This final derivation is for all categories. This derivation uses:

- ZeroObject × 1
- IdentityMorphism × 1

This derivation can only be triggered if the following operations are not installed:

- TerminalObject
- UniversalMorphismIntoTerminalObject

```
function ( category )
    return IdentityMorphism( ZeroObject( category ) );
end;
```

Back to index

### Final derivation for IsomorphismFromZeroObjectToTerminalObject

This final derivation is for all categories. This derivation uses:

- ZeroObject × 1
- IdentityMorphism × 1

This derivation can only be triggered if the following operations are not installed:

- TerminalObject
- UniversalMorphismIntoTerminalObject

```
function ( category )
    return IdentityMorphism( ZeroObject( category ) );
end;
```

Back to index

### Final derivation for IsomorphismFromDirectSumToDirectProduct

This final derivation is for all categories. This derivation uses:

- IdentityMorphism × 1

This derivation can only be triggered if the following operations are not installed:

- DirectProductFunctorialWithGivenDirectProducts
- ProjectionInFactorOfDirectProduct

```
function ( diagram )
    return IdentityMorphism( DirectSum( diagram ) );
end;
```

Back to index

### Final derivation for IsomorphismFromDirectProductToDirectSum

This final derivation is for all categories. This derivation uses:

- IdentityMorphism × 1

This derivation can only be triggered if the following operations are not installed:

- DirectProductFunctorialWithGivenDirectProducts
- ProjectionInFactorOfDirectProduct

```
function ( diagram )
    return IdentityMorphism( DirectSum( diagram ) );
end;
```

Back to index

### Final derivation for IsomorphismFromCoproductToDirectSum

This final derivation is for all categories. This derivation uses:

- IdentityMorphism × 1

This derivation can only be triggered if the following operations are not installed:

- Coproduct
- CoproductFunctorialWithGivenCoproducts
- InjectionOfCofactorOfCoproduct

```
function ( diagram )
    return IdentityMorphism( DirectSum( diagram ) );
end;
```

Back to index

### Final derivation for IsomorphismFromDirectSumToCoproduct

This final derivation is for all categories. This derivation uses:

- IdentityMorphism × 1

This derivation can only be triggered if the following operations are not installed:

- Coproduct
- CoproductFunctorialWithGivenCoproducts
- InjectionOfCofactorOfCoproduct

```
function ( diagram )
    return IdentityMorphism( DirectSum( diagram ) );
end;
```

Back to index

### Final derivation for IsEqualForObjects

This final derivation is for all categories. This derivation can only be triggered if the following operations are not installed:

- IsEqualForObjects

```
ReturnFail
```

Back to index

### Final derivation for IsCongruentForMorphisms

This final derivation is for all categories. This derivation can only be triggered if the following operations are not installed:

- IsCongruentForMorphisms
- IsEqualForMorphisms

```
ReturnFail
```

Back to index

### Final derivation for IsCongruentForMorphisms

This final derivation is for all categories. This derivation uses:

- IsEqualForMorphisms $\times 1$

This derivation can only be triggered if the following operations are not installed:

- IsCongruentForMorphisms

```
IsEqualForMorphisms
```

Back to index

### Final derivation for IsEqualForMorphisms

This final derivation is for all categories. This derivation can only be triggered if the following operations are not installed:

- IsCongruentForMorphisms
- IsEqualForMorphisms

```
ReturnFail
```

Back to index

### Final derivation for IsEqualForMorphisms

This final derivation is for all categories. This derivation uses:

- IsCongruentForMorphisms $\times 1$

This derivation can only be triggered if the following operations are not installed:

- IsEqualForMorphisms

```
IsCongruentForMorphisms
```

Back to index

### Final derivation for IsEqualForCacheForMorphisms

This final derivation is for all categories. This derivation uses:

- IsEqualForMorphismsOnMor $\times 1$

```
function ( mor1, mor2 )
    return IsEqualForMorphismsOnMor( mor1, mor2 ) = true;
end;
```

Back to index

### Final derivation for IsEqualForCacheForMorphisms

This final derivation is for all categories. This derivation uses:

- IsCongruentForMorphisms $\times 1$

This derivation can only be triggered if the following operations are not installed:

- IsEqualForMorphisms

```
ReturnFail
```

Back to index

### Final derivation for IsomorphismFromTensorProductToInternalHom

This final derivation is for rigid symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism $\times 1$
- DualOnObjects $\times 1$
- TensorProductOnObjects $\times 1$

This derivation can only be triggered if the following operations are not installed:

- InternalHomOnObjects
- InternalHomOnMorphismsWithGivenInternalHoms
- EvaluationMorphismWithGivenSource
- CoevaluationMorphismWithGivenRange
- TensorProductToInternalHomAdjunctionMap
- InternalHomToTensorProductAdjunctionMap

- MonoidalPreComposeMorphismWithGivenObjects
- MonoidalPostComposeMorphismWithGivenObjects
- TensorProductInternalHomCompatibilityMorphismWithGivenObjects
- TensorProductDualityCompatibilityMorphismWithGivenObjects
- MorphismFromTensorProductToInternalHomWithGivenObjects
- MorphismFromInternalHomToTensorProductWithGivenObjects
- IsomorphismFromTensorProductToInternalHom
- IsomorphismFromInternalHomToTensorProduct

```
function ( object_1, object_2 )
    return
     IdentityMorphism(
       TensorProductOnObjects( DualOnObjects( object_1 ), object_2
         ) );
end;
```

Back to index

### Final derivation for IsomorphismFromInternalHomToTensorProduct

This final derivation is for rigid symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism $\times 1$
- DualOnObjects $\times 1$
- TensorProductOnObjects $\times 1$

This derivation can only be triggered if the following operations are not installed:

- InternalHomOnObjects
- InternalHomOnMorphismsWithGivenInternalHoms
- EvaluationMorphismWithGivenSource
- CoevaluationMorphismWithGivenRange
- TensorProductToInternalHomAdjunctionMap
- InternalHomToTensorProductAdjunctionMap
- MonoidalPreComposeMorphismWithGivenObjects
- MonoidalPostComposeMorphismWithGivenObjects
- TensorProductInternalHomCompatibilityMorphismWithGivenObjects
- TensorProductDualityCompatibilityMorphismWithGivenObjects
- MorphismFromTensorProductToInternalHomWithGivenObjects
- MorphismFromInternalHomToTensorProductWithGivenObjects
- IsomorphismFromTensorProductToInternalHom
- IsomorphismFromInternalHomToTensorProduct

```
function ( object_1, object_2 )
    return
     IdentityMorphism(
       TensorProductOnObjects( DualOnObjects( object_1 ), object_2
         ) );
end;
```

Back to index

### Final derivation for IsomorphismFromInternalHomToDual

This final derivation is for rigid symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism × 1
- DualOnObjects × 1
- TensorProductOnObjects × 1

This derivation can only be triggered if the following operations are not installed:

- InternalHomOnObjects
- InternalHomOnMorphismsWithGivenInternalHoms
- EvaluationMorphismWithGivenSource
- CoevaluationMorphismWithGivenRange
- TensorProductToInternalHomAdjunctionMap
- InternalHomToTensorProductAdjunctionMap
- MonoidalPreComposeMorphismWithGivenObjects
- MonoidalPostComposeMorphismWithGivenObjects
- TensorProductInternalHomCompatibilityMorphismWithGivenObjects
- TensorProductDualityCompatibilityMorphismWithGivenObjects
- MorphismFromTensorProductToInternalHomWithGivenObjects
- MorphismFromInternalHomToTensorProductWithGivenObjects
- IsomorphismFromTensorProductToInternalHom
- IsomorphismFromInternalHomToTensorProduct

```
function ( object )
    return IdentityMorphism( DualOnObjects( object ) );
end;
```

Back to index

### Final derivation for IsomorphismFromInternalHomToDual

This final derivation is for symmetric monoidal categories. This derivation uses:

- IdentityMorphism × 1
- InternalHomOnObjects × 1
- TensorUnit × 1

This derivation can only be triggered if the following operations are not installed:

- DualOnObjects
- DualOnMorphismsWithGivenDuals

- MorphismToBidualWithGivenBidual
- MorphismFromBidualWithGivenBidual
- IsomorphismFromDualToInternalHom
- IsomorphismFromInternalHomToDual
- UniversalPropertyOfDual
- TensorProductDualityCompatibilityMorphismWithGivenObjects
- EvaluationForDualWithGivenTensorProduct
- CoevaluationForDualWithGivenTensorProduct
- MorphismFromInternalHomToTensorProductWithGivenObjects
- MorphismFromTensorProductToInternalHomWithGivenObjects

```
function ( object )
    local  category;
    category := CapCategory( object );
    return
     IdentityMorphism(
       InternalHomOnObjects( object, TensorUnit( category ) ) );
end;
```

Back to index

## Final derivation for IsomorphismFromDualToInternalHom

This final derivation is for rigid symmetric closed monoidal categories. This derivation uses:

- IdentityMorphism $\times 1$
- DualOnObjects $\times 1$
- TensorProductOnObjects $\times 1$

This derivation can only be triggered if the following operations are not installed:

- InternalHomOnObjects
- InternalHomOnMorphismsWithGivenInternalHoms
- EvaluationMorphismWithGivenSource
- CoevaluationMorphismWithGivenRange
- TensorProductToInternalHomAdjunctionMap
- InternalHomToTensorProductAdjunctionMap
- MonoidalPreComposeMorphismWithGivenObjects
- MonoidalPostComposeMorphismWithGivenObjects
- TensorProductInternalHomCompatibilityMorphismWithGivenObjects
- TensorProductDualityCompatibilityMorphismWithGivenObjects
- MorphismFromTensorProductToInternalHomWithGivenObjects
- MorphismFromInternalHomToTensorProductWithGivenObjects
- IsomorphismFromTensorProductToInternalHom
- IsomorphismFromInternalHomToTensorProduct

```
function ( object )
    return IdentityMorphism( DualOnObjects( object ) );
end;
```

Back to index

## Final derivation for IsomorphismFromDualToInternalHom

This final derivation is for symmetric monoidal categories. This derivation uses:

- IdentityMorphism $\times 1$
- InternalHomOnObjects $\times 1$
- TensorUnit $\times 1$

This derivation can only be triggered if the following operations are not installed:

- DualOnObjects
- DualOnMorphismsWithGivenDuals
- MorphismToBidualWithGivenBidual
- MorphismFromBidualWithGivenBidual
- IsomorphismFromDualToInternalHom
- IsomorphismFromInternalHomToDual
- UniversalPropertyOfDual
- TensorProductDualityCompatibilityMorphismWithGivenObjects
- EvaluationForDualWithGivenTensorProduct
- CoevaluationForDualWithGivenTensorProduct
- MorphismFromInternalHomToTensorProductWithGivenObjects
- MorphismFromTensorProductToInternalHomWithGivenObjects

```
function ( object )
    local  category;
    category := CapCategory( object );
    return
     IdentityMorphism(
       InternalHomOnObjects( object, TensorUnit( category ) ) );
end;
```

Back to index

APPENDIX F

# Installed basic operations

## 1. Primitive operation index

### 1.a. Primitive operations for left module presentations.

(1) IdentityMorphism
(2) KernelEmbedding
(3) CokernelProjection
(4) ZeroObject
(5) TensorUnit
(6) Lift
(7) KernelEmbeddingWithGivenKernelObject
(8) CokernelProjectionWithGivenCokernelObject
(9) CokernelColiftWithGivenCokernelObject
(10) PreCompose
(11) UniversalMorphismFromZeroObjectWithGivenZeroObject
(12) UniversalMorphismIntoZeroObjectWithGivenZeroObject
(13) ZeroMorphism
(14) DirectSum
(15) ProjectionInFactorOfDirectSumWithGivenDirectSum
(16) UniversalMorphismIntoDirectSumWithGivenDirectSum
(17) InjectionOfCofactorOfDirectSumWithGivenDirectSum
(18) UniversalMorphismFromDirectSumWithGivenDirectSum
(19) IsCongruentForMorphisms
(20) IsEqualForMorphisms
(21) IsEqualForObjects
(22) IsEqualForCacheForObjects
(23) IsEqualForCacheForMorphisms
(24) AdditionForMorphisms
(25) AdditiveInverseForMorphisms
(26) IsWellDefinedForMorphisms
(27) IsWellDefinedForObjects
(28) TensorProductOnObjects
(29) TensorProductOnMorphismsWithGivenTensorProducts
(30) BraidingWithGivenTensorProducts
(31) InternalHomOnObjects
(32) InternalHomOnMorphismsWithGivenInternalHoms

(33) EvaluationMorphismWithGivenSource
(34) CoevaluationMorphismWithGivenRange

### 1.b. Primitive operations for right module presentations.

(1) IdentityMorphism
(2) KernelEmbedding
(3) CokernelProjection
(4) ZeroObject
(5) TensorUnit
(6) Lift
(7) KernelEmbeddingWithGivenKernelObject
(8) CokernelProjectionWithGivenCokernelObject
(9) CokernelColiftWithGivenCokernelObject
(10) PreCompose
(11) UniversalMorphismFromZeroObjectWithGivenZeroObject
(12) UniversalMorphismIntoZeroObjectWithGivenZeroObject
(13) ZeroMorphism
(14) DirectSum
(15) ProjectionInFactorOfDirectSumWithGivenDirectSum
(16) UniversalMorphismIntoDirectSumWithGivenDirectSum
(17) InjectionOfCofactorOfDirectSumWithGivenDirectSum
(18) UniversalMorphismFromDirectSumWithGivenDirectSum
(19) IsCongruentForMorphisms
(20) IsEqualForMorphisms
(21) IsEqualForObjects
(22) IsEqualForCacheForObjects
(23) IsEqualForCacheForMorphisms
(24) AdditionForMorphisms
(25) AdditiveInverseForMorphisms
(26) IsWellDefinedForMorphisms
(27) IsWellDefinedForObjects
(28) TensorProductOnObjects
(29) TensorProductOnMorphismsWithGivenTensorProducts
(30) BraidingWithGivenTensorProducts
(31) InternalHomOnObjects
(32) InternalHomOnMorphismsWithGivenInternalHoms
(33) EvaluationMorphismWithGivenSource
(34) CoevaluationMorphismWithGivenRange

### 1.c. Primitive operations for graded left module presentations.

(1) IdentityMorphism
(2) KernelEmbedding
(3) CokernelObject

(4) ZeroObject
(5) TensorUnit
(6) Lift
(7) KernelEmbeddingWithGivenKernelObject
(8) CokernelProjectionWithGivenCokernelObject
(9) CokernelColiftWithGivenCokernelObject
(10) PreCompose
(11) UniversalMorphismFromZeroObjectWithGivenZeroObject
(12) UniversalMorphismIntoZeroObjectWithGivenZeroObject
(13) ZeroMorphism
(14) DirectSum
(15) ProjectionInFactorOfDirectSumWithGivenDirectSum
(16) UniversalMorphismIntoDirectSumWithGivenDirectSum
(17) InjectionOfCofactorOfDirectSumWithGivenDirectSum
(18) UniversalMorphismFromDirectSumWithGivenDirectSum
(19) IsCongruentForMorphisms
(20) IsEqualForMorphisms
(21) IsEqualForObjects
(22) IsEqualForCacheForObjects
(23) IsEqualForCacheForMorphisms
(24) AdditionForMorphisms
(25) AdditiveInverseForMorphisms
(26) IsWellDefinedForMorphisms
(27) IsWellDefinedForObjects
(28) TensorProductOnObjects
(29) TensorProductOnMorphismsWithGivenTensorProducts
(30) InternalHomOnObjects
(31) InternalHomOnMorphismsWithGivenInternalHoms

### 1.d. Primitive operations for graded right module presentations.

(1) IdentityMorphism
(2) KernelEmbedding
(3) CokernelObject
(4) ZeroObject
(5) TensorUnit
(6) Lift
(7) KernelEmbeddingWithGivenKernelObject
(8) CokernelProjectionWithGivenCokernelObject
(9) CokernelColiftWithGivenCokernelObject
(10) PreCompose
(11) UniversalMorphismFromZeroObjectWithGivenZeroObject
(12) UniversalMorphismIntoZeroObjectWithGivenZeroObject
(13) ZeroMorphism

(14) DirectSum
(15) ProjectionInFactorOfDirectSumWithGivenDirectSum
(16) UniversalMorphismIntoDirectSumWithGivenDirectSum
(17) InjectionOfCofactorOfDirectSumWithGivenDirectSum
(18) UniversalMorphismFromDirectSumWithGivenDirectSum
(19) IsCongruentForMorphisms
(20) IsEqualForMorphisms
(21) IsEqualForObjects
(22) IsEqualForCacheForObjects
(23) IsEqualForCacheForMorphisms
(24) AdditionForMorphisms
(25) AdditiveInverseForMorphisms
(26) IsWellDefinedForMorphisms
(27) IsWellDefinedForObjects
(28) TensorProductOnObjects
(29) TensorProductOnMorphismsWithGivenTensorProducts

### 1.e. Primitive operations for generalized morphisms by cospans.

(1) IdentityMorphism
(2) PreCompose
(3) ZeroMorphism
(4) IsCongruentForMorphisms
(5) IsEqualForObjects
(6) IsEqualForCacheForObjects
(7) IsEqualForCacheForMorphisms
(8) AdditionForMorphisms
(9) AdditiveInverseForMorphisms
(10) IsWellDefinedForMorphisms
(11) IsWellDefinedForObjects

### 1.f. Primitive operations for generalized morphisms by spans.

(1) IdentityMorphism
(2) PreCompose
(3) ZeroMorphism
(4) IsCongruentForMorphisms
(5) IsEqualForObjects
(6) IsEqualForCacheForObjects
(7) IsEqualForCacheForMorphisms
(8) AdditionForMorphisms
(9) AdditiveInverseForMorphisms
(10) IsWellDefinedForMorphisms
(11) IsWellDefinedForObjects

### 1.g. Primitive operations for generalized morphisms by three arrows.

(1) IdentityMorphism
(2) PreCompose
(3) IsCongruentForMorphisms
(4) IsEqualForObjects
(5) IsEqualForCacheForObjects
(6) IsEqualForCacheForMorphisms
(7) AdditionForMorphisms
(8) IsWellDefinedForMorphisms
(9) IsWellDefinedForObjects

### 1.h. Primitive operations for Serre quotient by cospans.

(1) IdentityMorphism
(2) KernelEmbedding
(3) CokernelProjection
(4) ZeroObject
(5) LiftAlongMonomorphism
(6) ColiftAlongEpimorphism
(7) PreCompose
(8) ZeroMorphism
(9) DirectSum
(10) ProjectionInFactorOfDirectSumWithGivenDirectSum
(11) UniversalMorphismIntoDirectSum
(12) InjectionOfCofactorOfDirectSumWithGivenDirectSum
(13) UniversalMorphismFromDirectSum
(14) IsCongruentForMorphisms
(15) IsEqualForObjects
(16) AdditionForMorphisms
(17) AdditiveInverseForMorphisms
(18) IsZeroForObjects

### 1.i. Primitive operations for Serre quotient by spans.

(1) InverseImmutable
(2) IdentityMorphism
(3) KernelEmbedding
(4) CokernelProjection
(5) ZeroObject
(6) DualOnObjects
(7) LiftAlongMonomorphism
(8) ColiftAlongEpimorphism
(9) Lift
(10) PreCompose
(11) ZeroMorphism

