CAP

Categories, Algorithms, Programming

2026.05-08

26 May 2026

Sebastian Gutsche
Sebastian Posur
Øystein Skartsæterhagen
Fabian Zickgraf

Sebastian Gutsche Email: sebastian.gutsche@gmail.com
Homepage: https://sebasguts.github.io/
Address: Department Mathematik
Universität Siegen
Walter-Flex-Straße 3
57068 Siegen
Germany

Sebastian Posur Email: sebastian.posur@uni-muenster.de
Homepage: https://sebastianpos.github.io
Address: Department Mathematik
Universität Siegen
Walter-Flex-Straße 3
57068 Siegen
Germany

Øystein Skartsæterhagen Email: oysteini@math.ntnu.no
Homepage: http://www.math.ntnu.no/~oysteini/
Address: NTNU
Institutt for matematiske fag
7491 Trondheim
Norway

Fabian Zickgraf Email: f.zickgraf@dashdos.com
Homepage: https://github.com/zickgraf/
Address: Walter-Flex-Str. 3
57068 Siegen
Germany

Contents

Chapter 1

CAP Categories

Categories are the main GAP objects in CAP. They are used to associate GAP objects which represent objects and morphisms with their category. By associating a GAP object to the category, one of two filters belonging to the category (ObjectFilter/MorphismFilter) are set to true. Via Add methods, functions for specific existential quantifiers can be associated to the category and after that can be applied to GAP objects in the category. A GAP category object also knows which constructions are currently possible in this category.

Classically, a category consists of a class of objects, a set of morphisms, identity morphisms, and a composition function satisfying some simple axioms. In CAP, we use a slightly different notion of a category.

A CAP category $\mathbf{C}$ consists of the following data:

- A set $\mathrm{Obj}_{\mathbf{C}}$ of *objects*.
- For every pair $a,b \in \mathrm{Obj}_{\mathbf{C}}$, a set $\mathrm{Hom}_{\mathbf{C}}(a,b)$ of *morphisms*.
- For every pair $a,b \in \mathrm{Obj}_{\mathbf{C}}$, an equivalence relation $\sim_{a,b}$ on $\mathrm{Hom}_{\mathbf{C}}(a,b)$ called *congruence for morphisms*.
- For every $a \in \mathrm{Obj}_{\mathbf{C}}$, an *identity morphism* $\mathrm{id}_a \in \mathrm{Hom}_{\mathbf{C}}(a,a)$.
- For every triple $a,b,c \in \mathrm{Obj}_{\mathbf{C}}$, a *composition function*

  $$\circ : \mathrm{Hom}_{\mathbf{C}}(b,c) \times \mathrm{Hom}_{\mathbf{C}}(a,b) \to \mathrm{Hom}_{\mathbf{C}}(a,c)$$

  compatible with the congruence, i.e., if $\alpha,\alpha' \in \mathrm{Hom}_{\mathbf{C}}(a,b)$, $\beta,\beta' \in \mathrm{Hom}_{\mathbf{C}}(b,c)$, $\alpha \sim_{a,b} \alpha'$ and $\beta \sim_{b,c} \beta'$, then $\beta \circ \alpha \sim_{a,c} \beta' \circ \alpha'$.
- For all $a,b \in \mathrm{Obj}_{\mathbf{C}}$, $\alpha \in \mathrm{Hom}_{\mathbf{C}}(a,b)$, we have

  $$(\mathrm{id}_b \circ \alpha) \sim_{a,b} \alpha$$

  and

  $$\alpha \sim_{a,b} (\alpha \circ \mathrm{id}_a).$$
- For all $a,b,c,d \in \mathrm{Obj}_{\mathbf{C}}$, $\alpha \in \mathrm{Hom}_{\mathbf{C}}(a,b)$, $\beta \in \mathrm{Hom}_{\mathbf{C}}(b,c)$, $\gamma \in \mathrm{Hom}_{\mathbf{C}}(c,d)$, we have

  $$((\gamma \circ \beta) \circ \alpha) \sim_{a,d} (\gamma \circ (\beta \circ \alpha))$$

1.1 Categories

1.1.1 IsCapCategory (for IsAttributeStoringRep)

▷ `IsCapCategory(object)` (Category)

Returns: `true` or `false`

The GAP category of CAP categories. Objects of this type handle the CAP category information, the caching, and filters for objects in the CAP category. Please note that the object itself is not related to methods, you only need it as a handler and a presentation of the CAP category.