(12) DirectSum
(13) ProjectionInFactorOfDirectSumWithGivenDirectSum
(14) UniversalMorphismIntoDirectSum
(15) InjectionOfCofactorOfDirectSumWithGivenDirectSum
(16) UniversalMorphismFromDirectSum
(17) IsCongruentForMorphisms
(18) IsEqualForObjects
(19) IsEqualForCacheForObjects
(20) IsEqualForCacheForMorphisms
(21) AdditionForMorphisms
(22) AdditiveInverseForMorphisms
(23) IsZeroForObjects
(24) DualOnMorphismsWithGivenDuals

### 1.j. Primitive operations for Serre quotient by three arrows.

(1) IdentityMorphism
(2) KernelEmbedding
(3) CokernelProjection
(4) ZeroObject
(5) LiftAlongMonomorphism
(6) ColiftAlongEpimorphism
(7) PreCompose
(8) ZeroMorphism
(9) DirectSum
(10) ProjectionInFactorOfDirectSumWithGivenDirectSum
(11) UniversalMorphismIntoDirectSum
(12) InjectionOfCofactorOfDirectSumWithGivenDirectSum
(13) UniversalMorphismFromDirectSum
(14) IsCongruentForMorphisms
(15) IsEqualForObjects
(16) IsZeroForMorphisms
(17) AdditionForMorphisms
(18) AdditiveInverseForMorphisms
(19) IsZeroForObjects

## 2. Primitive operations for left module presentations

**IdentityMorphism.** Back to index.

```
function ( object )
    local  matrix;
    matrix
     := HomalgIdentityMatrix( NrColumns( UnderlyingMatrix( object )
         ), homalg_ring );
    return PresentationMorphism( object, matrix, object );
end;
```

**KernelEmbedding.** Back to index.

```
function ( morphism )
    local  kernel, embedding;
    embedding := SyzygiesOfRows( UnderlyingMatrix( morphism ),
       UnderlyingMatrix( Range( morphism ) ) );
    kernel
     := SyzygiesOfRows( embedding,
       UnderlyingMatrix( Source( morphism ) ) );
    kernel := AsLeftPresentation( kernel );
    return PresentationMorphism( kernel, embedding,
       Source( morphism ) );
end;
```

**CokernelProjection.** Back to index.

```
function ( morphism )
    local  cokernel_object, projection;
    cokernel_object := UnionOfRows( UnderlyingMatrix( morphism ),
       UnderlyingMatrix( Range( morphism ) ) );
    cokernel_object := AsLeftPresentation( cokernel_object );
    projection
     := HomalgIdentityMatrix(
       NrColumns( UnderlyingMatrix( Range( morphism ) ) ),
       homalg_ring );
    return PresentationMorphism( Range( morphism ), projection,
       cokernel_object );
end;
```

**ZeroObject.** Back to index.

```
function (  )
    local  matrix;
    matrix := HomalgZeroMatrix( 0, 0, homalg_ring );
    return AsLeftPresentation( matrix );
end;
```

**TensorUnit.** Back to index.

```
function (  )
    return
```

```
     AsLeftPresentation( HomalgZeroMatrix( 0, 1, homalg_ring ) );
end;
```

**Lift.** Back to index.

```
function ( alpha, beta )
    local  lift;
    lift := RightDivide( UnderlyingMatrix( alpha ),
       UnderlyingMatrix( beta ), UnderlyingMatrix( Range( beta ) ) );
    if lift = fail  then
        return fail;
    fi;
    return PresentationMorphism( Source( alpha ), lift,
       Source( beta ) );
end;
```

**KernelEmbeddingWithGivenKernelObject.** Back to index.

```
function ( morphism, kernel )
    local  embedding;
    embedding := SyzygiesOfRows( UnderlyingMatrix( morphism ),
       UnderlyingMatrix( Range( morphism ) ) );
    return PresentationMorphism( kernel, embedding,
       Source( morphism ) );
end;
```

**CokernelProjectionWithGivenCokernelObject.** Back to index.

```
function ( morphism, cokernel_object )
    local  projection;
    projection
     := HomalgIdentityMatrix(
       NrColumns( UnderlyingMatrix( Range( morphism ) ) ),
       homalg_ring );
    return PresentationMorphism( Range( morphism ), projection,
       cokernel_object );
end;
```

**CokernelColiftWithGivenCokernelObject.** Back to index.

```
function ( morphism, test_morphism, cokernel_object )
    return PresentationMorphism( cokernel_object,
       UnderlyingMatrix( test_morphism ), Range( test_morphism ) );
end;
```

**PreCompose.** Back to index.

```
function ( left_morphism, right_morphism )
    return PresentationMorphism( Source( left_morphism ),
       UnderlyingMatrix( left_morphism )
        * UnderlyingMatrix( right_morphism ),
       Range( right_morphism ) );
end;
```

```
function ( left_morphism, identity_morphism )
    return left_morphism;
end;
```

This function uses the following extra filters:

- IsIdenticalToIdentityMorphism for the 2nd argument.

```
function ( identity_morphism, right_morphism )
    return right_morphism;
end;
```

This function uses the following extra filters:

- IsIdenticalToIdentityMorphism for the 1st argument.

```
function ( left_morphism, zero_morphism )
    return PresentationMorphism( Source( left_morphism ),
       HomalgZeroMatrix( NrRows( UnderlyingMatrix( left_morphism ) )
          , NrColumns( UnderlyingMatrix( zero_morphism ) ),
         homalg_ring ), Range( zero_morphism ) );
end;
```

This function uses the following extra filters:

- IsIdenticalToZeroMorphism for the 2nd argument.

```
function ( zero_morphism, right_morphism )
    return PresentationMorphism( Source( zero_morphism ),
       HomalgZeroMatrix( NrRows( UnderlyingMatrix( zero_morphism ) )
          , NrColumns( UnderlyingMatrix( right_morphism ) ),
         homalg_ring ), Range( right_morphism ) );
end;
```

This function uses the following extra filters:

- IsIdenticalToZeroMorphism for the 1st argument.

**UniversalMorphismFromZeroObjectWithGivenZeroObject.** Back to index.

```
function ( object, initial_object )
    local  nr_columns, morphism;
    nr_columns := NrColumns( UnderlyingMatrix( object ) );
    morphism := HomalgZeroMatrix( 0, nr_columns, homalg_ring );
    return PresentationMorphism( initial_object, morphism, object );
end;
```

**UniversalMorphismIntoZeroObjectWithGivenZeroObject.** Back to index.

```
function ( object, terminal_object )
    local  nr_columns, morphism;
    nr_columns := NrColumns( UnderlyingMatrix( object ) );
    morphism := HomalgZeroMatrix( nr_columns, 0, homalg_ring );
    return PresentationMorphism( object, morphism, terminal_object );
end;
```

**ZeroMorphism.** Back to index.

```
function ( source, range )
    local  matrix;
    matrix
     := HomalgZeroMatrix( NrColumns( UnderlyingMatrix( source ) ),
       NrColumns( UnderlyingMatrix( range ) ), homalg_ring );
    return PresentationMorphism( source, matrix, range );
end;
```

**DirectSum.** Back to index.

```
function ( product_object )
    local  objects, direct_sum;
    objects := product_object;
    objects := List( objects, UnderlyingMatrix );
    direct_sum := DiagMat( objects );
    return AsLeftPresentation( direct_sum );
end;
```

**ProjectionInFactorOfDirectSumWithGivenDirectSum.** Back to index.

```
function ( product_object, component_number, direct_sum_object )
    local  objects, object_column_dimension, dimension_of_factor,
    projection, projection_matrix, i;
    objects := product_object;
    object_column_dimension := List( objects, function ( i )
            return NrColumns( UnderlyingMatrix( i ) );
        end );
    dimension_of_factor := object_column_dimension[component_number];
    projection := List( object_column_dimension, function ( i )
            return HomalgZeroMatrix( i, dimension_of_factor,
               homalg_ring );
        end );
    projection[component_number]
     := HomalgIdentityMatrix(
       object_column_dimension[component_number], homalg_ring );
    projection_matrix := projection[1];
    for i  in [ 2 .. Length( objects ) ]  do
        projection_matrix := UnionOfRows( projection_matrix,
           projection[i] );
    od;
    return PresentationMorphism( direct_sum_object,
       projection_matrix, objects[component_number] );
end;
```

**UniversalMorphismIntoDirectSumWithGivenDirectSum.** Back to index.

```
function ( diagram, product_morphism, direct_sum )
    local  components, number_of_components, map_into_product, i;
    components := product_morphism;
    number_of_components := Length( components );
    map_into_product := UnderlyingMatrix( components[1] );
    for i  in [ 2 .. number_of_components ]  do
        map_into_product := UnionOfColumns( map_into_product,
           UnderlyingMatrix( components[i] ) );
    od;
    return PresentationMorphism( Source( components[1] ),
       map_into_product, direct_sum );
end;
```

**InjectionOfCofactorOfDirectSumWithGivenDirectSum.** Back to index.

```
function ( product_object, component_number, direct_sum_object )
    local  objects, object_column_dimension, dimension_of_cofactor,
    injection, injection_matrix, i;
    objects := product_object;
    object_column_dimension := List( objects, function ( i )
            return NrColumns( UnderlyingMatrix( i ) );
        end );
    dimension_of_cofactor
     := object_column_dimension[component_number];
    injection := List( object_column_dimension, function ( i )
            return HomalgZeroMatrix( dimension_of_cofactor, i,
               homalg_ring );
        end );
    injection[component_number]
     := HomalgIdentityMatrix(
       object_column_dimension[component_number], homalg_ring );
    injection_matrix := injection[1];
    for i  in [ 2 .. Length( objects ) ]  do
        injection_matrix := UnionOfColumns( injection_matrix,
           injection[i] );
    od;
    return PresentationMorphism( objects[component_number],
       injection_matrix, direct_sum_object );
end;
```

**UniversalMorphismFromDirectSumWithGivenDirectSum.** Back to index.

```
function ( diagram, product_morphism, direct_sum )
    local  components, number_of_components, map_into_product, i;
    components := product_morphism;
    number_of_components := Length( components );
    map_into_product := UnderlyingMatrix( components[1] );
    for i  in [ 2 .. number_of_components ]  do
        map_into_product := UnionOfRows( map_into_product,
           UnderlyingMatrix( components[i] ) );
    od;
    return PresentationMorphism( direct_sum, map_into_product,
       Range( components[1] ) );
end;
```

**IsCongruentForMorphisms.** Back to index.

```
function ( morphism_1, morphism_2 )
    local  result_of_divide;
    result_of_divide
     := DecideZeroRows( UnderlyingMatrix( morphism_1 )
        - UnderlyingMatrix( morphism_2 ),
       UnderlyingMatrix( Range( morphism_1 ) ) );
    return IsZero( result_of_divide );
end;
```

**IsEqualForMorphisms.** Back to index.

```
function ( morphism_1, morphism_2 )
    return UnderlyingMatrix( morphism_1 )
```

```
      = UnderlyingMatrix( morphism_2 );
end;
```

**IsEqualForObjects.** Back to index.

```
function ( object1, object2 )
    return UnderlyingMatrix( object1 ) = UnderlyingMatrix( object2 );
end;
```

**IsEqualForCacheForObjects.** Back to index.

```
IsIdenticalObj
```

**IsEqualForCacheForMorphisms.** Back to index.

```
IsIdenticalObj
```

**AdditionForMorphisms.** Back to index.

```
function ( morphism_1, morphism_2 )
    return PresentationMorphism( Source( morphism_1 ),
       UnderlyingMatrix( morphism_1 )
        + UnderlyingMatrix( morphism_2 ), Range( morphism_1 ) );
end;
```

**AdditiveInverseForMorphisms.** Back to index.

```
function ( morphism_1 )
    return PresentationMorphism( Source( morphism_1 ),
       - UnderlyingMatrix( morphism_1 ), Range( morphism_1 ) );
end;
```

**IsWellDefinedForMorphisms.** Back to index.

```
function ( morphism )
    local  source_matrix, range_matrix, morphism_matrix;
    source_matrix := UnderlyingMatrix( Source( morphism ) );
    range_matrix := UnderlyingMatrix( Range( morphism ) );
    morphism_matrix := UnderlyingMatrix( morphism );
    if
     not (NrColumns( source_matrix ) = NrRows( morphism_matrix )
           and NrColumns( morphism_matrix )
             = NrColumns( range_matrix ))  then
        return false;
    fi;
    if RightDivide( source_matrix * morphism_matrix, range_matrix )
        = fail  then
        return false;
    fi;
    return true;
end;
```

**IsWellDefinedForObjects.** Back to index.

```
function ( object )
    return IsHomalgMatrix( UnderlyingMatrix( object ) )
      and IsHomalgRing( UnderlyingHomalgRing( object ) );
end;
```

**TensorProductOnObjects.** Back to index.

```
function ( object_1, object_2 )
    local  identity_1, identity_2, presentation_matrix_1,
    presentation_matrix_2, presentation_matrix;
    presentation_matrix_1 := UnderlyingMatrix( object_1 );
    presentation_matrix_2 := UnderlyingMatrix( object_2 );
    identity_1
     := HomalgIdentityMatrix( NrColumns( presentation_matrix_1 ),
       homalg_ring );
    identity_2
     := HomalgIdentityMatrix( NrColumns( presentation_matrix_2 ),
       homalg_ring );
    presentation_matrix
     :=
      UnionOfRows( KroneckerMat( identity_1, presentation_matrix_2 )
        , KroneckerMat( presentation_matrix_1, identity_2 ) );
    return AsLeftPresentation( presentation_matrix );
end;
```

**TensorProductOnMorphismsWithGivenTensorProducts.** Back to index.

```
function ( new_source, morphism_1, morphism_2, new_range )
    return PresentationMorphism( new_source,
       KroneckerMat( UnderlyingMatrix( morphism_1 ),
         UnderlyingMatrix( morphism_2 ) ), new_range );
end;
```

**BraidingWithGivenTensorProducts.** Back to index.

```
function ( object_1_tensored_object_2, object_1, object_2,
    object_2_tensored_object_1 )
    local  homalg_ring, permutation_matrix, rank_1, rank_2, rank;
    homalg_ring := UnderlyingHomalgRing( object_1 );
    rank_1 := NrColumns( UnderlyingMatrix( object_1 ) );
    rank_2 := NrColumns( UnderlyingMatrix( object_2 ) );
    rank
     := NrColumns( UnderlyingMatrix( object_1_tensored_object_2 ) );
    permutation_matrix
     := PermutationMat(
       PermList( List( [ 1 .. rank ], function ( i )
                return
                 RemInt( (i - 1), rank_2 ) * rank_1
                    + QuoInt( (i - 1), rank_2 ) + 1;
            end ) ), rank );
    return PresentationMorphism( object_1_tensored_object_2,
       HomalgMatrix( permutation_matrix, rank, rank, homalg_ring ),
       object_2_tensored_object_1 );
end;
```

**InternalHomOnObjects.** Back to index.

```
function ( object_1, object_2 )
    return
     Source( INTERNAL_HOM_EMBEDDING_IN_TENSOR_PRODUCT_LEFT(
         object_1, object_2 ) );
end;
```

**InternalHomOnMorphismsWithGivenInternalHoms.** Back to index.

```
function ( new_source, morphism_1, morphism_2, new_range )
    local  internal_hom_embedding_source,
    internal_hom_embedding_range, morphism_between_tensor_products;
    internal_hom_embedding_source
     := INTERNAL_HOM_EMBEDDING_IN_TENSOR_PRODUCT_LEFT(
       Range( morphism_1 ), Source( morphism_2 ) );
    internal_hom_embedding_range
     := INTERNAL_HOM_EMBEDDING_IN_TENSOR_PRODUCT_LEFT(
       Source( morphism_1 ), Range( morphism_2 ) );
    morphism_between_tensor_products
     := PresentationMorphism( Range( internal_hom_embedding_source )
        , KroneckerMat( Involution( UnderlyingMatrix( morphism_1 ) )
          , UnderlyingMatrix( morphism_2 ) ),
       Range( internal_hom_embedding_range ) );
    return LiftAlongMonomorphism( internal_hom_embedding_range,
       PreCompose( internal_hom_embedding_source,
         morphism_between_tensor_products ) );
end;
```

**EvaluationMorphismWithGivenSource.** Back to index.

```
function ( object_1, object_2, internal_hom_tensored_object_1 )
    local  internal_hom_embedding, rank_1, morphism, free_module,
    column, zero_column, i, matrix, rank_2, lifted_evaluation;
    internal_hom_embedding
     := INTERNAL_HOM_EMBEDDING_IN_TENSOR_PRODUCT_LEFT( object_1,
       object_2 );
    rank_1 := NrColumns( UnderlyingMatrix( object_1 ) );
    free_module := FreeLeftPresentation( rank_1, homalg_ring );
    morphism := PreCompose( internal_hom_embedding,
       Braiding( free_module, object_2 ) );
    morphism := TensorProductOnMorphisms( morphism,
       IdentityMorphism( object_1 ) );
    column := [  ];
    zero_column := List( [ 1 .. rank_1 ], function ( i )
            return 0;
        end );
    for i  in [ 1 .. rank_1 - 1 ]  do
        Add( column, 1 );
        Append( column, zero_column );
    od;
    if rank_1 > 0  then
        Add( column, 1 );
    fi;
    matrix := HomalgMatrix( column, rank_1 * rank_1, 1, homalg_ring
       );
    rank_2 := NrColumns( UnderlyingMatrix( object_2 ) );
    matrix
     := KroneckerMat( HomalgIdentityMatrix( rank_2, homalg_ring ),
       matrix );
    lifted_evaluation := PresentationMorphism( Range( morphism ),
       matrix, object_2 );
    return PreCompose( morphism, lifted_evaluation );
end;
```

**CoevaluationMorphismWithGivenRange.** Back to index.

```
function ( object_1, object_2, internal_hom )
    local  object_1_tensored_object_2, internal_hom_embedding,
    rank_2, free_module, morphism, row, zero_row, i, matrix,
    rank_1, lifted_coevaluation;
    object_1_tensored_object_2 := TensorProductOnObjects( object_1,
       object_2 );
    internal_hom_embedding
     := INTERNAL_HOM_EMBEDDING_IN_TENSOR_PRODUCT_LEFT( object_2,
       object_1_tensored_object_2 );
    rank_2 := NrColumns( UnderlyingMatrix( object_2 ) );
    free_module := FreeLeftPresentation( rank_2, homalg_ring );
    morphism := PreCompose( internal_hom_embedding,
       Braiding( free_module, object_1_tensored_object_2 ) );
    row := [  ];
    zero_row := List( [ 1 .. rank_2 ], function ( i )
            return 0;
        end );
    for i  in [ 1 .. rank_2 - 1 ]  do
        Add( row, 1 );
        Append( row, zero_row );
    od;
    if rank_2 > 0  then
        Add( row, 1 );
    fi;
    matrix := HomalgMatrix( row, 1, rank_2 * rank_2, homalg_ring );
    rank_1 := NrColumns( UnderlyingMatrix( object_1 ) );
    matrix
     := KroneckerMat( HomalgIdentityMatrix( rank_1, homalg_ring ),
       matrix );
    lifted_coevaluation := PresentationMorphism( object_1, matrix,
       Range( morphism ) );
    return LiftAlongMonomorphism( morphism, lifted_coevaluation );
end;
```

## 3. Primitive operations for right module presentations

**IdentityMorphism.** Back to index.

```
function ( object )
    local  matrix;
    matrix
     := HomalgIdentityMatrix( NrRows( UnderlyingMatrix( object ) ),
       homalg_ring );
    return PresentationMorphism( object, matrix, object );
end;
```

**KernelEmbedding.** Back to index.

```
function ( morphism )
    local  kernel, embedding;
    embedding := SyzygiesOfColumns( UnderlyingMatrix( morphism ),
       UnderlyingMatrix( Range( morphism ) ) );
    kernel := SyzygiesOfColumns( embedding,
       UnderlyingMatrix( Source( morphism ) ) );
    kernel := AsRightPresentation( kernel );
    return PresentationMorphism( kernel, embedding,
       Source( morphism ) );
end;
```

**CokernelProjection.** Back to index.

```
function ( morphism )
    local  cokernel_object, projection;
    cokernel_object
     := UnionOfColumns( UnderlyingMatrix( morphism ),
       UnderlyingMatrix( Range( morphism ) ) );
    cokernel_object := AsRightPresentation( cokernel_object );
    projection
     := HomalgIdentityMatrix(
       NrRows( UnderlyingMatrix( Range( morphism ) ) ), homalg_ring
       );
    return PresentationMorphism( Range( morphism ), projection,
       cokernel_object );
end;
```

**ZeroObject.** Back to index.

```
function (  )
    local  matrix;
    matrix := HomalgZeroMatrix( 0, 0, homalg_ring );
    return AsRightPresentation( matrix );
end;
```

**TensorUnit.** Back to index.

```
function (  )
    return
     AsRightPresentation( HomalgZeroMatrix( 1, 0, homalg_ring ) );
end;
```

**Lift.** Back to index.

```
function ( beta, alpha )
    local  lift;
    lift := LeftDivide( UnderlyingMatrix( alpha ),
       UnderlyingMatrix( beta ), UnderlyingMatrix( Range( alpha ) )
       );
    if lift = fail  then
        return fail;
    fi;
    return PresentationMorphism( Source( beta ), lift,
       Source( alpha ) );
end;
```

**KernelEmbeddingWithGivenKernelObject.** Back to index.

```
function ( morphism, kernel )
    local  embedding;
    embedding := SyzygiesOfColumns( UnderlyingMatrix( morphism ),
       UnderlyingMatrix( Range( morphism ) ) );
    return PresentationMorphism( kernel, embedding,
       Source( morphism ) );
end;
```

**CokernelProjectionWithGivenCokernelObject.** Back to index.

```
function ( morphism, cokernel_object )
    local  projection;
    projection
     := HomalgIdentityMatrix(
       NrRows( UnderlyingMatrix( Range( morphism ) ) ), homalg_ring
       );
    return PresentationMorphism( Range( morphism ), projection,
       cokernel_object );
end;
```

**CokernelColiftWithGivenCokernelObject.** Back to index.

```
function ( morphism, test_morphism, cokernel_object )
    return PresentationMorphism( cokernel_object,
       UnderlyingMatrix( test_morphism ), Range( test_morphism ) );
end;
```

**PreCompose.** Back to index.

```
function ( left_morphism, right_morphism )
    return PresentationMorphism( Source( left_morphism ),
       UnderlyingMatrix( right_morphism )
        * UnderlyingMatrix( left_morphism ),
       Range( right_morphism ) );
end;
```

```
function ( left_morphism, identity_morphism )
    return left_morphism;
end;
```

This function uses the following extra filters:

- IsIdenticalToIdentityMorphism for the 2nd argument.

```
function ( identity_morphism, right_morphism )
    return right_morphism;
end;
```

This function uses the following extra filters:

- IsIdenticalToIdentityMorphism for the 1st argument.

```
function ( left_morphism, zero_morphism )
    return PresentationMorphism( Source( left_morphism ),
       HomalgZeroMatrix( NrRows( UnderlyingMatrix( zero_morphism ) )
          , NrColumns( UnderlyingMatrix( left_morphism ) ),
         homalg_ring ), Range( zero_morphism ) );
end;
```

This function uses the following extra filters:

- IsIdenticalToZeroMorphism for the 2nd argument.

```
function ( zero_morphism, right_morphism )
    return PresentationMorphism( Source( zero_morphism ),
       HomalgZeroMatrix( NrRows( UnderlyingMatrix( right_morphism )
           ), NrColumns( UnderlyingMatrix( zero_morphism ) ),
         homalg_ring ), Range( right_morphism ) );
end;
```

This function uses the following extra filters:

- IsIdenticalToZeroMorphism for the 1st argument.

**UniversalMorphismFromZeroObjectWithGivenZeroObject.** Back to index.

```
function ( object, initial_object )
    local  nr_rows, morphism;
    nr_rows := NrRows( UnderlyingMatrix( object ) );
    morphism := HomalgZeroMatrix( nr_rows, 0, homalg_ring );
    return PresentationMorphism( initial_object, morphism, object );
end;
```

**UniversalMorphismIntoZeroObjectWithGivenZeroObject.** Back to index.

```
function ( object, terminal_object )
    local  nr_rows, morphism;
    nr_rows := NrRows( UnderlyingMatrix( object ) );
    morphism := HomalgZeroMatrix( 0, nr_rows, homalg_ring );
    return PresentationMorphism( object, morphism, terminal_object );
end;
```

**ZeroMorphism.** Back to index.

```
function ( source, range )
    local  matrix;
    matrix := HomalgZeroMatrix( NrRows( UnderlyingMatrix( range ) )
        , NrRows( UnderlyingMatrix( source ) ), homalg_ring );
    return PresentationMorphism( source, matrix, range );
end;
```

**DirectSum.** Back to index.

```
function ( product_object )
    local  objects, direct_sum;
    objects := product_object;
    objects := List( objects, UnderlyingMatrix );
    direct_sum := DiagMat( objects );
    return AsRightPresentation( direct_sum );
end;
```

**ProjectionInFactorOfDirectSumWithGivenDirectSum.** Back to index.

```
function ( product_object, component_number, direct_sum_object )
    local  objects, object_column_dimension, dimension_of_factor,
    projection, projection_matrix, i;
    objects := product_object;
    object_column_dimension := List( objects, function ( i )
            return NrRows( UnderlyingMatrix( i ) );
        end );
    dimension_of_factor := object_column_dimension[component_number];
    projection := List( object_column_dimension, function ( i )
            return HomalgZeroMatrix( dimension_of_factor, i,
               homalg_ring );
        end );
    projection[component_number]
     := HomalgIdentityMatrix(
       object_column_dimension[component_number], homalg_ring );
    projection_matrix := projection[1];
    for i  in [ 2 .. Length( objects ) ]  do
        projection_matrix := UnionOfColumns( projection_matrix,
           projection[i] );
    od;
    return PresentationMorphism( direct_sum_object,
       projection_matrix, objects[component_number] );
end;
```

**UniversalMorphismIntoDirectSumWithGivenDirectSum.** Back to index.

```
function ( diagram, product_morphism, direct_sum )
    local  components, number_of_components, map_into_product, i;
    components := product_morphism;
    number_of_components := Length( components );
    map_into_product := UnderlyingMatrix( components[1] );
    for i  in [ 2 .. number_of_components ]  do
        map_into_product := UnionOfRows( map_into_product,
           UnderlyingMatrix( components[i] ) );
    od;
    return PresentationMorphism( Source( components[1] ),
       map_into_product, direct_sum );
end;
```

**InjectionOfCofactorOfDirectSumWithGivenDirectSum.** Back to index.

```
function ( product_object, component_number, direct_sum_object )
    local  objects, object_column_dimension, dimension_of_cofactor,
    injection, injection_matrix, i;
    objects := product_object;
    object_column_dimension := List( objects, function ( i )
            return NrRows( UnderlyingMatrix( i ) );
        end );
    dimension_of_cofactor
     := object_column_dimension[component_number];
    injection := List( object_column_dimension, function ( i )
            return HomalgZeroMatrix( i, dimension_of_cofactor,
               homalg_ring );
        end );
    injection[component_number]
     := HomalgIdentityMatrix(
       object_column_dimension[component_number], homalg_ring );
    injection_matrix := injection[1];
    for i  in [ 2 .. Length( objects ) ]  do
        injection_matrix := UnionOfRows( injection_matrix,
           injection[i] );
    od;
    return PresentationMorphism( objects[component_number],
       injection_matrix, direct_sum_object );
end;
```

**UniversalMorphismFromDirectSumWithGivenDirectSum.** Back to index.

```
function ( diagram, product_morphism, direct_sum )
    local  components, number_of_components, map_into_product, i;
    components := product_morphism;
    number_of_components := Length( components );
    map_into_product := UnderlyingMatrix( components[1] );
    for i  in [ 2 .. number_of_components ]  do
        map_into_product := UnionOfColumns( map_into_product,
           UnderlyingMatrix( components[i] ) );
    od;
    return PresentationMorphism( direct_sum, map_into_product,
       Range( components[1] ) );
end;
```

**IsCongruentForMorphisms.** Back to index.

```
function ( morphism_1, morphism_2 )
    local  result_of_divide;
    result_of_divide
     := DecideZeroColumns( UnderlyingMatrix( morphism_1 )
         - UnderlyingMatrix( morphism_2 ),
       UnderlyingMatrix( Range( morphism_1 ) ) );
    return IsZero( result_of_divide );
end;
```

**IsEqualForMorphisms.** Back to index.

```
function ( morphism_1, morphism_2 )
    return UnderlyingMatrix( morphism_1 )
      = UnderlyingMatrix( morphism_2 );
end;
```

**IsEqualForObjects.** Back to index.

```
function ( object1, object2 )
    return UnderlyingMatrix( object1 ) = UnderlyingMatrix( object2 );
end;
```

**IsEqualForCacheForObjects.** Back to index.

```
IsIdenticalObj
```

**IsEqualForCacheForMorphisms.** Back to index.

```
IsIdenticalObj
```

**AdditionForMorphisms.** Back to index.

```
function ( morphism_1, morphism_2 )
    return PresentationMorphism( Source( morphism_1 ),
       UnderlyingMatrix( morphism_1 )
        + UnderlyingMatrix( morphism_2 ), Range( morphism_1 ) );
end;
```

**AdditiveInverseForMorphisms.** Back to index.

```
function ( morphism_1 )
    return PresentationMorphism( Source( morphism_1 ),
       - UnderlyingMatrix( morphism_1 ), Range( morphism_1 ) );
end;
```

**IsWellDefinedForMorphisms.** Back to index.

```
function ( morphism )
    local  source_matrix, range_matrix, morphism_matrix;
    source_matrix := UnderlyingMatrix( Source( morphism ) );
    range_matrix := UnderlyingMatrix( Range( morphism ) );
    morphism_matrix := UnderlyingMatrix( morphism );
    if
     not (NrRows( source_matrix ) = NrColumns( morphism_matrix )
           and NrRows( morphism_matrix ) = NrRows( range_matrix ))
         then
        return false;
    fi;
    if
     LeftDivide( range_matrix, morphism_matrix * source_matrix )
        = fail  then
        return false;
    fi;
    return true;
end;
```

**IsWellDefinedForObjects.** Back to index.

```
function ( object )
    return IsHomalgMatrix( UnderlyingMatrix( object ) )
      and IsHomalgRing( UnderlyingHomalgRing( object ) );
end;
```

**TensorProductOnObjects.** Back to index.

```
function ( object_1, object_2 )
    local  identity_1, identity_2, presentation_matrix_1,
    presentation_matrix_2, presentation_matrix;
    presentation_matrix_1 := UnderlyingMatrix( object_1 );
    presentation_matrix_2 := UnderlyingMatrix( object_2 );
    identity_1
     := HomalgIdentityMatrix( NrRows( presentation_matrix_1 ),
       homalg_ring );
    identity_2
     := HomalgIdentityMatrix( NrRows( presentation_matrix_2 ),
       homalg_ring );
    presentation_matrix
     := UnionOfColumns(
       KroneckerMat( identity_1, presentation_matrix_2 ),
       KroneckerMat( presentation_matrix_1, identity_2 ) );
    return AsRightPresentation( presentation_matrix );
end;
```

**TensorProductOnMorphismsWithGivenTensorProducts.** Back to index.

```
function ( new_source, morphism_1, morphism_2, new_range )
    return PresentationMorphism( new_source,
       KroneckerMat( UnderlyingMatrix( morphism_1 ),
         UnderlyingMatrix( morphism_2 ) ), new_range );
end;
```

**BraidingWithGivenTensorProducts.** Back to index.

```
function ( object_1_tensored_object_2, object_1, object_2,
    object_2_tensored_object_1 )
    local  homalg_ring, permutation_matrix, rank_1, rank_2, rank;
    homalg_ring := UnderlyingHomalgRing( object_1 );
    rank_1 := NrRows( UnderlyingMatrix( object_1 ) );
    rank_2 := NrRows( UnderlyingMatrix( object_2 ) );
    rank := NrRows( UnderlyingMatrix( object_1_tensored_object_2 ) );
    permutation_matrix
     := PermutationMat(
       PermList( List( [ 1 .. rank ], function ( i )
                return
                 RemInt( (i - 1), rank_2 ) * rank_1
                    + QuoInt( (i - 1), rank_2 ) + 1;
            end ) ), rank );
    return PresentationMorphism( object_1_tensored_object_2,
       Involution( HomalgMatrix( permutation_matrix, rank, rank,
           homalg_ring ) ), object_2_tensored_object_1 );
end;
```

**InternalHomOnObjects.** Back to index.

```
function ( object_1, object_2 )
    return
     Source( INTERNAL_HOM_EMBEDDING_IN_TENSOR_PRODUCT_RIGHT(
         object_1, object_2 ) );
end;
```

**InternalHomOnMorphismsWithGivenInternalHoms.** Back to index.

```
function ( new_source, morphism_1, morphism_2, new_range )
    local  internal_hom_embedding_source,
    internal_hom_embedding_range, morphism_between_tensor_products;
    internal_hom_embedding_source
     := INTERNAL_HOM_EMBEDDING_IN_TENSOR_PRODUCT_RIGHT(
       Range( morphism_1 ), Source( morphism_2 ) );
    internal_hom_embedding_range
     := INTERNAL_HOM_EMBEDDING_IN_TENSOR_PRODUCT_RIGHT(
       Source( morphism_1 ), Range( morphism_2 ) );
    morphism_between_tensor_products
     := PresentationMorphism( Range( internal_hom_embedding_source )
        , KroneckerMat( Involution( UnderlyingMatrix( morphism_1 ) )
          , UnderlyingMatrix( morphism_2 ) ),
       Range( internal_hom_embedding_range ) );
    return LiftAlongMonomorphism( internal_hom_embedding_range,
       PreCompose( internal_hom_embedding_source,
         morphism_between_tensor_products ) );
end;
```

**EvaluationMorphismWithGivenSource.** Back to index.

```
function ( object_1, object_2, internal_hom_tensored_object_1 )
    local  internal_hom_embedding, rank_1, morphism, free_module,
    row, zero_row, i, matrix, rank_2, lifted_evaluation;
    internal_hom_embedding
     := INTERNAL_HOM_EMBEDDING_IN_TENSOR_PRODUCT_RIGHT( object_1,
       object_2 );
    rank_1 := NrRows( UnderlyingMatrix( object_1 ) );
    free_module := FreeRightPresentation( rank_1, homalg_ring );
    morphism := PreCompose( internal_hom_embedding,
       Braiding( free_module, object_2 ) );
    morphism := TensorProductOnMorphisms( morphism,
       IdentityMorphism( object_1 ) );
    row := [  ];
    zero_row := List( [ 1 .. rank_1 ], function ( i )
            return 0;
        end );
    for i  in [ 1 .. rank_1 - 1 ]  do
        Add( row, 1 );
        Append( row, zero_row );
    od;
    if rank_1 > 0  then
        Add( row, 1 );
    fi;
    matrix := HomalgMatrix( row, 1, rank_1 * rank_1, homalg_ring );
    rank_2 := NrRows( UnderlyingMatrix( object_2 ) );
    matrix
     := KroneckerMat( HomalgIdentityMatrix( rank_2, homalg_ring ),
       matrix );
    lifted_evaluation := PresentationMorphism( Range( morphism ),
       matrix, object_2 );
    return PreCompose( morphism, lifted_evaluation );
end;
```

**CoevaluationMorphismWithGivenRange.** Back to index.

```
function ( object_1, object_2, internal_hom )
    local  object_1_tensored_object_2, internal_hom_embedding,
    rank_2, free_module, morphism, column, zero_column, i, matrix,
    rank_1, lifted_coevaluation;
    object_1_tensored_object_2 := TensorProductOnObjects( object_1,
       object_2 );
    internal_hom_embedding
```

```
     := INTERNAL_HOM_EMBEDDING_IN_TENSOR_PRODUCT_RIGHT( object_2,
       object_1_tensored_object_2 );
    rank_2 := NrRows( UnderlyingMatrix( object_2 ) );
    free_module := FreeRightPresentation( rank_2, homalg_ring );
    morphism := PreCompose( internal_hom_embedding,
       Braiding( free_module, object_1_tensored_object_2 ) );
    column := [  ];
    zero_column := List( [ 1 .. rank_2 ], function ( i )
            return 0;
        end );
    for i  in [ 1 .. rank_2 - 1 ]  do
        Add( column, 1 );
        Append( column, zero_column );
    od;
    if rank_2 > 0  then
        Add( column, 1 );
    fi;
    matrix := HomalgMatrix( column, rank_2 * rank_2, 1, homalg_ring
       );
    rank_1 := NrRows( UnderlyingMatrix( object_1 ) );
    matrix
     := KroneckerMat( HomalgIdentityMatrix( rank_1, homalg_ring ),
       matrix );
    lifted_coevaluation := PresentationMorphism( object_1, matrix,
       Range( morphism ) );
    return LiftAlongMonomorphism( morphism, lifted_coevaluation );
end;
```

## 4. Primitive operations for graded left module presentations

**IdentityMorphism.** Back to index.

```
function ( object )
    local  morphism;
    morphism
     := IdentityMorphism( UnderlyingPresentationObject( object ) );
    return GradedPresentationMorphism( object, morphism, object );
end;
```

**KernelEmbedding.** Back to index.

```
function ( morphism )
    local  underlying_embedding, kernel_object, range_degrees,
    new_degrees;
    underlying_embedding
     := KernelEmbedding( UnderlyingPresentationMorphism( morphism )
       );
    kernel_object := Source( underlying_embedding );
    new_degrees
     := NonTrivialDegreePerRow(
       UnderlyingMatrix( underlying_embedding ),
       GeneratorDegrees( Source( morphism ) ) );
    kernel_object := AsGradedLeftPresentation( kernel_object,
       new_degrees );
    return GradedPresentationMorphism( kernel_object,
       underlying_embedding, Source( morphism ) );
end;
```

**CokernelObject.** Back to index.

```
function ( object )
    local  result;
    result
     := CokernelObject( UnderlyingPresentationMorphism( object ) );
    return
     object_constructor( result, GeneratorDegrees( Range( object )
         ) );
end;
```

**ZeroObject.** Back to index.

```
function (  )
    local  zero_object;
    zero_object := ZeroObject( underlying_presentation_category );
    return object_constructor( zero_object );
end;
```

**TensorUnit.** Back to index.

```
function (  )
    local  unit, new_degrees;
    unit := TensorUnit( underlying_presentation_category );
    return object_constructor( unit );
end;
```

**Lift.** Back to index.

```
function ( alpha, beta )
    local  lift;
    lift := Lift( UnderlyingPresentationMorphism( alpha ),
       UnderlyingPresentationMorphism( beta ) );
    if lift = fail  then
        return fail;
    fi;
    return GradedPresentationMorphism( Source( alpha ), lift,
       Source( beta ) );
end;
```