1.1.2 IsCapCategoryCell (for IsAttributeStoringRep)

▷ `IsCapCategoryCell(object)` (Category)

Returns: `true` or `false`

The GAP category of CAP category cells. Every object, morphism, and 2-cell of a CAP category lies in this GAP category.

1.1.3 IsCapCategoryObject (for IsCapCategoryCell)

▷ `IsCapCategoryObject(object)` (Category)

Returns: `true` or `false`

The GAP category of CAP category objects. Every object of a CAP category lies in this GAP category.

1.1.4 IsCapCategoryMorphism (for IsCapCategoryCell)

▷ `IsCapCategoryMorphism(object)` (Category)

Returns: `true` or `false`

The GAP category of CAP category morphisms. Every morphism of a CAP category lies in this GAP category.

1.1.5 IsCapCategoryTwoCell (for IsCapCategoryCell)

▷ `IsCapCategoryTwoCell(object)` (Category)

Returns: `true` or `false`

The GAP category of CAP category 2-cells. Every 2-cell of a CAP category lies in this GAP category.

1.2 Categorical properties

1.2.1 AddCategoricalProperty

▷ `AddCategoricalProperty(list)` (function)

Adds a categorical property to the list of CAP categorical properties. `list` must be a list containing one entry, if the property is self dual, or two, if the dual property has a different name. If the first entry of the list is empty and the second is a property name, the property is assumed to have no dual.

1.2.2 IsObjectFiniteCategory (for IsCapCategory)

▷ `IsObjectFiniteCategory(C)` (property)

Returns: `true` or `false`

The (evil) property of *C* being a category with finitely many objects.

1.2.3 IsFiniteCategory (for IsCapCategory)

▷ `IsFiniteCategory(C)` (property)

Returns: `true` or `false`

The (evil) property of *C* being a finite category.

1.2.4 IsFinite (for IsCapCategory)

▷ `IsFinite(C)` (property)

Returns: `true` or `false`

Synonym for `IsFiniteCategory`.

1.2.5 IsEquivalentToFiniteCategory (for IsCapCategory)

▷ `IsEquivalentToFiniteCategory(C)` (property)

Returns: `true` or `false`

The property of *C* being equivalent to a finite category.

1.2.6 IsFinitelyPresentedCategory (for IsCapCategory)

▷ `IsFinitelyPresentedCategory(C)` (property)

Returns: `true` or `false`

The (evil) property of *C* being a finitely presented.

1.2.7 IsEquippedWithHomomorphismStructure (for IsCapCategory)

▷ `IsEquippedWithHomomorphismStructure(C)` (property)

Returns: `true` or `false`

The property of the category *C* being equipped with a homomorphism structure.

1.2.8 IsCategoryWithDecidableLifts (for IsCapCategory)

▷ `IsCategoryWithDecidableLifts(C)` (property)

Returns: `true` or `false`

The property of the category *C* having decidable lifts.

1.2.9 IsCategoryWithDecidableColifts (for IsCapCategory)

▷ `IsCategoryWithDecidableColifts(C)` (property)

Returns: `true` or `false`

The property of the category *C* having decidable colifts.

1.2.10 IsCategoryWithInitialObject (for IsCapCategory)

▷ `IsCategoryWithInitialObject(C)` (property)

Returns: `true` or `false`

The property of the category C having an initial object.

1.2.11 IsCategoryWithTerminalObject (for IsCapCategory)

▷ `IsCategoryWithTerminalObject(C)` (property)

Returns: `true` or `false`

The property of the category C having a terminal object.

1.2.12 IsCategoryWithZeroObject (for IsCapCategory)

▷ `IsCategoryWithZeroObject(C)` (property)

Returns: `true` or `false`

The property of the category C having a zero object.

1.2.13 IsCategoryWithEqualizers (for IsCapCategory)

▷ `IsCategoryWithEqualizers(C)` (property)

Returns: `true` or `false`

The property of the category C having all equalizers.

1.2.14 IsCategoryWithCoequalizers (for IsCapCategory)

▷ `IsCategoryWithCoequalizers(C)` (property)

Returns: `true` or `false`

The property of the category C having all coequalizers.