**KernelEmbeddingWithGivenKernelObject.** Back to index.

```
function ( morphism, kernel )
    local  underlying_embedding;
    underlying_embedding
     := KernelEmbedding( UnderlyingPresentationMorphism( morphism )
       );
    return GradedPresentationMorphism( kernel, underlying_embedding
        , Source( morphism ) );
end;
```

**CokernelProjectionWithGivenCokernelObject.** Back to index.

```
function ( morphism, cokernel_object )
    local  projection;
    projection := CokernelProjectionWithGivenCokernelObject(
       UnderlyingPresentationMorphism( morphism ),
       UnderlyingPresentationObject( cokernel_object ) );
    return GradedPresentationMorphism( Range( morphism ),
       projection, cokernel_object );
end;
```

**CokernelColiftWithGivenCokernelObject.** Back to index.

```
function ( morphism, test_morphism, cokernel_object )
    local  lift;
    lift := CokernelColiftWithGivenCokernelObject(
       UnderlyingPresentationMorphism( morphism ),
       UnderlyingPresentationMorphism( test_morphism ),
       UnderlyingPresentationObject( cokernel_object ) );
    return GradedPresentationMorphism( cokernel_object, lift,
       Range( test_morphism ) );
end;
```

**PreCompose.** Back to index.

```
function ( left_morphism, right_morphism )
    return GradedPresentationMorphism( Source( left_morphism ),
       PreCompose( UnderlyingPresentationMorphism( left_morphism ),
         UnderlyingPresentationMorphism( right_morphism ) ),
       Range( right_morphism ) );
end;
```

**UniversalMorphismFromZeroObjectWithGivenZeroObject.** Back to index.

```
function ( object, initial_object )
    local  morphism;
    morphism := UniversalMorphismFromZeroObjectWithGivenZeroObject(
       UnderlyingPresentationObject( object ),
       UnderlyingPresentationObject( initial_object ) );
    return GradedPresentationMorphism( initial_object, morphism,
       object );
end;
```

**UniversalMorphismIntoZeroObjectWithGivenZeroObject.** Back to index.

```
function ( object, terminal_object )
    local  morphism;
    morphism := UniversalMorphismIntoZeroObjectWithGivenZeroObject(
       UnderlyingPresentationObject( object ),
       UnderlyingPresentationObject( terminal_object ) );
    return GradedPresentationMorphism( object, morphism,
       terminal_object );
end;
```

**ZeroMorphism.** Back to index.

```
function ( source, range )
    local  morphism;
    morphism := ZeroMorphism( UnderlyingPresentationObject( source )
        , UnderlyingPresentationObject( range ) );
    return GradedPresentationMorphism( source, morphism, range );
end;
```

**DirectSum.** Back to index.

```
function ( product_object )
    local  objects, degrees;
    objects
     :=
      DirectSum( List( product_object, UnderlyingPresentationObject
         ) );
    degrees
     := Concatenation( List( product_object, GeneratorDegrees ) );
    return object_constructor( objects, degrees );
end;
```

**ProjectionInFactorOfDirectSumWithGivenDirectSum.** Back to index.

```
function ( product_object, component_number, direct_sum_object )
    local  underlying_objects, underlying_direct_sum, projection;
    underlying_objects
     := List( product_object, UnderlyingPresentationObject );
    underlying_direct_sum := UnderlyingPresentationObject(
       direct_sum_object );
    projection := ProjectionInFactorOfDirectSumWithGivenDirectSum(
       underlying_objects, component_number, underlying_direct_sum );
    return GradedPresentationMorphism( direct_sum_object,
       projection, product_object[component_number] );
end;
```

**UniversalMorphismIntoDirectSumWithGivenDirectSum.** Back to index.

```
function ( diagram, product_morphism, direct_sum )
    local  underlying_diagram, underlying_product_morphism,
    underlying_direct_sum, universal_morphism;
    underlying_diagram
     := List( diagram, UnderlyingPresentationObject );
    underlying_product_morphism
     := List( product_morphism, UnderlyingPresentationMorphism );
    underlying_direct_sum := UnderlyingPresentationObject(
       direct_sum );
    universal_morphism
     := UniversalMorphismIntoDirectSumWithGivenDirectSum(
       underlying_diagram, underlying_product_morphism,
       underlying_direct_sum );
    return GradedPresentationMorphism( Source( product_morphism[1] )
        , universal_morphism, direct_sum );
end;
```

**InjectionOfCofactorOfDirectSumWithGivenDirectSum.** Back to index.

```
function ( product_object, component_number, direct_sum_object )
    local  underlying_objects, underlying_direct_sum, injection;
    underlying_objects
```

```
     := List( product_object, UnderlyingPresentationObject );
    underlying_direct_sum := UnderlyingPresentationObject(
       direct_sum_object );
    injection := InjectionOfCofactorOfDirectSumWithGivenDirectSum(
       underlying_objects, component_number, underlying_direct_sum );
    return
     GradedPresentationMorphism( product_object[component_number],
       injection, direct_sum_object );
end;
```

**UniversalMorphismFromDirectSumWithGivenDirectSum.** Back to index.

```
function ( diagram, product_morphism, direct_sum )
    local  underlying_diagram, underlying_product_morphism,
    underlying_direct_sum, universal_morphism;
    underlying_diagram
     := List( diagram, UnderlyingPresentationObject );
    underlying_product_morphism
     := List( product_morphism, UnderlyingPresentationMorphism );
    underlying_direct_sum := UnderlyingPresentationObject(
       direct_sum );
    universal_morphism
     := UniversalMorphismFromDirectSumWithGivenDirectSum(
       underlying_diagram, underlying_product_morphism,
       underlying_direct_sum );
    return GradedPresentationMorphism( direct_sum,
       universal_morphism, Range( product_morphism[1] ) );
end;
```

**IsCongruentForMorphisms.** Back to index.

```
function ( morphism_1, morphism_2 )
    return
     IsCongruentForMorphisms( UnderlyingPresentationMorphism(
         morphism_1 ), UnderlyingPresentationMorphism( morphism_2 )
       );
end;
```

**IsEqualForMorphisms.** Back to index.

```
function ( morphism_1, morphism_2 )
    return UnderlyingMatrix( morphism_1 )
      = UnderlyingMatrix( morphism_2 );
end;
```

**IsEqualForObjects.** Back to index.

```
function ( object1, object2 )
    if UnderlyingMatrix( object1 ) = UnderlyingMatrix( object2 )
         then
        return GeneratorDegrees( object1 )
          = GeneratorDegrees( object2 );
    fi;
    return false;
end;
```

**IsEqualForCacheForObjects.** Back to index.

```
IsIdenticalObj
```

**IsEqualForCacheForMorphisms.** Back to index.

```
IsIdenticalObj
```

**AdditionForMorphisms.** Back to index.

```
function ( morphism_1, morphism_2 )
    return GradedPresentationMorphism( Source( morphism_1 ),
       UnderlyingPresentationMorphism( morphism_1 )
        + UnderlyingPresentationMorphism( morphism_2 ),
       Range( morphism_1 ) );
end;
```

**AdditiveInverseForMorphisms.** Back to index.

```
function ( morphism_1 )
    return GradedPresentationMorphism( Source( morphism_1 ),
       - UnderlyingPresentationMorphism( morphism_1 ),
       Range( morphism_1 ) );
end;
```

**IsWellDefinedForMorphisms.** Back to index.

```
function ( morphism )
    local  matrix_degrees, matrix_entries, source_degrees,
    range_degrees;
    if
     not IsWellDefined( UnderlyingPresentationMorphism( morphism ) )
       then
        return false;
    fi;
    return GeneratorDegrees( Source( morphism ) )
```

```
      = NonTrivialDegreePerRow( UnderlyingMatrix( morphism ),
         GeneratorDegrees( Range( morphism ) ) );
end;
```

**IsWellDefinedForObjects.** Back to index.

```
function ( object )
    local  relation_degrees, generator_degrees, relation_entries;
    if
     not IsHomalgMatrix( UnderlyingMatrix( object ) )
        or not IsHomalgRing( UnderlyingHomalgRing( object ) )  then
        return false;
    fi;
    relation_degrees
     := DegreesOfEntries( UnderlyingMatrix( object ) );
    relation_entries := EntriesOfHomalgMatrixAsListList(
       UnderlyingMatrix( object ) );
    generator_degrees := GeneratorDegrees( object );
    return
     CAP_INTERNAL_CHECK_DEGREES_FOR_IS_WELL_DEFINED_FOR_OBJECTS(
       relation_degrees, relation_entries, generator_degrees );
end;
```

**TensorProductOnObjects.** Back to index.

```
function ( object_1, object_2 )
    local  new_object, degrees_1, degrees_2, new_degrees, i, j;
    new_object
     := TensorProductOnObjects(
       UnderlyingPresentationObject( object_1 ),
       UnderlyingPresentationObject( object_2 ) );
    degrees_1 := GeneratorDegrees( object_1 );
    degrees_2 := GeneratorDegrees( object_2 );
    new_degrees := [  ];
    for i  in [ 1 .. Length( degrees_1 ) ]  do
        for j  in [ 1 .. Length( degrees_2 ) ]  do
            Add( new_degrees, degrees_1[i] + degrees_2[j] );
        od;
    od;
    return object_constructor( new_object, new_degrees );
end;
```

**TensorProductOnMorphismsWithGivenTensorProducts.** Back to index.

```
function ( new_source, morphism_1, morphism_2, new_range )
    local  new_morphism;
    new_morphism := TensorProductOnMorphismsWithGivenTensorProducts
        ( UnderlyingPresentationObject( new_source ),
       UnderlyingPresentationMorphism( morphism_1 ),
       UnderlyingPresentationMorphism( morphism_2 ),
       UnderlyingPresentationObject( new_range ) );
    return GradedPresentationMorphism( new_source, new_morphism,
       new_range );
end;
```

**InternalHomOnObjects.** Back to index.

```
function ( object_1, object_2 )
    return
     Source( INTERNAL_GRADED_HOM_EMBEDDING_IN_TENSOR_PRODUCT_LEFT(
         object_1, object_2 ) );
end;
```

**InternalHomOnMorphismsWithGivenInternalHoms.** Back to index.

```
function ( new_source, morphism_1, morphism_2, new_range )
    local  internal_hom_embedding_source,
    internal_hom_embedding_range, morphism_between_tensor_products;
    internal_hom_embedding_source
     := INTERNAL_GRADED_HOM_EMBEDDING_IN_TENSOR_PRODUCT_LEFT(
       Range( morphism_1 ), Source( morphism_2 ) );
    internal_hom_embedding_range
     := INTERNAL_GRADED_HOM_EMBEDDING_IN_TENSOR_PRODUCT_LEFT(
       Source( morphism_1 ), Range( morphism_2 ) );
    morphism_between_tensor_products
     := GradedPresentationMorphism(
       Range( internal_hom_embedding_source ),
       KroneckerMat( Involution( UnderlyingMatrix( morphism_1 ) ),
         UnderlyingMatrix( morphism_2 ) ),
       Range( internal_hom_embedding_range ) );
    return LiftAlongMonomorphism( internal_hom_embedding_range,
       PreCompose( internal_hom_embedding_source,
         morphism_between_tensor_products ) );
end;
```

## 5. Primitive operations for graded right module presentations

**IdentityMorphism.** Back to index.

```
function ( object )
    local  morphism;
    morphism
     := IdentityMorphism( UnderlyingPresentationObject( object ) );
    return GradedPresentationMorphism( object, morphism, object );
end;
```

**KernelEmbedding.** Back to index.

```
function ( morphism )
    local  underlying_embedding, kernel_object, new_degrees,
    range_degrees;
    underlying_embedding
     := KernelEmbedding( UnderlyingPresentationMorphism( morphism )
       );
    kernel_object := Source( underlying_embedding );
    new_degrees
     := NonTrivialDegreePerColumn(
       UnderlyingMatrix( underlying_embedding ),
       GeneratorDegrees( Source( morphism ) ) );
    kernel_object := AsGradedRightPresentation( kernel_object,
       new_degrees );
    return GradedPresentationMorphism( kernel_object,
       underlying_embedding, Source( morphism ) );
end;
```

**CokernelObject.** Back to index.

```
function ( object )
    local  result;
    result
     := CokernelObject( UnderlyingPresentationMorphism( object ) );
    return
     object_constructor( result, GeneratorDegrees( Range( object )
         ) );
end;
```

**ZeroObject.** Back to index.

```
function (  )
    local  zero_object;
    zero_object := ZeroObject( underlying_presentation_category );
    return object_constructor( zero_object );
end;
```

**TensorUnit.** Back to index.

```
function (  )
    local  unit, new_degrees;
    unit := TensorUnit( underlying_presentation_category );
    return object_constructor( unit );
end;
```

**Lift.** Back to index.

```
function ( beta, alpha )
    local  lift;
    lift := Lift( UnderlyingPresentationMorphism( beta ),
       UnderlyingPresentationMorphism( alpha ) );
    if lift = fail  then
        return fail;
    fi;
    return GradedPresentationMorphism( Source( beta ), lift,
       Source( alpha ) );
end;
```

**KernelEmbeddingWithGivenKernelObject.** Back to index.

```
function ( morphism, kernel )
    local  underlying_embedding;
    underlying_embedding
     := KernelEmbedding( UnderlyingPresentationMorphism( morphism )
       );
    return GradedPresentationMorphism( kernel, underlying_embedding
        , Source( morphism ) );
end;
```

**CokernelProjectionWithGivenCokernelObject.** Back to index.

```
function ( morphism, cokernel_object )
    local  projection;
    projection := CokernelProjectionWithGivenCokernelObject(
       UnderlyingPresentationMorphism( morphism ),
       UnderlyingPresentationObject( cokernel_object ) );
    return GradedPresentationMorphism( Range( morphism ),
       projection, cokernel_object );
end;
```

**CokernelColiftWithGivenCokernelObject.** Back to index.

```
function ( morphism, test_morphism, cokernel_object )
    local  lift;
    lift := CokernelColiftWithGivenCokernelObject(
       UnderlyingPresentationMorphism( morphism ),
       UnderlyingPresentationMorphism( test_morphism ),
       UnderlyingPresentationObject( cokernel_object ) );
    return GradedPresentationMorphism( cokernel_object, lift,
       Range( test_morphism ) );
end;
```

**PreCompose.** Back to index.

```
function ( left_morphism, right_morphism )
    return GradedPresentationMorphism( Source( left_morphism ),
       PreCompose( UnderlyingPresentationMorphism( left_morphism ),
         UnderlyingPresentationMorphism( right_morphism ) ),
       Range( right_morphism ) );
end;
```

**UniversalMorphismFromZeroObjectWithGivenZeroObject.** Back to index.

```
function ( object, initial_object )
    local  morphism;
    morphism := UniversalMorphismFromZeroObjectWithGivenZeroObject(
       UnderlyingPresentationObject( object ),
       UnderlyingPresentationObject( initial_object ) );
    return GradedPresentationMorphism( initial_object, morphism,
       object );
end;
```

**UniversalMorphismIntoZeroObjectWithGivenZeroObject.** Back to index.

```
function ( object, terminal_object )
    local  morphism;
    morphism := UniversalMorphismIntoZeroObjectWithGivenZeroObject(
       UnderlyingPresentationObject( object ),
       UnderlyingPresentationObject( terminal_object ) );
    return GradedPresentationMorphism( object, morphism,
       terminal_object );
end;
```

**ZeroMorphism.** Back to index.

```
function ( source, range )
    local  morphism;
    morphism := ZeroMorphism( UnderlyingPresentationObject( source )
         , UnderlyingPresentationObject( range ) );
    return GradedPresentationMorphism( source, morphism, range );
end;
```

**DirectSum.** Back to index.

```
function ( product_object )
    local  objects, degrees;
    objects
     :=
      DirectSum( List( product_object, UnderlyingPresentationObject
         ) );
    degrees
     := Concatenation( List( product_object, GeneratorDegrees ) );
    return object_constructor( objects, degrees );
end;
```

**ProjectionInFactorOfDirectSumWithGivenDirectSum.** Back to index.

```
function ( product_object, component_number, direct_sum_object )
    local  underlying_objects, underlying_direct_sum, projection;
    underlying_objects
     := List( product_object, UnderlyingPresentationObject );
    underlying_direct_sum := UnderlyingPresentationObject(
       direct_sum_object );
    projection := ProjectionInFactorOfDirectSumWithGivenDirectSum(
       underlying_objects, component_number, underlying_direct_sum );
    return GradedPresentationMorphism( direct_sum_object,
       projection, product_object[component_number] );
end;
```

**UniversalMorphismIntoDirectSumWithGivenDirectSum.** Back to index.

```
function ( diagram, product_morphism, direct_sum )
    local  underlying_diagram, underlying_product_morphism,
    underlying_direct_sum, universal_morphism;
    underlying_diagram
     := List( diagram, UnderlyingPresentationObject );
    underlying_product_morphism
     := List( product_morphism, UnderlyingPresentationMorphism );
    underlying_direct_sum := UnderlyingPresentationObject(
       direct_sum );
    universal_morphism
     := UniversalMorphismIntoDirectSumWithGivenDirectSum(
       underlying_diagram, underlying_product_morphism,
       underlying_direct_sum );
    return GradedPresentationMorphism( Source( product_morphism[1] )
        , universal_morphism, direct_sum );
end;
```

**InjectionOfCofactorOfDirectSumWithGivenDirectSum.** Back to index.

```
function ( product_object, component_number, direct_sum_object )
    local  underlying_objects, underlying_direct_sum, injection;
    underlying_objects
     := List( product_object, UnderlyingPresentationObject );
    underlying_direct_sum := UnderlyingPresentationObject(
       direct_sum_object );
    injection := InjectionOfCofactorOfDirectSumWithGivenDirectSum(
       underlying_objects, component_number, underlying_direct_sum );
    return
     GradedPresentationMorphism( product_object[component_number],
       injection, direct_sum_object );
end;
```

**UniversalMorphismFromDirectSumWithGivenDirectSum.** Back to index.

```
function ( diagram, product_morphism, direct_sum )
    local  underlying_diagram, underlying_product_morphism,
    underlying_direct_sum, universal_morphism;
    underlying_diagram
     := List( diagram, UnderlyingPresentationObject );
    underlying_product_morphism
     := List( product_morphism, UnderlyingPresentationMorphism );
    underlying_direct_sum := UnderlyingPresentationObject(
       direct_sum );
    universal_morphism
     := UniversalMorphismFromDirectSumWithGivenDirectSum(
       underlying_diagram, underlying_product_morphism,
       underlying_direct_sum );
    return GradedPresentationMorphism( direct_sum,
       universal_morphism, Range( product_morphism[1] ) );
end;
```

**IsCongruentForMorphisms.** Back to index.

```
function ( morphism_1, morphism_2 )
    return
     IsCongruentForMorphisms( UnderlyingPresentationMorphism(
         morphism_1 ), UnderlyingPresentationMorphism( morphism_2 )
       );
end;
```

**IsEqualForMorphisms.** Back to index.

```
function ( morphism_1, morphism_2 )
    return UnderlyingMatrix( morphism_1 )
      = UnderlyingMatrix( morphism_2 );
end;
```

**IsEqualForObjects.** Back to index.

```
function ( object1, object2 )
    if UnderlyingMatrix( object1 ) = UnderlyingMatrix( object2 )
         then
        return GeneratorDegrees( object1 )
          = GeneratorDegrees( object2 );
    fi;
    return false;
end;
```

**IsEqualForCacheForObjects.** Back to index.

```
IsIdenticalObj
```

**IsEqualForCacheForMorphisms.** Back to index.

```
IsIdenticalObj
```

**AdditionForMorphisms.** Back to index.

```
function ( morphism_1, morphism_2 )
    return GradedPresentationMorphism( Source( morphism_1 ),
       UnderlyingPresentationMorphism( morphism_1 )
        + UnderlyingPresentationMorphism( morphism_2 ),
       Range( morphism_1 ) );
end;
```

**AdditiveInverseForMorphisms.** Back to index.

```
function ( morphism_1 )
    return GradedPresentationMorphism( Source( morphism_1 ),
       - UnderlyingPresentationMorphism( morphism_1 ),
       Range( morphism_1 ) );
end;
```

**IsWellDefinedForMorphisms.** Back to index.

```
function ( morphism )
    local  matrix_degrees, matrix_entries, source_degrees,
    range_degrees;
    if
     not IsWellDefined( UnderlyingPresentationMorphism( morphism ) )
       then
        return false;
    fi;
    matrix_degrees
     :=
      TransposedMat( DegreesOfEntries( UnderlyingMatrix( morphism )
         ) );
    matrix_entries
     := TransposedMat( EntriesOfHomalgMatrixAsListList(
         UnderlyingMatrix( morphism ) ) );
    source_degrees := GeneratorDegrees( Source( morphism ) );
    range_degrees := GeneratorDegrees( Range( morphism ) );
    return GeneratorDegrees( Source( morphism ) )
      = NonTrivialDegreePerColumn( UnderlyingMatrix( morphism ),
         GeneratorDegrees( Range( morphism ) ) );
end;
```

**IsWellDefinedForObjects.** Back to index.

```
function ( object )
    local  relation_degrees, generator_degrees, relation_entries;
    if not IsWellDefined( UnderlyingPresentationObject( object ) )
         then
        return false;
    fi;
    relation_degrees
     := TransposedMat( DegreesOfEntries( UnderlyingMatrix( object )
         ) );
    relation_entries
     := TransposedMat( EntriesOfHomalgMatrixAsListList(
         UnderlyingMatrix( object ) ) );
    generator_degrees := GeneratorDegrees( object );
    return
     CAP_INTERNAL_CHECK_DEGREES_FOR_IS_WELL_DEFINED_FOR_OBJECTS(
       relation_degrees, relation_entries, generator_degrees );
end;
```

**TensorProductOnObjects.** Back to index.

```
function ( object_1, object_2 )
    local  new_object, degrees_1, degrees_2, new_degrees, i, j;
    new_object
     := TensorProductOnObjects(
       UnderlyingPresentationObject( object_1 ),
       UnderlyingPresentationObject( object_2 ) );
    degrees_1 := GeneratorDegrees( object_1 );
    degrees_2 := GeneratorDegrees( object_2 );
    new_degrees := [  ];
    for i  in [ 1 .. Length( degrees_1 ) ]  do
        for j  in [ 1 .. Length( degrees_2 ) ]  do
            Add( new_degrees, degrees_1[i] + degrees_2[j] );
        od;
    od;
    return object_constructor( new_object, new_degrees );
end;
```

**TensorProductOnMorphismsWithGivenTensorProducts.** Back to index.

```
function ( new_source, morphism_1, morphism_2, new_range )
    local  new_morphism;
    new_morphism := TensorProductOnMorphismsWithGivenTensorProducts
        ( UnderlyingPresentationObject( new_source ),
       UnderlyingPresentationMorphism( morphism_1 ),
       UnderlyingPresentationMorphism( morphism_2 ),
       UnderlyingPresentationObject( new_range ) );
    return GradedPresentationMorphism( new_source, new_morphism,
       new_range );
end;
```

## 6. Primitive operations for generalized morphisms by cospans

**IdentityMorphism.** Back to index.

```
function ( generalized_object )
    local  identity_morphism;
    identity_morphism
     := IdentityMorphism(
       UnderlyingHonestObject( generalized_object ) );
    return AsGeneralizedMorphismByCospan( identity_morphism );
end;
```

**PreCompose.** Back to index.

```
function ( morphism1, morphism2 )
    local  pushout_diagram, injection_left, injection_right;
    pushout_diagram
     := [ ReversedArrow( morphism1 ), Arrow( morphism2 ) ];
    injection_left := InjectionOfCofactorOfPushout(
       pushout_diagram, 1 );
    injection_right := InjectionOfCofactorOfPushout(
       pushout_diagram, 2 );
    return
     GeneralizedMorphismByCospan( PreCompose( Arrow( morphism1 ),
         injection_left ), PreCompose( ReversedArrow( morphism2 ),
         injection_right ) );
end;
```

```
function ( morphism1, morphism2 )
    local  arrow, reversed_arrow;
    arrow := PreCompose( Arrow( morphism1 ), Arrow( morphism2 ) );
    return AsGeneralizedMorphismByCospan( arrow );
end;
```

This function uses the following extra filters:

- HasIdentityAsReversedArrow for the 1st argument.
- HasIdentityAsReversedArrow for the 2nd argument.

```
function ( morphism1, morphism2 )
    local  arrow;
    arrow := PreCompose( Arrow( morphism1 ), Arrow( morphism2 ) );
    return GeneralizedMorphismByCospan( arrow,
       ReversedArrow( morphism2 ) );
end;
```

This function uses the following extra filters:

- HasIdentityAsReversedArrow for the 1st argument.

**ZeroMorphism.** Back to index.

```
function ( obj1, obj2 )
    local  morphism;
    morphism := ZeroMorphism( UnderlyingHonestObject( obj1 ),
       UnderlyingHonestObject( obj2 ) );
    return AsGeneralizedMorphismByCospan( morphism );
end;
```

**IsCongruentForMorphisms.** Back to index.

```
function ( morphism1, morphism2 )
    local  arrow_tuple, pullback_diagram1, pullback_diagram2,
    subobject1, subobject2;
    arrow_tuple := [ Arrow( morphism1 ), ReversedArrow( morphism1 )
       ];
    pullback_diagram1
     := [ ProjectionInFactorOfFiberProduct( arrow_tuple, 1 ),
        ProjectionInFactorOfFiberProduct( arrow_tuple, 2 ) ];
    arrow_tuple := [ Arrow( morphism2 ), ReversedArrow( morphism2 )
       ];
    pullback_diagram2
     := [ ProjectionInFactorOfFiberProduct( arrow_tuple, 1 ),
        ProjectionInFactorOfFiberProduct( arrow_tuple, 2 ) ];
    subobject1 := UniversalMorphismIntoDirectSum( pullback_diagram1
       );
    subobject2 := UniversalMorphismIntoDirectSum( pullback_diagram2
       );
    return IsEqualAsSubobjects( subobject1, subobject2 );
end;
```

**IsEqualForObjects.** Back to index.

```
function ( object_1, object_2 )
    return IsEqualForObjects( UnderlyingHonestObject( object_1 ),
       UnderlyingHonestObject( object_2 ) );
end;
```

**IsEqualForCacheForObjects.** Back to index.

```
IsIdenticalObj
```

**IsEqualForCacheForMorphisms.** Back to index.

```
IsIdenticalObj
```

**AdditionForMorphisms.** Back to index.

```
function ( morphism1, morphism2 )
    local  pushout_diagram, pushout_left, pushout_right, arrow,
    reversed_arrow;
    pushout_diagram
     := [ ReversedArrow( morphism1 ), ReversedArrow( morphism2 ) ];
    pushout_left := InjectionOfCofactorOfPushout( pushout_diagram,
       1 );
    pushout_right := InjectionOfCofactorOfPushout( pushout_diagram,
       2 );
    arrow := PreCompose( Arrow( morphism1 ), pushout_left )
      + PreCompose( Arrow( morphism2 ), pushout_right );
    reversed_arrow := PreCompose( pushout_diagram[1], pushout_left );
    return GeneralizedMorphismByCospan( arrow, reversed_arrow );
end;
```

```
function ( morphism1, morphism2 )
    return AsGeneralizedMorphismByCospan( Arrow( morphism1 )
        + Arrow( morphism2 ) );
end;
```

This function uses the following extra filters:

- HasIdentityAsReversedArrow for the 1st argument.
- HasIdentityAsReversedArrow for the 2nd argument.

**AdditiveInverseForMorphisms.** Back to index.

```
function ( morphism )
    return GeneralizedMorphismByCospan( - Arrow( morphism ),
       ReversedArrow( morphism ) );
end;
```

```
function ( morphism )
    return AsGeneralizedMorphismByCospan( - Arrow( morphism ) );
end;
```

This function uses the following extra filters:

- HasIdentityAsReversedArrow for the 1st argument.

**IsWellDefinedForMorphisms.** Back to index.

```
function ( generalized_morphism )
    local  category;
    category := CapCategory( Arrow( generalized_morphism ) );
    if
     not ForAll(
            [ Arrow( generalized_morphism ),
               ReversedArrow( generalized_morphism ) ],
            function ( x )
                 return IsIdenticalObj( CapCategory( x ), category );
             end )  then
        return false;
    fi;
    if
     not ForAll(
            [ Arrow( generalized_morphism ),
               ReversedArrow( generalized_morphism ) ],
            IsWellDefined )  then
        return false;
    fi;
    return true;
end;
```

**IsWellDefinedForObjects.** Back to index.

```
function ( object )
    return IsWellDefined( UnderlyingHonestObject( object ) );
end;
```

## 7. Primitive operations for generalized morphisms by spans

**IdentityMorphism.** Back to index.

```
function ( generalized_object )
    local  identity_morphism;
    identity_morphism
     := IdentityMorphism(
       UnderlyingHonestObject( generalized_object ) );
    return AsGeneralizedMorphismBySpan( identity_morphism );
end;
```

**PreCompose.** Back to index.

```
function ( morphism1, morphism2 )
    local  pullback_diagram, projection_left, projection_right;
    pullback_diagram
     := [ Arrow( morphism1 ), ReversedArrow( morphism2 ) ];
    projection_left := ProjectionInFactorOfFiberProduct(
       pullback_diagram, 1 );
    projection_right := ProjectionInFactorOfFiberProduct(
       pullback_diagram, 2 );
    return
     GeneralizedMorphismBySpan(
       PreCompose( projection_left, ReversedArrow( morphism1 ) ),
       PreCompose( projection_right, Arrow( morphism2 ) ) );
end;
```

```
function ( morphism1, morphism2 )
    local  arrow;
    arrow := PreCompose( Arrow( morphism1 ), Arrow( morphism2 ) );
    return AsGeneralizedMorphismBySpan( arrow );
end;
```

This function uses the following extra filters:

- HasIdentityAsReversedArrow for the 1st argument.
- HasIdentityAsReversedArrow for the 2nd argument.

```
function ( morphism1, morphism2 )
    local  arrow;
    arrow := PreCompose( Arrow( morphism1 ), Arrow( morphism2 ) );
    return GeneralizedMorphismBySpan( ReversedArrow( morphism1 ),
       arrow );
end;
```

This function uses the following extra filters:

- HasIdentityAsReversedArrow for the 2nd argument.

**ZeroMorphism.** Back to index.

```
function ( obj1, obj2 )
    local  morphism;
    morphism := ZeroMorphism( UnderlyingHonestObject( obj1 ),
       UnderlyingHonestObject( obj2 ) );
    return AsGeneralizedMorphismBySpan( morphism );
end;
```

**IsCongruentForMorphisms.** Back to index.

```
function ( morphism1, morphism2 )
    local  arrow_tuple, pushout_diagram1, pushout_diagram2,
    factorobject1, factorobject2;
    arrow_tuple := [ Arrow( morphism1 ), ReversedArrow( morphism1 )
       ];
    pushout_diagram1
     := [ InjectionOfCofactorOfPushout( arrow_tuple, 1 ),
        InjectionOfCofactorOfPushout( arrow_tuple, 2 ) ];
    arrow_tuple := [ Arrow( morphism2 ), ReversedArrow( morphism2 )
```

```
      ];
    pushout_diagram2
     := [ InjectionOfCofactorOfPushout( arrow_tuple, 1 ),
        InjectionOfCofactorOfPushout( arrow_tuple, 2 ) ];
    factorobject1 := UniversalMorphismFromDirectSum(
       pushout_diagram1 );
    factorobject2 := UniversalMorphismFromDirectSum(
       pushout_diagram2 );
    return IsEqualAsFactorobjects( factorobject1, factorobject2 );
end;
```

**IsEqualForObjects.** Back to index.

```
function ( object_1, object_2 )
    return IsEqualForObjects( UnderlyingHonestObject( object_1 ),
       UnderlyingHonestObject( object_2 ) );
end;
```

**IsEqualForCacheForObjects.** Back to index.

```
IsIdenticalObj
```

**IsEqualForCacheForMorphisms.** Back to index.

```
IsIdenticalObj
```

**AdditionForMorphisms.** Back to index.

```
function ( morphism1, morphism2 )
    local  pullback_diagram, pullback_left, pullback_right, arrow,
    reversed_arrow;
    pullback_diagram
     := [ ReversedArrow( morphism1 ), ReversedArrow( morphism2 ) ];
    pullback_left := ProjectionInFactorOfFiberProduct(
       pullback_diagram, 1 );
    pullback_right := ProjectionInFactorOfFiberProduct(
       pullback_diagram, 2 );
    arrow := PreCompose( pullback_left, Arrow( morphism1 ) )
      + PreCompose( pullback_right, Arrow( morphism2 ) );
    reversed_arrow
     := PreCompose( pullback_left, pullback_diagram[1] );
    return GeneralizedMorphismBySpan( reversed_arrow, arrow );
end;
```

```
function ( morphism1, morphism2 )
    return AsGeneralizedMorphismBySpan( Arrow( morphism1 )
        + Arrow( morphism2 ) );
end;
```

This function uses the following extra filters:

- HasIdentityAsReversedArrow for the 1st argument.
- HasIdentityAsReversedArrow for the 2nd argument.

**AdditiveInverseForMorphisms.** Back to index.

```
function ( morphism )
    return GeneralizedMorphismBySpan( ReversedArrow( morphism ),
       - Arrow( morphism ) );
end;
```

```
function ( morphism )
    return AsGeneralizedMorphismBySpan( - Arrow( morphism ) );
end;
```

This function uses the following extra filters:

- HasIdentityAsReversedArrow for the 1st argument.

**IsWellDefinedForMorphisms.** Back to index.

```
function ( generalized_morphism )
    local  category;
    category := CapCategory( Arrow( generalized_morphism ) );
    if
     not ForAll(
            [ Arrow( generalized_morphism ),
               ReversedArrow( generalized_morphism ) ],
            function ( x )
                return IsIdenticalObj( CapCategory( x ), category );
             end )  then
        return false;
    fi;
    if
     not ForAll(
            [ Arrow( generalized_morphism ),
               ReversedArrow( generalized_morphism ) ],
            IsWellDefined )  then
        return false;
    fi;
    return true;
end;
```

**IsWellDefinedForObjects.** Back to index.

```
function ( object )
    return IsWellDefined( UnderlyingHonestObject( object ) );
end;
```

## 8. Primitive operations for generalized morphisms by three arrows

**IdentityMorphism.** Back to index.

```
function ( generalized_object )
    local  identity_morphism;
    identity_morphism
     := IdentityMorphism(
       UnderlyingHonestObject( generalized_object ) );
    return AsGeneralizedMorphismByThreeArrows( identity_morphism );
end;
```

**PreCompose.** Back to index.

```
function ( mor1, mor2 )
    return GeneralizedMorphismByThreeArrows( SourceAid( mor1 ),
       PreCompose( Arrow( mor1 ), Arrow( mor2 ) ), RangeAid( mor2 )
       );
end;
```

This function uses the following extra filters:

- HasIdentityAsRangeAid for the 1st argument.
- HasIdentityAsSourceAid for the 2nd argument.

```
function ( mor1, mor2 )
    local  category, pullback_diagram, new_source_aid,
    new_morphism_aid;
    pullback_diagram := [ Arrow( mor1 ), SourceAid( mor2 ) ];
    new_source_aid
     :=
      PreCompose( ProjectionInFactorOfFiberProduct( pullback_diagram
          , 1 ), SourceAid( mor1 ) );
    new_morphism_aid
     :=
      PreCompose( ProjectionInFactorOfFiberProduct( pullback_diagram
          , 2 ), Arrow( mor2 ) );
    return GeneralizedMorphismByThreeArrowsWithSourceAid(
       new_source_aid, new_morphism_aid );
end;
```

This function uses the following extra filters:

- HasIdentityAsRangeAid for the 1st argument.
- HasIdentityAsRangeAid for the 2nd argument.

```
function ( mor1, mor2 )
    local  category, diagram, injection_of_cofactor1,
    injection_of_cofactor2, new_morphism_aid, new_range_aid;
    diagram := [ RangeAid( mor1 ), Arrow( mor2 ) ];
    injection_of_cofactor1 := InjectionOfCofactorOfPushout(
       diagram, 1 );
    injection_of_cofactor2 := InjectionOfCofactorOfPushout(
       diagram, 2 );
    new_morphism_aid
     := PreCompose( Arrow( mor1 ), injection_of_cofactor1 );
    new_range_aid := PreCompose( RangeAid( mor2 ),
       injection_of_cofactor2 );
    return GeneralizedMorphismByThreeArrowsWithRangeAid(
       new_morphism_aid, new_range_aid );
end;
```

This function uses the following extra filters:

- HasIdentityAsSourceAid for the 1st argument.
- HasIdentityAsSourceAid for the 2nd argument.

```
function ( mor1, mor2 )
    local  category;
    return AsGeneralizedMorphismByThreeArrows(
       PreCompose( Arrow( mor1 ), Arrow( mor2 ) ) );
end;
```

This function uses the following extra filters:

- HasIdentityAsSourceAid for the 1st argument.
- HasIdentityAsSourceAid for the 2nd argument.

```
function ( mor1, mor2 )
    local  generalized_mor_factor_sub, pullback_diagram,
    pushout_diagram, new_associated, new_source_aid, new_range_aid;
    generalized_mor_factor_sub
     := GeneralizedMorphismFromFactorToSubobjectByThreeArrows(
       RangeAid( mor1 ), SourceAid( mor2 ) );
    pullback_diagram
     := [ Arrow( mor1 ), SourceAid( generalized_mor_factor_sub ) ];
    pushout_diagram
     := [ RangeAid( generalized_mor_factor_sub ), Arrow( mor2 ) ];
    new_source_aid
     :=
      PreCompose( ProjectionInFactorOfFiberProduct( pullback_diagram
          , 1 ), SourceAid( mor1 ) );
    new_associated
     :=
      PreCompose( ProjectionInFactorOfFiberProduct( pullback_diagram
          , 2 ), InjectionOfCofactorOfPushout( pushout_diagram, 1 )
        );
    new_range_aid := PreCompose( RangeAid( mor2 ),
       InjectionOfCofactorOfPushout( pushout_diagram, 2 ) );
    return GeneralizedMorphismByThreeArrows( new_source_aid,
       new_associated, new_range_aid );
end;
```

**IsCongruentForMorphisms.** Back to index.

```
function ( generalized_morphism1, generalized_morphism2 )
    local  subobject1, subobject2, factorobject1, factorobject2,
    isomorphism_of_subobjects, isomorphism_of_factorobjects;
    subobject1 := DomainOfGeneralizedMorphism(
       generalized_morphism1 );
    subobject2 := DomainOfGeneralizedMorphism(
       generalized_morphism2 );
    if not IsEqualAsSubobjects( subobject1, subobject2 )  then
        return false;
    fi;
    factorobject1 := Codomain( generalized_morphism1 );
    factorobject2 := Codomain( generalized_morphism2 );
    if not IsEqualAsFactorobjects( factorobject1, factorobject2 )
         then
        return false;
    fi;
    isomorphism_of_subobjects := LiftAlongMonomorphism( subobject2,
       subobject1 );
    isomorphism_of_factorobjects
     := ColiftAlongEpimorphism( factorobject2, factorobject1 );
    return
     IsCongruentForMorphisms(
       AssociatedMorphism( generalized_morphism1 ),
       PreCompose( PreCompose( isomorphism_of_subobjects,
           AssociatedMorphism( generalized_morphism2 ) ),
         isomorphism_of_factorobjects ) );
end;
```