1.2.15 IsCategoryWithCoequalizersOfIdentityAndAutomorphisms (for IsCapCategory)

▷ `IsCategoryWithCoequalizersOfIdentityAndAutomorphisms(C)` (property)

Returns: `true` or `false`

The property of the category C having all coequalizers of an identity and automorphisms.

1.2.16 IsCategoryWithEqualizersOfIdentityAndAutomorphisms (for IsCapCategory)

▷ `IsCategoryWithEqualizersOfIdentityAndAutomorphisms(C)` (property)

Returns: `true` or `false`

The property of the category C having all equalizers of an identity and automorphisms.

1.2.17 IsEnrichedOverCommutativeRegularSemigroup (for IsCapCategory)

▷ `IsEnrichedOverCommutativeRegularSemigroup(C)` (property)

Returns: `true` or `false`

The property of the category C being enriched over a commutative regular semigroup.

1.2.18 IsSkeletalCategory (for IsCapCategory)

▷ `IsSkeletalCategory(C)` (property)

Returns: `true` or `false`

The property of the category C being skeletal, that is, whether `IsEqualForObjects` and `IsIsomorphicForObjects` coincide.

1.2.19 IsAbCategory (for IsCapCategory)

▷ `IsAbCategory(C)` (property)

Returns: `true` or `false`

The property of the category C being preadditive.

1.2.20 IsLinearCategoryOverCommutativeSemiring (for IsCapCategory)

▷ `IsLinearCategoryOverCommutativeSemiring(C)` (property)

Returns: `true` or `false`

The property of the category C being linear over a commutative semiring.

1.2.21 IsLinearCategoryOverCommutativeSemiringWithFinitelyGeneratedFreeExternalHoms (for IsCapCategory)

▷ `IsLinearCategoryOverCommutativeSemiringWithFinitelyGeneratedFreeExternalHoms(C)` (property)

Returns: `true` or `false`

The property of the category C being linear over a commutative semiring k such that all external homs are finitely generated free k-modules.

1.2.22 IsLinearCategoryOverCommutativeRing (for IsCapCategory)

▷ `IsLinearCategoryOverCommutativeRing(C)` (property)

Returns: `true` or `false`

The property of the category C being linear over a commutative ring.

1.2.23 IsLinearCategoryOverCommutativeRingWithFinitelyGeneratedFreeExternalHoms (for IsCapCategory)

▷ `IsLinearCategoryOverCommutativeRingWithFinitelyGeneratedFreeExternalHoms(C)` (property)

Returns: `true` or `false`

The property of the category C being linear over a commutative ring k such that all external homs are finitely generated free k-modules.

1.2.24 IsCategoryWithKernels (for IsCapCategory)

▷ `IsCategoryWithKernels(C)` (property)

Returns: `true` or `false`

The property of the category C having all kernels.

1.2.25 IsCategoryWithCokernels (for IsCapCategory)

▷ `IsCategoryWithCokernels(C)` (property)

Returns: `true` or `false`

The property of the category *C* having all cokernels.

1.2.26 IsAdditiveCategory (for IsCapCategory)

▷ `IsAdditiveCategory(C)` (property)

Returns: `true` or `false`

The property of the category *C* being additive.

1.2.27 IsPreAbelianCategory (for IsCapCategory)

▷ `IsPreAbelianCategory(C)` (property)

Returns: `true` or `false`

The property of the category *C* being preabelian.

1.2.28 IsAbelianCategory (for IsCapCategory)

▷ `IsAbelianCategory(C)` (property)

Returns: `true` or `false`

The property of the category *C* being abelian.

1.2.29 IsAbelianCategoryWithEnoughProjectives (for IsCapCategory)

▷ `IsAbelianCategoryWithEnoughProjectives(C)` (property)

Returns: `true` or `false`

The property of the category *C* being abelian with enough projectives.

1.2.30 IsAbelianCategoryWithEnoughInjectives (for IsCapCategory)

▷ `IsAbelianCategoryWithEnoughInjectives(C)` (property)

Returns: `true` or `false`

The property of the category *C* being abelian with enough injectives.

1.2.31 IsLocallyOfFiniteProjectiveDimension (for IsCapCategory)

▷ `IsLocallyOfFiniteProjectiveDimension(C)` (property)

Returns: `true` or `false`

The property of the category *C* being locally of finite projective dimension.

1.2.32 IsLocallyOfFiniteInjectiveDimension (for IsCapCategory)

▷ `IsLocallyOfFiniteInjectiveDimension(C)` (property)

Returns: `true` or `false`

The property of the category *C* being locally of finite injective dimension.