**IsEqualForObjects.** Back to index.

```
function ( object_1, object_2 )
    return IsEqualForObjects( UnderlyingHonestObject( object_1 ),
       UnderlyingHonestObject( object_2 ) );
end;
```

**IsEqualForCacheForObjects.** Back to index.

```
IsIdenticalObj
```

**IsEqualForCacheForMorphisms.** Back to index.

```
IsIdenticalObj
```

**AdditionForMorphisms.** Back to index.

```
function ( mor1, mor2 )
    local  return_value, pullback_of_sourceaids_diagram,
    pushout_of_rangeaids_diagram, restricted_mor1, restricted_mor2;
    pullback_of_sourceaids_diagram
     := [ SourceAid( mor1 ), SourceAid( mor2 ) ];
    pushout_of_rangeaids_diagram
     := [ RangeAid( mor1 ), RangeAid( mor2 ) ];
    restricted_mor1
     :=
      PreCompose( ProjectionInFactorOfFiberProduct(
         pullback_of_sourceaids_diagram, 1 ), Arrow( mor1 ) );
    restricted_mor1 := PreCompose( restricted_mor1,
       InjectionOfCofactorOfPushout( pushout_of_rangeaids_diagram,
         1 ) );
    restricted_mor2
     :=
      PreCompose( ProjectionInFactorOfFiberProduct(
         pullback_of_sourceaids_diagram, 2 ), Arrow( mor2 ) );
    restricted_mor2 := PreCompose( restricted_mor2,
       InjectionOfCofactorOfPushout( pushout_of_rangeaids_diagram,
         2 ) );
    return_value := GeneralizedMorphismByThreeArrows(
       PreCompose( ProjectionInFactorOfFiberProduct(
           pullback_of_sourceaids_diagram, 1 ), SourceAid( mor1 ) )
```

```
        , restricted_mor1 + restricted_mor2,
       PreCompose( RangeAid( mor1 ),
         InjectionOfCofactorOfPushout( pushout_of_rangeaids_diagram
            , 1 ) ) );
    return return_value;
end;
```

**IsWellDefinedForMorphisms.** Back to index.

```
function ( generalized_morphism )
    local  category;
    category := CapCategory( SourceAid( generalized_morphism ) );
    if
     not ForAll(
            [ Arrow( generalized_morphism ),
               RangeAid( generalized_morphism ) ], function ( x )
                 return IsIdenticalObj( CapCategory( x ), category );
             end )  then
        return false;
    fi;
    if
     not ForAll(
            [ SourceAid( generalized_morphism ),
               Arrow( generalized_morphism ),
               RangeAid( generalized_morphism ) ], IsWellDefined )
         then
        return false;
    fi;
    return true;
end;
```

**IsWellDefinedForObjects.** Back to index.

```
function ( object )
    return IsWellDefined( UnderlyingHonestObject( object ) );
end;
```

## 9. Primitive operations for Serre quotient by cospans

**IdentityMorphism.** Back to index.

```
function ( object )
    return AsSerreQuotientCategoryByCospansMorphism( category,
        IdentityMorphism( UnderlyingHonestObject( object ) ) );
end;
```

**KernelEmbedding.** Back to index.

```
function ( morphism )
    local  underlying_general, kernel_mor;
    underlying_general := UnderlyingGeneralizedMorphism( morphism );
    kernel_mor := KernelEmbedding( Arrow( underlying_general ) );
    return AsSerreQuotientCategoryByCospansMorphism( category,
       kernel_mor );
end;
```

**CokernelProjection.** Back to index.

```
function ( morphism )
    local  underlying_general, cokernel_mor, triple;
    underlying_general := UnderlyingGeneralizedMorphism( morphism );
    triple := DomainAssociatedMorphismCodomainTriple(
       underlying_general );
    cokernel_mor := CokernelProjection( triple[2] );
    return AsSerreQuotientCategoryByCospansMorphism( category,
       PreCompose( triple[3], cokernel_mor ) );
end;
```

**ZeroObject.** Back to index.

```
function (  )
    local  generalized_zero;
    generalized_zero
     := ZeroObject( UnderlyingHonestCategory( category ) );
    return AsSerreQuotientCategoryByCospansObject( category,
       generalized_zero );
end;
```

**LiftAlongMonomorphism.** Back to index.

```
function ( monomorphism, test_morphism )
    local  inverse_of_mono, composition;
    inverse_of_mono
     := PseudoInverse( UnderlyingGeneralizedMorphism( monomorphism
         ) );
    composition
     := PreCompose( UnderlyingGeneralizedMorphism( test_morphism ),
       inverse_of_mono );
    return SerreQuotientCategoryByCospansMorphism( category,
       composition );
end;
```

**ColiftAlongEpimorphism.** Back to index.

```
function ( epimorphism, test_morphism )
    local  inverse_of_epi, composition;
    inverse_of_epi
     := PseudoInverse( UnderlyingGeneralizedMorphism( epimorphism )
       );
    composition
     := PreCompose( inverse_of_epi,
       UnderlyingGeneralizedMorphism( test_morphism ) );
    return SerreQuotientCategoryByCospansMorphism( category,
       composition );
end;
```

**PreCompose.** Back to index.

```
function ( morphism1, morphism2 )
    local  composition;
    composition
     := PreCompose( UnderlyingGeneralizedMorphism( morphism1 ),
       UnderlyingGeneralizedMorphism( morphism2 ) );
    return SerreQuotientCategoryByCospansMorphism( category,
       composition );
end;
```

**ZeroMorphism.** Back to index.

```
function ( source, range )
    local  new_general;
    new_general
     := ZeroMorphism( UnderlyingGeneralizedObject( source ),
       UnderlyingGeneralizedObject( range ) );
    return SerreQuotientCategoryByCospansMorphism( category,
       new_general );
end;
```

**DirectSum.** Back to index.

```
function ( obj_list )
    local  honest_list, honest_sum;
    honest_list := List( obj_list, UnderlyingHonestObject );
    honest_sum := CallFuncList( DirectSum, honest_list );
    return AsSerreQuotientCategoryByCospansObject( category,
       honest_sum );
end;
```

**ProjectionInFactorOfDirectSumWithGivenDirectSum.** Back to index.

```
function ( product_object, component_number, direct_sum_object )
    local  underlying_objects, underlying_direct_sum,
    honest_projection;
    underlying_objects
     := List( product_object, UnderlyingHonestObject );
    underlying_direct_sum
     := UnderlyingHonestObject( direct_sum_object );
    honest_projection
     := ProjectionInFactorOfDirectSumWithGivenDirectSum(
       underlying_objects, component_number, underlying_direct_sum );
    return AsSerreQuotientCategoryByCospansMorphism( category,
       honest_projection );
end;
```

**UniversalMorphismIntoDirectSum.** Back to index.

```
function ( diagram, morphism_list )
    local  generalized_list, arrow_list, reversedarrow_list,
    new_arrow, new_reversed_arrow, object_list;
    generalized_list
     := List( morphism_list, UnderlyingGeneralizedMorphism );
    arrow_list := List( generalized_list, Arrow );
    new_arrow := UniversalMorphismIntoDirectSum(
       List( arrow_list, Range ), arrow_list );
    reversedarrow_list := List( generalized_list, ReversedArrow );
    new_reversed_arrow := DirectSumFunctorial( reversedarrow_list );
    return SerreQuotientCategoryByCospansMorphism( category,
       new_arrow, new_reversed_arrow );
end;
```

**InjectionOfCofactorOfDirectSumWithGivenDirectSum.** Back to index.

```
function ( object_product_list, injection_number, direct_sum_object
     )
    local  underlying_objects, underlying_direct_sum,
    honest_injection;
    underlying_objects
     := List( object_product_list, UnderlyingHonestObject );
    underlying_direct_sum
     := UnderlyingHonestObject( direct_sum_object );
    honest_injection
     := AddInjectionOfCofactorOfDirectSumWithGivenDirectSum(
       underlying_objects, injection_number, underlying_direct_sum );
    return AsSerreQuotientCategoryByCospansMorphism( category,
       honest_injection );
end;
```

**UniversalMorphismFromDirectSum.** Back to index.

```
function ( diagram, morphism_list )
    local  generalized_list, arrow_list, reversedarrow_list,
    new_arrow, new_reversed_arrow, object_list;
    generalized_list
     := List( morphism_list, UnderlyingGeneralizedMorphism );
    generalized_list := CommonCoastriction( generalized_list );
    arrow_list := List( generalized_list, Arrow );
    new_arrow := UniversalMorphismFromDirectSum(
       List( diagram, UnderlyingHonestObject ), arrow_list );
    new_reversed_arrow := ReversedArrow( generalized_list[1] );
    return SerreQuotientCategoryByCospansMorphism( category,
       new_arrow, new_reversed_arrow );
end;
```

**IsCongruentForMorphisms.** Back to index.

```
function ( morphism1, morphism2 )
    local  underlying_general, new_morphism_aid, new_general,
    sum_general, sum_associated, sum_image;
    new_general := AdditiveInverse( underlying_general );
    sum_general
     := AdditionForMorphisms(
       UnderlyingGeneralizedMorphism( morphism1 ), new_general );
    sum_associated := AssociatedMorphism( sum_general );
    sum_image := ImageObject( sum_associated );
    return membership_function( sum_image );
end;
```

**IsEqualForObjects.** Back to index.

```
function ( obj1, obj2 )
    return IsEqualForObjects( UnderlyingHonestObject( obj1 ),
        UnderlyingHonestObject( obj2 ) );
end;
```

**AdditionForMorphisms.** Back to index.

```
function ( morphism1, morphism2 )
    local  sum;
    sum
     := AdditionForMorphisms(
       UnderlyingGeneralizedMorphism( morphism1 ),
       UnderlyingGeneralizedMorphism( morphism2 ) );
    return SerreQuotientCategoryByCospansMorphism( category, sum );
end;
```

**AdditiveInverseForMorphisms.** Back to index.

```
function ( morphism )
    local  new_general;
    new_general := AdditiveInverseForMorphisms(
       UnderlyingGeneralizedMorphism( morphism ) );
    return SerreQuotientCategoryByCospansMorphism( category,
       new_general );
end;
```

**IsZeroForObjects.** Back to index.

```
function ( obj )
    return membership_function( UnderlyingHonestObject( obj ) );
end;
```

## 10. Primitive operations for Serre quotient by spans

**InverseImmutable.** Back to index.

```
function ( morphism )
    local  underlying_general, inverse;
    underlying_general := UnderlyingGeneralizedMorphism( morphism );
    inverse := PseudoInverse( underlying_general );
    return SerreQuotientCategoryBySpansMorphism( category, inverse );
end;
```

**IdentityMorphism.** Back to index.

```
function ( object )
    return AsSerreQuotientCategoryBySpansMorphism( category,
       IdentityMorphism( UnderlyingHonestObject( object ) ) );
end;
```

**KernelEmbedding.** Back to index.

```
function ( morphism )
    local  underlying_general, kernel_mor;
    underlying_general := UnderlyingGeneralizedMorphism( morphism );
    kernel_mor := KernelEmbedding( Arrow( underlying_general ) );
    return AsSerreQuotientCategoryBySpansMorphism( category,
       PreCompose( kernel_mor, ReversedArrow( underlying_general )
         ) );
end;
```

**CokernelProjection.** Back to index.

```
function ( morphism )
    local  underlying_general, cokernel_mor;
    underlying_general := UnderlyingGeneralizedMorphism( morphism );
    cokernel_mor := CokernelProjection( Arrow( underlying_general )
       );
    return AsSerreQuotientCategoryBySpansMorphism( category,
       cokernel_mor );
end;
```

**ZeroObject.** Back to index.

```
function (  )
    local  generalized_zero;
    generalized_zero
     := ZeroObject( UnderlyingHonestCategory( category ) );
    return AsSerreQuotientCategoryBySpansObject( category,
       generalized_zero );
end;
```

**DualOnObjects.** Back to index.

```
function ( object )
    return AsSerreQuotientCategoryBySpansObject( category,
       DualOnObjects( UnderlyingHonestObject( object ) ) );
end;
```

**LiftAlongMonomorphism.** Back to index.

```
function ( monomorphism, test_morphism )
    local  inverse_of_mono, composition;
    inverse_of_mono
     := PseudoInverse( UnderlyingGeneralizedMorphism( monomorphism
         ) );
    composition
     := PreCompose( UnderlyingGeneralizedMorphism( test_morphism ),
       inverse_of_mono );
    return SerreQuotientCategoryBySpansMorphism( category,
       composition );
end;
```

**ColiftAlongEpimorphism.** Back to index.

```
function ( epimorphism, test_morphism )
    local  inverse_of_epi, composition;
    inverse_of_epi
     := PseudoInverse( UnderlyingGeneralizedMorphism( epimorphism )
       );
    composition
     := PreCompose( inverse_of_epi,
       UnderlyingGeneralizedMorphism( test_morphism ) );
    return SerreQuotientCategoryBySpansMorphism( category,
       composition );
end;
```

**Lift.** Back to index.

```
function ( test_morphism, monomorphism )
    local  inverse_of_mono, composition;
    test_morphism := UnderlyingGeneralizedMorphism( test_morphism );
    monomorphism := UnderlyingGeneralizedMorphism( monomorphism );
    if not IsHonest( test_morphism ) or not IsHonest( monomorphism )
       then
        return fail;
    fi;
    test_morphism := HonestRepresentative( test_morphism );
    monomorphism := HonestRepresentative( monomorphism );
    composition := Lift( test_morphism, monomorphism );
    if composition = fail  then
        return fail;
    fi;
    return AsSerreQuotientCategoryBySpansMorphism( category,
       composition );
end;
```

**PreCompose.** Back to index.

```
function ( morphism1, morphism2 )
    local  composition;
    composition
     := PreCompose( UnderlyingGeneralizedMorphism( morphism1 ),
       UnderlyingGeneralizedMorphism( morphism2 ) );
    return SerreQuotientCategoryBySpansMorphism( category,
       composition );
end;
```

**ZeroMorphism.** Back to index.

```
function ( source, range )
    local  new_general;
    new_general := ZeroMorphism( UnderlyingHonestObject( source ),
       UnderlyingHonestObject( range ) );
    return AsSerreQuotientCategoryBySpansMorphism( category,
       new_general );
end;
```

**DirectSum.** Back to index.

```
function ( obj_list )
    local  honest_list, honest_sum;
    honest_list := List( obj_list, UnderlyingHonestObject );
    honest_sum := DirectSum( honest_list );
    return AsSerreQuotientCategoryBySpansObject( category,
       honest_sum );
end;
```

**ProjectionInFactorOfDirectSumWithGivenDirectSum.** Back to index.

```
function ( product_object, component_number, direct_sum_object )
    local  underlying_objects, honest_projection;
    underlying_objects
     := List( product_object, UnderlyingHonestObject );
    honest_projection := ProjectionInFactorOfDirectSum(
       underlying_objects, component_number );
    return AsSerreQuotientCategoryBySpansMorphism( category,
       honest_projection );
end;
```

**UniversalMorphismIntoDirectSum.** Back to index.

```
function ( diagram, morphism_list )
    local  generalized_list, arrow_list, reversedarrow_list,
    new_arrow, new_reversed_arrow, object_list;
    generalized_list
     := List( morphism_list, UnderlyingGeneralizedMorphism );
    generalized_list := CommonRestriction( generalized_list );
    new_reversed_arrow := ReversedArrow( generalized_list[1] );
    arrow_list := List( generalized_list, Arrow );
    new_arrow := UniversalMorphismIntoDirectSum(
       List( diagram, UnderlyingHonestObject ), arrow_list );
    return SerreQuotientCategoryBySpansMorphism( category,
       new_reversed_arrow, new_arrow );
end;
```

**InjectionOfCofactorOfDirectSumWithGivenDirectSum.** Back to index.

```
function ( object_product_list, injection_number, direct_sum_object
     )
    local  underlying_objects, honest_injection;
    underlying_objects
     := List( object_product_list, UnderlyingHonestObject );
    honest_injection := InjectionOfCofactorOfDirectSum(
       underlying_objects, injection_number );
    return AsSerreQuotientCategoryBySpansMorphism( category,
       honest_injection );
end;
```

**UniversalMorphismFromDirectSum.** Back to index.

```
function ( diagram, morphism_list )
    local  generalized_list, arrow_list, reversedarrow_list,
    new_arrow, new_reversed_arrow, object_list;
    generalized_list
     := List( morphism_list, UnderlyingGeneralizedMorphism );
    arrow_list := List( generalized_list, Arrow );
    reversedarrow_list := List( generalized_list, ReversedArrow );
    new_arrow := UniversalMorphismFromDirectSum(
       List( arrow_list, Source ), arrow_list );
    new_reversed_arrow := DirectSumFunctorial( reversedarrow_list );
    return SerreQuotientCategoryBySpansMorphism( category,
       new_reversed_arrow, new_arrow );
end;
```

**IsCongruentForMorphisms.** Back to index.

```
function ( morphism1, morphism2 )
    local  underlying_general, new_general, sum_general,
    sum_associated, sum_image;
    underlying_general := UnderlyingGeneralizedMorphism( morphism2 );
    new_general := AdditiveInverse( underlying_general );
    sum_general
     := AdditionForMorphisms(
       UnderlyingGeneralizedMorphism( morphism1 ), new_general );
    sum_associated := AssociatedMorphism( sum_general );
    sum_image := ImageObject( sum_associated );
    return membership_function( sum_image );
end;
```

**IsEqualForObjects.** Back to index.

```
function ( obj1, obj2 )
    return IsEqualForObjects( UnderlyingHonestObject( obj1 ),
       UnderlyingHonestObject( obj2 ) );
end;
```

**IsEqualForCacheForObjects.** Back to index.

```
IsIdenticalObj
```

**IsEqualForCacheForMorphisms.** Back to index.

```
IsIdenticalObj
```

**AdditionForMorphisms.** Back to index.

```
function ( morphism1, morphism2 )
    local  underlying_generalized, common_restriction, new_arrow;
    underlying_generalized := List( [ morphism1, morphism2 ],
       UnderlyingGeneralizedMorphism );
    common_restriction := CommonRestriction( underlying_generalized
       );
    new_arrow := Arrow( common_restriction[1] )
      + Arrow( common_restriction[2] );
    return SerreQuotientCategoryBySpansMorphism( category,
       ReversedArrow( common_restriction[1] ), new_arrow );
end;
```

**AdditiveInverseForMorphisms.** Back to index.

```
function ( morphism )
    local  general;
    general := UnderlyingGeneralizedMorphism( morphism );
    return SerreQuotientCategoryBySpansMorphism( category,
       ReversedArrow( general ), - Arrow( general ) );
end;
```

**IsZeroForObjects.** Back to index.

```
function ( obj )
    return membership_function( UnderlyingHonestObject( obj ) );
end;
```

**DualOnMorphismsWithGivenDuals.** Back to index.

```
function ( new_source, morphism, new_range )
    local  arrow, reversed_arrow, new_arrow, new_reversed_arrow;
    arrow := Arrow( UnderlyingGeneralizedMorphism( morphism ) );
    reversed_arrow
     := ReversedArrow( UnderlyingGeneralizedMorphism( morphism ) );
    arrow := DualOnMorphisms( arrow );
    reversed_arrow := DualOnMorphisms( reversed_arrow );
    new_reversed_arrow := ProjectionInFactorOfFiberProduct(
       [ reversed_arrow, arrow ], 2 );
    new_arrow := ProjectionInFactorOfFiberProduct(
       [ reversed_arrow, arrow ], 1 );
    return SerreQuotientCategoryBySpansMorphism( category,
       new_reversed_arrow, new_arrow );
end;
```

## 11. Primitive operations for Serre quotient by three arrows

**IdentityMorphism.** Back to index.

```
function ( object )
    return AsSerreQuotientCategoryByThreeArrowsMorphism( category,
       IdentityMorphism( UnderlyingHonestObject( object ) ) );
end;
```

**KernelEmbedding.** Back to index.

```
function ( morphism )
    local  underlying_general, kernel_mor;
    underlying_general := UnderlyingGeneralizedMorphism( morphism );
    kernel_mor
     := KernelEmbedding( AssociatedMorphism( underlying_general ) );
    kernel_mor
     := PreCompose( kernel_mor,
       DomainOfGeneralizedMorphism( underlying_general ) );
    return AsSerreQuotientCategoryByThreeArrowsMorphism( category,
       kernel_mor );
end;
```

**CokernelProjection.** Back to index.

```
function ( morphism )
    local  underlying_general, cokernel_mor;
    underlying_general := UnderlyingGeneralizedMorphism( morphism );
    cokernel_mor
     := CokernelProjection( AssociatedMorphism( underlying_general
         ) );
    cokernel_mor := PreCompose( Codomain( underlying_general ),
       cokernel_mor );
    return AsSerreQuotientCategoryByThreeArrowsMorphism( category,
       cokernel_mor );
end;
```

**ZeroObject.** Back to index.

```
function (  )
    local  generalized_zero;
    generalized_zero
     := ZeroObject( UnderlyingHonestCategory( category ) );
    return AsSerreQuotientCategoryByThreeArrowsObject( category,
       generalized_zero );
end;
```

**LiftAlongMonomorphism.** Back to index.

```
function ( monomorphism, test_morphism )
    local  inverse_of_mono, composition;
    inverse_of_mono
     := PseudoInverse( UnderlyingGeneralizedMorphism( monomorphism
         ) );
    composition
     := PreCompose( UnderlyingGeneralizedMorphism( test_morphism ),
       inverse_of_mono );
    return SerreQuotientCategoryByThreeArrowsMorphism( category,
       composition );
end;
```

**ColiftAlongEpimorphism.** Back to index.

```
function ( epimorphism, test_morphism )
    local  inverse_of_epi, composition;
    inverse_of_epi
     := PseudoInverse( UnderlyingGeneralizedMorphism( epimorphism )
       );
    composition
     := PreCompose( inverse_of_epi,
       UnderlyingGeneralizedMorphism( test_morphism ) );
    return SerreQuotientCategoryByThreeArrowsMorphism( category,
       composition );
end;
```

**PreCompose.** Back to index.

```
function ( morphism1, morphism2 )
    local  composition;
    composition
     := PreCompose( UnderlyingGeneralizedMorphism( morphism1 ),
       UnderlyingGeneralizedMorphism( morphism2 ) );
    return SerreQuotientCategoryByThreeArrowsMorphism( category,
       composition );
end;
```

**ZeroMorphism.** Back to index.

```
function ( source, range )
    local  source_aid, range_aid, morphism_aid;
    source := UnderlyingHonestObject( source );
    range := UnderlyingHonestObject( range );
    source_aid := IdentityMorphism( source );
    range_aid := IdentityMorphism( range );
    morphism_aid := ZeroMorphism( source, range );
    return SerreQuotientCategoryByThreeArrowsMorphism( category,
       source_aid, morphism_aid, range_aid );
end;
```

**DirectSum.** Back to index.

```
function ( obj_list )
    local  honest_list, honest_sum;
    honest_list := List( obj_list, UnderlyingGeneralizedObject );
    honest_sum := CallFuncList( DirectSum, honest_list );
    return AsSerreQuotientCategoryByThreeArrowsObject( category,
       UnderlyingHonestObject( honest_sum ) );
end;
```

**ProjectionInFactorOfDirectSumWithGivenDirectSum.** Back to index.

```
function ( product_object, component_number, direct_sum_object )
    local  underlying_objects, underlying_direct_sum,
    honest_projection;
    underlying_objects
     := List( product_object, UnderlyingHonestObject );
    underlying_direct_sum
     := UnderlyingHonestObject( direct_sum_object );
    honest_projection
     := ProjectionInFactorOfDirectSumWithGivenDirectSum(
       underlying_objects, component_number, underlying_direct_sum );
    return AsSerreQuotientCategoryByThreeArrowsMorphism( category,
       honest_projection );
end;
```

**UniversalMorphismIntoDirectSum.** Back to index.

```
function ( diagram, morphism_list )
    local  generalized_morphisms, source_aid, associated,
    range_aid, associated_list;
    generalized_morphisms
     := List( morphism_list, UnderlyingGeneralizedMorphism );
    generalized_morphisms
     := CommonRestriction( generalized_morphisms );
    generalized_morphisms := List( generalized_morphisms,
       DomainAssociatedMorphismCodomainTriple );
    source_aid := generalized_morphisms[1][1];
    associated_list := List( generalized_morphisms, function ( i )
            return i[2];
        end );
    associated := UniversalMorphismIntoDirectSum( associated_list );
    range_aid
     := DirectSumFunctorial(
       List( generalized_morphisms, function ( i )
              return i[3];
          end ) );
    return SerreQuotientCategoryByThreeArrowsMorphism( category,
       source_aid, associated, range_aid );
end;
```

**InjectionOfCofactorOfDirectSumWithGivenDirectSum.** Back to index.

```
function ( object_product_list, injection_number, direct_sum_object
     )
    local  underlying_objects, underlying_direct_sum,
    honest_injection;
    underlying_objects
     := List( object_product_list, UnderlyingHonestObject );
    underlying_direct_sum
     := UnderlyingHonestObject( direct_sum_object );
    honest_injection
     := AddInjectionOfCofactorOfDirectSumWithGivenDirectSum(
       underlying_objects, injection_number, underlying_direct_sum );
    return AsSerreQuotientCategoryByThreeArrowsMorphism( category,
       honest_injection );
end;
```

**UniversalMorphismFromDirectSum.** Back to index.

```
function ( diagram, morphism_list )
    local  generalized_morphisms, source_aid, associated, range_aid;
    generalized_morphisms
     := List( morphism_list, UnderlyingGeneralizedMorphism );
    generalized_morphisms
     := CommonCoastriction( generalized_morphisms );
    generalized_morphisms := List( generalized_morphisms,
       DomainAssociatedMorphismCodomainTriple );
    range_aid := generalized_morphisms[1][3];
    associated := UniversalMorphismFromDirectSum(
       List( generalized_morphisms, function ( i )
              return i[2];
          end ) );
    source_aid
     := DirectSumFunctorial(
       List( generalized_morphisms, function ( i )
              return i[1];
          end ) );
    return SerreQuotientCategoryByThreeArrowsMorphism( category,
       source_aid, associated, range_aid );
end;
```

**IsCongruentForMorphisms.** Back to index.

```
function ( morphism1, morphism2 )
    local  underlying_general, new_morphism_aid, new_general,
    sum_general, sum_associated, sum_image;
    underlying_general := UnderlyingGeneralizedMorphism( morphism2 );
    new_morphism_aid
     := AdditiveInverse( Arrow( underlying_general ) );
    new_general := GeneralizedMorphismByThreeArrows(
       SourceAid( underlying_general ), new_morphism_aid,
       RangeAid( underlying_general ) );
    sum_general
     := AdditionForMorphisms(
       UnderlyingGeneralizedMorphism( morphism1 ), new_general );
    sum_associated := AssociatedMorphism( sum_general );
    sum_image := ImageObject( sum_associated );
    return membership_function( sum_image );
end;
```

**IsEqualForObjects.** Back to index.

```
function ( obj1, obj2 )
    return IsEqualForObjects( UnderlyingHonestObject( obj1 ),
       UnderlyingHonestObject( obj2 ) );
end;
```

**IsZeroForMorphisms.** Back to index.

```
function ( morphism )
    local  associated, image;
    associated
     := AssociatedMorphism( UnderlyingGeneralizedMorphism( morphism
         ) );
    image := ImageObject( associated );
    return membership_function( image );
end;
```

**AdditionForMorphisms.** Back to index.

```
function ( morphism1, morphism2 )
    local  sum;
    sum
     := AdditionForMorphisms(
       UnderlyingGeneralizedMorphism( morphism1 ),
       UnderlyingGeneralizedMorphism( morphism2 ) );
    return SerreQuotientCategoryByThreeArrowsMorphism( category,
       sum );
end;
```

**AdditiveInverseForMorphisms.** Back to index.

```
function ( morphism )
    local  underlying_general, new_morphism_aid, new_general;
    underlying_general := UnderlyingGeneralizedMorphism( morphism );
    new_morphism_aid
     := AdditiveInverse( Arrow( underlying_general ) );
    new_general := GeneralizedMorphismByThreeArrows(
       SourceAid( underlying_general ), new_morphism_aid,
       RangeAid( underlying_general ) );
    return SerreQuotientCategoryByThreeArrowsMorphism( category,
       new_general );
end;
```

**IsZeroForObjects.** Back to index.

```
function ( obj )
    return membership_function( UnderlyingHonestObject( obj ) );
end;
```

APPENDIX G

# Application code

In this appendix the code for the algorithms used in Chapter VI is displayed.

## 1. Function ResolutionFunctor

```
function ( source_category, kernel_hull_function, complex )
    local  functor, object_function, morphism_function,
    recursion_function, constructor, category_constructor;
    if complex  then
        constructor := AsComplex;
        category_constructor := ComplexCategory;
    else
        constructor := AsCocomplex;
        category_constructor := CocomplexCategory;
    fi;
    functor
     := CapFunctor( Concatenation( "ResolutionFunctor for ",
         Name( source_category ) ), source_category,
       category_constructor( source_category ) );
    recursion_function := function ( morphism )
          local  kernel_emb, kernel, cover;
          kernel_emb := KernelEmbedding( morphism );
          kernel := Source( kernel_emb );
          cover := kernel_hull_function( kernel );
          return PreCompose( cover, kernel_emb );
      end;
    object_function := function ( object )
          local  initial_morphism, z_functor;
          initial_morphism := kernel_hull_function( object );
          z_functor
           := ZFunctorObjectByInitialMorphismAndRecursiveFunction(
             initial_morphism, recursion_function, 0 );
          return constructor( z_functor );
      end;
    AddObjectFunction( functor, object_function );
    return functor;
end;
```

## 2. Function ResolutionFunctorToComplex

```
function ( cat, func )
    return ResolutionFunctor( cat, func, true );
end;
```

## 3. Function ResolutionFunctorToCocomplex

```
function ( cat, func )
    return ResolutionFunctor( cat, func, false );
end;
```

## 4. Function FreeResolutionComplex

```
function ( module )
    return ResolutionTo( module, CoverByProjective, true );
end;
```

## 5. Function FreeResolutionCocomplex

```
function ( module )
    return ResolutionTo( module, CoverByProjective, false );
end;
```

## 6. Function ResolutionTo

```
function ( object, kernel_hull_function, as_complex )
    local  z_functor, complex, object_function, morphism_function,
    complex_constructor, connection_morphism;
    z_functor := ZFunctorObject( ReturnTrue, ReturnTrue,
       CapCategory( object ) );
    if as_complex = true  then
        complex_constructor := AsComplex;
    else
        complex_constructor := AsCocomplex;
    fi;
    complex := complex_constructor( z_functor );
    connection_morphism := kernel_hull_function( object );
    object_function := function ( i )
          return Source( Differential( z_functor, i ) );
```

```
      end;
    morphism_function := function ( i )
          local  kernel;
          if i = 0  then
              return UniversalMorphismIntoZeroObject(
                 Source( connection_morphism ) );
          elif i = -1  then
              kernel := KernelEmbedding( connection_morphism );
              return
               PreCompose( kernel_hull_function( Source( kernel ) )
                  , kernel );
          elif i < -1  then
              kernel
               := KernelEmbedding( Differential( z_functor, i + 1 )
                 );
              return
               PreCompose( kernel_hull_function( Source( kernel ) )
                  , kernel );
          else
              return
               IdentityMorphism( ZeroObject( CapCategory( object )
                   ) );
          fi;
          return;
      end;
    z_functor!.object_func := object_function;
    z_functor!.differential_func := morphism_function;
    return [ complex, connection_morphism ];
end;
```

## 7. Function CAP INTERNAL HORSE SHOE HELPER

```
function ( left_diff_i, right_diff_i, eps_prime, eps, eps_2prime,
    pi, iota )
    local  ker_eps_prime, ker_eps, ker_eps_2prime,
    eps_prime_1_to_ker, eps_2prime_1_to_ker, iota0, pi0,
    ker_eps_prime_to_eps, ker_eps_to_eps_2prime, first_morphism,
    second_morphism, sum_morphism, differential_morphism,
    range_left_diff, range_right_diff;
    ker_eps_prime := KernelEmbedding( eps_prime );
    ker_eps := KernelEmbedding( eps );
    ker_eps_2prime := KernelEmbedding( eps_2prime );
    eps_prime_1_to_ker := KernelLift( eps_prime, left_diff_i );
    eps_2prime_1_to_ker := KernelLift( eps_2prime, right_diff_i );
    range_left_diff := Range( left_diff_i );
    range_right_diff := Range( right_diff_i );
    iota0 := InjectionOfCofactorOfDirectSum(
       [ range_left_diff, range_right_diff ], 1 );
    pi0 := ProjectionInFactorOfDirectSum(
       [ range_left_diff, range_right_diff ], 2 );
    ker_eps_prime_to_eps
     := KernelLift( eps, PreCompose( ker_eps_prime, iota0 ) );
    ker_eps_to_eps_2prime
     := KernelLift( eps_2prime, PreCompose( ker_eps, pi0 ) );
    first_morphism := PreCompose( eps_prime_1_to_ker,
       ker_eps_prime_to_eps );
    second_morphism
     := Lift( eps_2prime_1_to_ker, ker_eps_to_eps_2prime );
    sum_morphism := UniversalMorphismFromDirectSum(
       [ first_morphism, second_morphism ] );
    differential_morphism := PreCompose( sum_morphism, ker_eps );
    return differential_morphism;
end;
```

## 8. Function HorseShoeLemma

```
function ( left_complex, right_complex, eps_prime, iota, pi,
    eps_2prime )
    local  middle_z_functor, middle_complex,
    kernel_resolution_morphism, cokernel_resolution_morphism,
    object_function, helper_function, morphism_function;
    middle_z_functor := ZFunctorObject( ReturnTrue, ReturnTrue,
       CapCategory( eps_prime ) );
    middle_complex := AsCocomplex( middle_z_functor );
    kernel_resolution_morphism := function ( i )
          return InjectionOfCofactorOfDirectSum(
             [ left_complex[i], right_complex[i] ], 1 );
      end;
    cokernel_resolution_morphism := function ( i )
          return
           ProjectionInFactorOfDirectSum(
             [ left_complex[i], right_complex[i] ], 2 );
      end;
    kernel_resolution_morphism
     := CochainMap( left_complex, kernel_resolution_morphism,
       middle_complex );
```

```
cokernel_resolution_morphism
 := CochainMap( middle_complex, cokernel_resolution_morphism,
   right_complex );
object_function := function ( i )
      return DirectSum( [ left_complex[i], right_complex[i] ] );
  end;
morphism_function := function ( i )
      local  eps;
      if i > 0  then
          return
           IdentityMorphism(
             ZeroObject( CapCategory( eps_prime ) ) );
      fi;
      if i = 0  then
          return UniversalMorphismIntoZeroObject(
             middle_complex[0] );
      fi;
      if i = -1  then
          eps := Lift( eps_2prime, pi );
          eps := UniversalMorphismFromDirectSum(
             [ PreCompose( eps_prime, iota ), eps ] );
          return CAP_INTERNAL_HORSE_SHOE_HELPER(
             Differential( left_complex, -1 ),
             Differential( right_complex, -1 ), eps_prime, eps,
             eps_2prime, pi, iota );
      else
          return CAP_INTERNAL_HORSE_SHOE_HELPER(
             Differential( left_complex, i ),
             Differential( right_complex, i ),
             Differential( left_complex, i + 1 ),
             Differential( middle_complex, i + 1 ),
             Differential( right_complex, i + 1 ),
             kernel_resolution_morphism[i + 1],
             cokernel_resolution_morphism[i + 1] );
      fi;
      return;
  end;
middle_z_functor!.object_func := object_function;
middle_z_functor!.differential_func := morphism_function;
return
 [ middle_complex, kernel_resolution_morphism,
        cokernel_resolution_morphism ];
end;
```

## 9. Function CartanEilenbergResolution

```
function ( complex, projective_resolution_function )
    local  object_function, morphism_function, bicomplex,
    helper_function, bicomplex_z_func;
    bicomplex_z_func := ZFunctorObject( ReturnTrue, ReturnTrue,
       CapCategory( complex ) );
    bicomplex := AsCocomplex( bicomplex_z_func );
    helper_function := function ( i )
          local  delta_im1, delta_i, first_morphism,
          second_morphism, first_complex, second_complex,
          horse_shoe, third_morphism, fourth_morphism,
          third_complex, second_horse_shoe, eps;
          delta_im1 := Differential( complex, i - 1 );
          delta_i := Differential( complex, i );
          first_morphism
           := KernelLift( delta_i, ImageEmbedding( delta_im1 ) );
          second_morphism := CokernelProjection( first_morphism );
          first_complex := projective_resolution_function(
             Source( first_morphism ) );
          second_complex := projective_resolution_function(
             Range( second_morphism ) );
          horse_shoe := HorseShoeLemma( first_complex[1],
             second_complex[1], first_complex[2], first_morphism,
             second_morphism, second_complex[2] );
          third_morphism := KernelEmbedding( delta_i );
          fourth_morphism := CoastrictionToImage( delta_i );
          eps := Lift( second_complex[2], second_morphism );
          eps := UniversalMorphismFromDirectSum(
             [ PreCompose( first_complex[2], first_morphism ), eps
               ] );
          third_complex := projective_resolution_function(
             Range( fourth_morphism ) );
          second_horse_shoe := HorseShoeLemma( horse_shoe[1],
             third_complex[1], eps, third_morphism,
             fourth_morphism, third_complex[2] );
          return [ horse_shoe, second_horse_shoe ];
      end;
    object_function := function ( i )
          local  those_boots, return_complex, old_diff_func,
          underlying_cell;
          those_boots := helper_function( i );
          return_complex := those_boots[2][1];
          if i mod 2 = 1  then
              underlying_cell
               := UnderlyingZFunctorCell( return_complex );
              old_diff_func := underlying_cell!.differential_func;
              underlying_cell!.differential_func := function ( i )
                    return - old_diff_func( i );
                end;
          fi;
          return return_complex;
      end;
    morphism_function := function ( i )
          local  those_boots1, those_boots2;
          those_boots1 := helper_function( i );
          those_boots2 := helper_function( i + 1 );
          return PreCompose( those_boots1[2][3],
             PreCompose( those_boots2[1][2], those_boots2[2][2] ) );
      end;
    bicomplex_z_func!.object_func := object_function;
    bicomplex_z_func!.differential_func := morphism_function;
    return bicomplex;
end;
```