1.2.33 ListOfDefiningOperations

▷ `ListOfDefiningOperations(categorical_property)` (function)

Returns: a list

The input *categorical_property* is the name of a valid categorical property. The output is the list of names of the CAP operations needed to make this categorical property constructive.

1.3 Constructor

1.3.1 CreateCapCategory

▷ `CreateCapCategory()` (operation)

Returns: a category

Creates a new CAP category from scratch. It gets a generic name.

1.3.2 CreateCapCategory (for IsString)

▷ `CreateCapCategory(s)` (operation)

Returns: a category

The argument is a string s. This operation creates a new CAP category from scratch. Its name is set to s.

1.3.3 CreateCapCategory (for IsString, IsFunction, IsFunction, IsFunction, IsFunction)

▷ `CreateCapCategory(s, category_filter, object_filter, morphism_filter, two_cell_filter)` (operation)

Returns: a category

The argument is a string s. This operation creates a new CAP category from scratch. Its name is set to s. The category, its objects, its morphisms, and its two cells will lie in the corresponding given filters.

1.3.4 CreateCapCategoryWithDataTypes

▷ `CreateCapCategoryWithDataTypes(s, category_filter, object_filter, morphism_filter, two_cell_filter, object_datum_type, morphism_datum_type, two_cell_datum_type)` (function)

Returns: a category

The argument is a string s. This operation creates a new CAP category from scratch. Its name is set to s. The category, its objects, its morphisms, and its two cells will lie in the corresponding given filters. The data types of the object/morphism/two cell datum can be given as described in `CapJitInferredDataTypes` (**CompilerForCAP: CapJitInferredDataTypes**). As a convenience, simply a filter can be given if this suffices to fully determine the data type. If a data type is not specified, pass `fail` instead.

1.4 Internal Attributes

1.4.1 Name (for IsCapCategory)

▷ `Name(C)` (attribute)

Returns: a string

The argument is a category C. The output is its name.

Each category C stores various filters. They are used to apply the right functions in the method selection.

1.4.2 CategoryFilter (for IsCapCategory)

▷ `CategoryFilter(C)` (attribute)

Returns: a filter

The argument is a category C. The output is a filter in which C lies.

1.4.3 ObjectFilter (for IsCapCategory)

▷ `ObjectFilter(C)` (attribute)

Returns: a filter

The argument is a category C. The output is a filter in which all objects of C shall lie.

1.4.4 MorphismFilter (for IsCapCategory)

▷ `MorphismFilter(C)` (attribute)

Returns: a filter

The argument is a category C. The output is a filter in which all morphisms of C shall lie.

1.4.5 TwoCellFilter (for IsCapCategory)

▷ `TwoCellFilter(C)` (attribute)

Returns: a filter

The argument is a category C. The output is a filter in which all 2-cells of C shall lie.

1.4.6 ObjectDatumType (for IsCapCategory)

▷ `ObjectDatumType(C)` (attribute)

Returns: a data type or `fail`

The argument is a category C. The output is the data type (see `CapJitInferredDataTypes` (**CompilerForCAP: CapJitInferredDataTypes**)) of object data of C (or `fail` if this data type is not specified).

1.4.7 MorphismDatumType (for IsCapCategory)

▷ `MorphismDatumType(C)` (attribute)

Returns: a data type or `fail`

The argument is a category C. The output is the data type (see `CapJitInferredDataTypes` (**CompilerForCAP: CapJitInferredDataTypes**)) of morphism data of C (or `fail` if this data type is not specified).

1.4.8 TwoCellDatumType (for IsCapCategory)

▷ `TwoCellDatumType(C)` (attribute)

Returns: a data type or `fail`

The argument is a category C. The output is the data type (see `CapJitInferredDataTypes` (**CompilerForCAP: CapJitInferredDataTypes**)) of two cell data of C (or `fail` if this data type is not specified).

1.4.9 CommutativeSemiringOfLinearCategory (for IsCapCategory)

▷ `CommutativeSemiringOfLinearCategory(C)` (attribute)

Returns: a ring

The argument is a category C which is expected to lie in the filter `IsLinearCategoryOverCommutativeRing`. The output is a commutative ring over which the category is linear.

1.4.10 RangeCategoryOfHomomorphismStructure (for IsCapCategory)

▷ `RangeCategoryOfHomomorphismStructure(C)` (attribute)

Returns: a category

The argument is a category C which is expected to lie in the filter `IsEquippedWithHomomorphismStructure`. The output is the range category D of the defining functor $H : C^{\mathrm{op}} \times C \to D$ of the homomorphism structure.

1.4.11 AdditiveGenerators (for IsCapCategory)

▷ `AdditiveGenerators(C)` (attribute)

Returns: a list of objects

The argument is an additive category C. The output is a list L of objects in C such that every object in C is a finite direct sum of objects in L.

1.4.12 IndecomposableProjectiveObjects (for IsCapCategory)

▷ `IndecomposableProjectiveObjects(C)` (attribute)

Returns: a list of objects

The argument is an Abelian category C with enough projectives. The output is the set of indecomposable projective objects in C up to isomorphism. That is every projective object in C is isomorphic to a finite direct sum over these objects.

1.4.13 IndecomposableInjectiveObjects (for IsCapCategory)

▷ `IndecomposableInjectiveObjects(C)` (attribute)

Returns: a list of objects

The argument is an Abelian category C with enough injectives. The output is the set of indecomposable injective objects in C up to isomorphism. That is every injective object in C is isomorphic to a finite direct sum over these objects.

1.5 Logic switcher

1.5.1 CapCategorySwitchLogicPropagationForObjectsOn

▷ `CapCategorySwitchLogicPropagationForObjectsOn(C)` (function)

Activates the predicate logic propagation between equal objects for the category *C*.

1.5.2 CapCategorySwitchLogicPropagationForObjectsOff

▷ `CapCategorySwitchLogicPropagationForObjectsOff(C)` (function)

Deactivates the predicate logic propagation between equal objects for the category *C*.

1.5.3 CapCategorySwitchLogicPropagationForMorphismsOn

▷ `CapCategorySwitchLogicPropagationForMorphismsOn(C)` (function)

Activates the predicate logic propagation between equal morphisms for the category *C*.

1.5.4 CapCategorySwitchLogicPropagationForMorphismsOff

▷ `CapCategorySwitchLogicPropagationForMorphismsOff(C)` (function)

Deactivates the predicate logic propagation between equal morphisms for the category *C*.

1.5.5 CapCategorySwitchLogicPropagationOn

▷ `CapCategorySwitchLogicPropagationOn(C)` (function)

Activates the predicate logic propagation between equal cells for the category *C*.

1.5.6 CapCategorySwitchLogicPropagationOff

▷ `CapCategorySwitchLogicPropagationOff(C)` (function)

Deactivates the predicate logic propagation between equal cells for the category *C*.

1.5.7 CapCategorySwitchLogicOn

▷ `CapCategorySwitchLogicOn(C)` (function)

Activates the predicate implication logic for the category *C*.

1.5.8 CapCategorySwitchLogicOff

▷ `CapCategorySwitchLogicOff(C)` (function)

Deactivates the predicate implication logic for the category *C*.

1.6 Tool functions

1.6.1 CanCompute (for IsCapCategory, IsString)

▷ `CanCompute(C, string)` (operation)
▷ `CanCompute(C, operation)` (operation)

Returns: `true` or `false`

The argument is a category `C` and a string `string`, which should be the name of a CAP operation, e.g., PreCompose. If applying this method is possible in C, the method returns `true`, `false` otherwise. If the string is not the name of a CAP operation, an error is raised. For debugging purposes one can also pass the CAP operation instead of its name.

1.6.2 OperationWeight (for IsCapCategory, IsString)

▷ `OperationWeight(cat, op_name)` (operation)

Returns: an integer

Returns the weight of the operation currently installed as `op_name` in `cat`.

1.6.3 MissingOperationsForConstructivenessOfCategory (for IsCapCategory, IsStringRep)

▷ `MissingOperationsForConstructivenessOfCategory(C, s)` (operation)

Returns: a list

The arguments are a category C and a string s. If s is a categorical property (e.g. `"IsAbelianCategory"`), the output is a list of strings with CAP operations which are missing in C to have the categorical property constructively. If s is not a categorical property, an error is raised.

1.7 Well-Definedness of Cells

1.7.1 IsWellDefined (for IsCapCategoryCell)

▷ `IsWellDefined(c)` (property)

Returns: a boolean

The argument is a cell c. The output is `true` if c is well-defined, otherwise the output is `false`.