## 10. Function DualOnComplex

```
function ( complex )
    local  object_func, morphism_func;
    object_func := function ( i )
          return DualOnObjects( complex[i] );
      end;
    morphism_func := function ( i )
          return DualOnMorphisms( Differential( complex, i + 1 ) );
      end;
    return
     AsCocomplex( ZFunctorObject( object_func, morphism_func,
         UnderlyingCategory( CapCategory( complex ) ) ) );
end;
```

## 11. Function DualOnCocomplex

```
function ( cocomplex )
    local  object_func, morphism_func, id_of_object;
    object_func := function ( i )
          return DualOnObjects( cocomplex[- i] );
      end;
    morphism_func := function ( i )
          return
           DualOnMorphisms( Differential( cocomplex, - i - 1 ) );
      end;
    return
     AsComplex( ZFunctorObject( object_func, morphism_func,
         UnderlyingCategory( CapCategory( cocomplex ) ) ) );
end;
```

## 12. Function DualOnCochainMap

```
function ( cochain_map, new_source, new_range )
    local  id_of_object, morphism_func;
    morphism_func := function ( i )
          return DualOnMorphisms( cochain_map[i] );
      end;
    return ChainMap( new_source, morphism_func, new_range );
end;
```

## 13. Function DualOnCocomplexCocomplex

```
function ( cocomplex )
    local  object_func, morphism_func, id_of_object, new_complex,
    new_z_functor;
    new_z_functor := ZFunctorObject( ReturnTrue, ReturnTrue,
       ComplexCategory(
         UnderlyingCategory(
           UnderlyingCategory( CapCategory( cocomplex ) ) ) ) );
    new_complex := AsComplex( new_z_functor );
    object_func := function ( i )
          return DualOnCocomplex( cocomplex[- i] );
      end;
    new_z_functor!.object_func := object_func;
    morphism_func := function ( i )
          return
           DualOnCochainMap( Differential( cocomplex, - i - 1 ),
           new_complex[i], new_complex[i - 1] );
      end;
    new_z_functor!.differential_func := morphism_func;
    return new_complex;
end;
```

## 14. Function TransposeComplexOfComplex

```
function ( complex )
    local  new_total_complex, new_z_functor, object_func,
    morphism_func;
    new_z_functor := ZFunctorObject( ReturnTrue, ReturnTrue,
       UnderlyingCategory( CapCategory( complex ) ) );
    new_total_complex := AsComplex( new_z_functor );
    object_func := function ( i )
          local  object_func, morphism_func;
          object_func := function ( j )
                return complex[- j][- i];
            end;
          morphism_func := function ( j )
                return Differential( complex, - j )[- i];
            end;
          return
           AsComplex( ZFunctorObject( object_func, morphism_func,
               UnderlyingCategory(
                 UnderlyingCategory( CapCategory( complex ) ) ) ) );
      end;
    morphism_func := function ( i )
          local  morphism_func;
          morphism_func := function ( j )
                return Differential( complex[j], - i );
            end;
          return ChainMap( new_total_complex[- i], morphism_func,
             new_total_complex[- i + 1] );
      end;
    new_z_functor!.object_func := object_func;
    new_z_functor!.differential_func := morphism_func;
    return new_total_complex;
end;
```

## 15. Function ResolutionLength

```
function ( complex )
    local  i;
    i := 0;
    while not IsZero( complex[i] )  do
        i := i + 1;
    od;
    return i;
end;
```

## 16. Function TotalComplexOfBicomplex

```
function ( bicomplex, length )
    local  object_function, morphism_function, z_functor_object,
    new_complex;
    z_functor_object := ZFunctorObject( ReturnTrue, ReturnTrue,
       UnderlyingCategory(
         UnderlyingCategory( CapCategory( bicomplex ) ) ) );
    new_complex := AsComplex( z_functor_object );
    morphism_function := function ( i )
          local  source_summands, range_summands, nr_range_summands
           , nr_source_summands, source_projections,
          range_injections, horizontal_morphisms, vertical_morphisms
           , end_horizontal, end_vertical;
          i := - i;
          nr_source_summands := length - i + 1;
          nr_range_summands := length - i + 1 + 1;
          source_summands
           := List( [ 0 .. length - i ], function ( j )
                  return bicomplex[j + i][- j];
              end );
          range_summands := List( [ 0 .. length - i + 1 ],
             function ( j )
                  return bicomplex[j + i - 1][- j];
              end );
          source_projections := List( [ 1 .. nr_source_summands ],
             function ( j )
                  return
                   ProjectionInFactorOfDirectSum( source_summands,
                     j );
              end );
          range_injections := List( [ 1 .. nr_range_summands ],
          function ( j )
               return InjectionOfCofactorOfDirectSum(
                  range_summands, j );
           end );
      horizontal_morphisms
       := List( [ 0 .. length - i ], function ( j )
               return Differential( bicomplex, j + i )[- j];
           end );
      vertical_morphisms
       := List( [ 0 .. length - i ], function ( j )
               return Differential( bicomplex[j + i], - j );
           end );
      horizontal_morphisms
       := List( [ 1 .. Length( horizontal_morphisms ) ],
          function ( j )
               return PreCompose( source_projections[j],
                  horizontal_morphisms[j] );
           end );
      vertical_morphisms
       := List( [ 1 .. Length( vertical_morphisms ) ],
          function ( j )
               return PreCompose( source_projections[j],
                  vertical_morphisms[j] );
           end );
      horizontal_morphisms
       := List( [ 1 .. Length( horizontal_morphisms ) ],
          function ( j )
               return PreCompose( horizontal_morphisms[j],
                  range_injections[j] );
           end );
      vertical_morphisms
       := List( [ 1 .. Length( vertical_morphisms ) ],
          function ( j )
               return PreCompose( vertical_morphisms[j],
                  range_injections[j + 1] );
           end );
      end_horizontal := Sum( horizontal_morphisms );
      end_vertical := Sum( vertical_morphisms );
      return end_horizontal + end_vertical;
  end;
object_function := function ( i )
      i := - i;
```

```
          return
           DirectSum( List( [ 0 .. length - i ], function ( j )
                    return bicomplex[j + i][- j];
                end ) );
      end;
    z_functor_object!.differential_func := morphism_function;
    z_functor_object!.object_func := object_function;
    return new_complex;
end;
```

## 17. Function EmbeddingInObjectOfTotalComplex

```
function ( bicomplex, length, position, embedding_number )
    local  object_list;
    object_list := List( [ 0 .. length - position ], function ( j )
            return bicomplex[j + position][- j];
        end );
    return InjectionOfCofactorOfDirectSum( object_list,
       embedding_number );
end;
```

## 18. Function ConnectingMorphismFromCocomplexToCartanEilenbergResolution

```
function ( cocomplex, position, projective_resolution_function )
    local  delta_im1, delta_i, first_morphism, second_morphism,
    first_complex, second_complex, third_morphism, third_complex,
    fourth_morphism, eps, eps2;
    delta_im1 := Differential( cocomplex, position - 1 );
    delta_i := Differential( cocomplex, position );
    first_morphism
     := KernelLift( delta_i, ImageEmbedding( delta_im1 ) );
    second_morphism := CokernelProjection( first_morphism );
    first_complex := projective_resolution_function(
       Source( first_morphism ) );
    second_complex := projective_resolution_function(
       Range( second_morphism ) );
    third_morphism := KernelEmbedding( delta_i );
    fourth_morphism := CoastrictionToImage( delta_i );
    eps := Lift( second_complex[2], second_morphism );
    eps := UniversalMorphismFromDirectSum(
       [ PreCompose( first_complex[2], first_morphism ), eps ] );
    third_complex := projective_resolution_function(
      Range( fourth_morphism ) );
    eps2 := Lift( third_complex[2], fourth_morphism );
    eps2 := UniversalMorphismFromDirectSum(
      [ PreCompose( eps, third_morphism ), eps2 ] );
    return eps2;
end;
```

## 19. Function GeneralizedEmbeddingOfHomology

```
function ( complex, i )
    local  differential_i, differential_ip1, image_embedding,
    kernel_lift, map_to_homology, kernel_emb;
    differential_i := Differential( complex, i );
    differential_ip1 := Differential( complex, i + 1 );
    image_embedding := ImageEmbedding( differential_ip1 );
    kernel_lift := KernelLift( differential_i, image_embedding );
    map_to_homology := CokernelProjection( kernel_lift );
    kernel_emb := KernelEmbedding( differential_i );
    return GeneralizedMorphismWithSourceAid( map_to_homology,
      kernel_emb );
end;
```

## 20. Function GeneralizedMorphismBetweenHomologies

```
function ( source_complex, range_complex, connecting_morphism, i )
    local  source_embedding, range_embedding, generalized_connection;
    source_embedding := GeneralizedEmbeddingOfHomology(
      source_complex, i );
    range_embedding := GeneralizedEmbeddingOfHomology(
      range_complex, i );
    generalized_connection
     := AsGeneralizedMorphism( connecting_morphism );
    return
     PreCompose(
      PreCompose( source_embedding, generalized_connection ),
      PseudoInverse( range_embedding ) );
end;
```

## 21. Function GeneralizedEmbeddingOfSpectralSequenceEntry

```
function ( trhomCE, diag_number, page, homCE, homres,
    connection_mor )
    local  homhomres, resolution_len, tot, connection_at_0,
    connection_at_1, homcon_at_0, homcon_at_1, emb0, emb1,
    homcon_at_0_in_tot, homcon_at_1_in_tot, homology_iso,
    M_as_homology, M_to_M_as_homology, M_to_hom_of_tot, entry,
    homology_proj_of_tot, emb01;
    homhomres := DualOnCocomplex( homres );
    resolution_len := ResolutionLength( homhomres );
    tot := TotalComplexOfBicomplex( homCE, resolution_len );
    connection_at_0
     := ConnectingMorphismFromCocomplexToCartanEilenbergResolution(
       homres, 0, FreeResolutionCocomplex );
    connection_at_1
     := ConnectingMorphismFromCocomplexToCartanEilenbergResolution(
       homres, 1, FreeResolutionCocomplex );
    homcon_at_0 := DualOnMorphisms( connection_at_0 );
    homcon_at_1 := DualOnMorphisms( connection_at_1 );
    emb0 := EmbeddingInObjectOfTotalComplex( homCE, resolution_len,
       0, 1 );
    emb1 := EmbeddingInObjectOfTotalComplex( homCE, resolution_len,
       1, 1 );
    homcon_at_0_in_tot := PreCompose( homcon_at_0, emb0 );
    homcon_at_1_in_tot := PreCompose( homcon_at_1, emb1 );
    homology_iso := GeneralizedMorphismBetweenHomologies(
       homhomres, tot, homcon_at_0_in_tot, 0 );
    homology_iso := HonestRepresentative( homology_iso );
    M_as_homology
     := HonestRepresentative(
       PseudoInverse( GeneralizedEmbeddingOfHomology( homhomres, 0
           ) ) );
    M_to_M_as_homology := ColiftAlongEpimorphism( connection_mor,
       M_as_homology );
    M_to_hom_of_tot := PreCompose( M_to_M_as_homology, homology_iso
       );
    entry := SpectralSequenceEntry( trhomCE, page, - diag_number,
       diag_number );
    homology_proj_of_tot
     := PseudoInverse( GeneralizedEmbeddingOfHomology( tot, 0 ) );
    emb01 := EmbeddingInObjectOfTotalComplex( homCE,
       resolution_len, 0, diag_number + 1 );
    return
     PreCompose(
       PreCompose(
         PreCompose( entry, AsGeneralizedMorphism( emb01 ) ),
         homology_proj_of_tot ),
       AsGeneralizedMorphism( Inverse( M_to_hom_of_tot ) ) );
end;
```

## 22. Function PurityFiltrationBySpectralSequence

```
function ( trhomCE, page, homCE, homres, connection_mor )
    local  homhomres, resolution_len, embedding_list,
    combined_image_embeddings, pi_list, functors, i, mp, nu,
    mp_mat, eta_0, iota_i, eta, kappa, rho, iso, iso_inv;
    homhomres := DualOnCocomplex( homres );
    resolution_len := ResolutionLength( homhomres );
    embedding_list := List( [ 0 .. resolution_len ], function ( i )
            return GeneralizedEmbeddingOfSpectralSequenceEntry(
               trhomCE, i, page, homCE, homres, connection_mor );
        end );
    for i  in Reversed( [ 1 .. Length( embedding_list ) ] )  do
        if
         IsZero(
             UnderlyingHonestObject( Source( embedding_list[i] ) ) )
           then
            Remove( embedding_list, i );
        fi;
    od;
    embedding_list := Reversed( embedding_list );
    combined_image_embeddings
     := List( embedding_list, CombinedImageEmbedding );
    functors := ValueOption( "Functors" );
    if functors <> fail  then
        for i  in functors  do
            combined_image_embeddings
             := List( combined_image_embeddings, function ( j )
                   return
                    PreCompose(
                      Inverse(
                        ApplyNaturalTransformation( i, Source( j )
                          ) ), j );
                end );
        od;
    fi;
    pi_list := List( [ 2 .. Length( embedding_list ) ],
       function ( i )
            return
```

```
             PreCompose(
               AsGeneralizedMorphism( combined_image_embeddings[i] )
                , PseudoInverse( embedding_list[i] ) );
        end );
    pi_list := List( pi_list, HonestRepresentative );
    if functors <> fail  then
        for i  in functors  do
            pi_list := List( pi_list, function ( j )
                    return
                     PreCompose( j,
                       ApplyNaturalTransformation( i, Range( j ) ) );
                end );
        od;
    fi;
    for i  in [ 2 .. Length( combined_image_embeddings ) ]  do
        nu := CoverByProjectiveWithLift( pi_list[i - 1] );
        eta_0 := nu[2];
        nu := nu[1];
        mp_mat := KernelEmbedding( nu );
        iota_i
         := LiftAlongMonomorphism( combined_image_embeddings[i],
           combined_image_embeddings[i - 1] );
        eta := Lift( PreCompose( mp_mat, eta_0 ), iota_i );
        kappa := UniversalMorphismIntoDirectSum( mp_mat, eta );
        rho := UniversalMorphismFromDirectSum( - eta_0, iota_i );
        iso := CokernelColift( kappa, rho );
        combined_image_embeddings[i]
         := PreCompose( iso, combined_image_embeddings[i] );
    od;
    return
     combined_image_embeddings[Length( combined_image_embeddings )];
end;
```

# Index

(locally small) category, 15
(locally small) category (with Hom-setoids), 16
Cartan-Eilenberg resolution, 135

abelian, 30
additive inverse, 22
arrow, 51, 55, 56
arrow of cospan, 50
arrow of span, 54
augmentation map, 130

bidualizing resolution, 141
bidualizing spectral sequence, 144
biproducts, 25
bounded, 143

cache, 166
categorical programming, 159
category of generalized morphisms by cospans, 51
category of generalized morphisms by spans, 54
category with computable composition, 19
category with computable identity morphism, 19
chain complex, 129
chain map, 130
character, 99
character function, 106
character lattice, 99
class group, 102
classical category, 16
coastriction, 32
cochain complex, 129
codimension, 139
codominates, 31
cohomological projective resolution, 130
cokernel, 27
cokernel colift, 27
combined image, 146
combined image embedding, 146
common coarsening, 91, 94
common restriction, 81, 94
compatible locally free lift, 153
computable, 13
computable abelian, 30
computable additive, 25
computable category, 19
computable cokernels, 28
computable coproducts, 24
computable direct sums, 25
computable enough projectives, 127
computable fiber products, 48
computable function, 15
computable isomorphisms between products and coproducts, 25
computable kernels, 27
computable preabelian, 28
computable preadditive, 22
computable products, 24
computable pushouts, 49
computable ring, 37
computable zero objects, 24
cone, 100
congruence of morphisms, 16
connecting chain map, 135
converges, 143
conversion functor from 3-arrows to cospans, 65
conversion functor from 3-arrows to spans, 66
conversion functor from cospans to 3-arrows, 64
conversion functor from cospans to spans, 63
conversion functor from spans to 3-arrows, 64
conversion functor from spans to cospans, 63
coproduct injections, 23
coproduct object, 23
coproduct universal morphism, 23
corresponding honest 3-arrow, 63
corresponding honest cospans, 53
corresponding honest span, 56

cospan, 50
Cox ring, 102
crisp caches, 166

decidable epimorphisms, 21
decidable equivalence relation, 14
decidable isomorphisms, 21
decidable monomorphisms, 21
decidable property, 20
decidable set, 13
decidable zero object, 26
degree group, 33
degree of morphism, 34
direct sums, 25
dominates, 31
dual cone, 100
dual morphism, 140
dual object, 140
dual of bicomplex, 141
dual of complex, 141

enough injectives, 127
enough projectives, 127
enriched over a commutative regular semigroup, 67
epimorphism, 21
equivalent categories, 17
exact complex, 129, 130
Ext-Functor, 138

face, 100
factor object, 31
fan, 100
fiber product, 47
filtered, 132
filtered morphism, 132
filtered presentation, 145
free module, 128
functionally defined chain complex, 162
functor, 16

Gabriel morphism, 77
generalized morphism by 3-arrows, 56
generator degrees, 35
grade, 139
grade filtration, 140
graded module, 34
graded morphism, 34
graded part, 132
graded ring, 33
grading function, 106
group of torus invariant Weil divisors, 102

Hom-Functor, 138
homogeneous, 33
homological bicomplex, 134
homological injective resolution, 130
homological projective resolution, 130
homological spectral sequence, 142
homology, 129
honest, 53, 56, 62
honest representative, 53, 56, 62

identity morphism, 15
image, 31
injective, 127
internal Ext-Functor, 139
internal Hom, 139
internal Hom-Functor, 138
invalid weak pointer, 166
irrelevant ideal, 103
isomorphisms, 21

kernel, 27
kernel lift, 27

lattice of one-parametric subgroups, 100
locally free, 152

method derivation graph, 172
monomial cone, 107
monomorphism, 21
morphism of graded modules, 34

normalized, 57
normalized cospan, 50
normalized span, 54
number of generators, 35

one-parametric subgroup, 99

page, 142
pointed, 100
pointer, 166
preabelian, 27
preadditive, 22
product object, 23
product projections, 23
product universal morphism, 23
projective, 127
pseudo-inverse, 70
pullback, 47
purity filtration, 140

pushout, 48

quotient object, 31

range aid, 56
rational polyhedral cone, 100
ray, 100
ray generator, 100
ray generators, 100
realization, 14
realization map, 14
realization of category, 18
realization of field, 38
realization of graded ring, 37
realization of group, 37
realization of ring, 36
realized category, 18
reduction of category, 17
relation matrix, 35
restriction, 32
restriction-coastriction, 32
reversed arrow, 51, 55
reversed arrow of cospan, 50
reversed arrow of span, 54
right derived functor, 138
row dominates, 34

saturated affine semigroup, 107
Serre morphism category, 79
Serre quotient category, 76
setoid interpretation, 16
source aid, 56
span, 53
spectral sequence of bicomplex, 144
subcategory of Gabriel morphisms, 79
subfactor embedding, 132
subfactor projection, 132
subobject, 31
sum of two morphisms, 22

thick subcategory, 76
ToDoList, 175
ToDoList entry, 174
toric variety, 99
toric variety of fan, 101
torus invariant Weil divisor., 102
total complex, 134
transposed bicomplex, 135

undecidable, 19

valid weak pointer, 166

weak caches, 166
weak pointers, 166

zero morphism, 22
zero object, 23, 26
zeroid, 79