1.8 Unpacking data structures

1.8.1 Down (for IsObject)

▷ `Down(x)` (attribute)

Returns: a GAP object

The argument is a GAP object x. If x is an object in a CAP category, the output consists of data which are needed to reconstruct x (e.g., by passing them to an appropriate constructor). If x is a morphism in a CAP category, the output consists of a triple whose first entry is the source of x, the third entry is the range of x, and the second entry consists of data which are needed to reconstruct x (e.g., by passing them to an appropriate constructor, possibly together with the source and range of x).

1.8.2 DownOnlyMorphismData (for IsCapCategoryMorphism)

▷ `DownOnlyMorphismData(x)` (attribute)

Returns: a GAP object

The argument is a morphism in a CAP category, the output consists of data which are needed to reconstruct x (e.g., by passing it to an appropriate constructor, possibly together with its source and range).

1.8.3 DownToBottom (for IsObject)

▷ `DownToBottom(x)` (attribute)

Returns: a GAP object

The argument is a GAP object x. This function iteratively calls `Down` until it becomes stable.

1.9 Caching

1.9.1 SetCachingOfCategory

▷ `SetCachingOfCategory(category, type)` (function)

Sets the caching of *category* to *type*.

1.9.2 SetCachingOfCategoryWeak

▷ `SetCachingOfCategoryWeak(category)` (function)
▷ `SetCachingOfCategoryCrisp(category)` (function)
▷ `DeactivateCachingOfCategory(category)` (function)

Sets the caching of *category* to `weak`, `crisp` or `none`, respectively.

1.9.3 SetDefaultCaching

▷ `SetDefaultCaching(type)` (function)
▷ `SetDefaultCachingWeak()` (function)
▷ `SetDefaultCachingCrisp()` (function)
▷ `DeactivateDefaultCaching()` (function)

Sets the default caching behaviour, all new categories will have their caching set to either `weak`, `crisp`, or `none`. The default at startup is `weak`.

1.10 Sanity checks

1.10.1 DisableInputSanityChecks

▷ `DisableInputSanityChecks(category)` (function)
▷ `DisableOutputSanityChecks(category)` (function)
▷ `EnablePartialInputSanityChecks(category)` (function)
▷ `EnablePartialOutputSanityChecks(category)` (function)

▷ `EnableFullInputSanityChecks(category)` (function)
▷ `EnableFullOutputSanityChecks(category)` (function)
▷ `DisableSanityChecks(category)` (function)
▷ `EnablePartialSanityChecks(category)` (function)
▷ `EnableFullSanityChecks(category)` (function)

Most operations can perform optional sanity checks on their arguments and results. The checks can either be partial (set by default), full, or disabled. With the following commands you can either enable the full checks, the partial checks or, for performance, disable the checks altogether. You can do this for input checks, output checks or for both at once.

1.11 Timing statistics

1.11.1 EnableTimingStatistics

▷ `EnableTimingStatistics(category)` (function)
▷ `DisableTimingStatistics(category)` (function)
▷ `ResetTimingStatistics(category)` (function)
▷ `DisplayTimingStatistics(category)` (function)
▷ `BrowseTimingStatistics(category)` (function)

Enable, disable, reset, display, or browse timing statistics of the primitive operations of `category`. Caution: If a primitive operation calls another primitive operation, the runtime of the later (including sanity checks etc.) is also included in the runtime of the former.

1.12 Enable automatic calls of `Add`

1.12.1 EnableAddForCategoricalOperations

▷ `EnableAddForCategoricalOperations(C)` (function)
▷ `DisableAddForCategoricalOperations(C)` (function)

Enables/disables the automatic call of `Add` for the output of primitively added functions for the category `C`. If the automatic call of `Add` is disabled (default), the output of primitively added functions must belong to the correct category. If the automatic call of `Add` is enabled, the output of primitively added functions only has to be a GAP object lying in `IsAttributeStoringRep` (with suitable attributes `Source` and `Range` if the output should be a morphism or a twocell).

1.13 Performance tweaks

For finding performance issues in primitive operations, you can collect timing statistics, see 1.11. You can use the package `CompilerForCAP` to compile your code. Additionally, CAP has several settings which can improve the performance. In the following some of these are listed.

- `DeactivateCachingOfCategory` or `DeactivateDefaultCaching`: see 1.9. This can either improve or degrade the performance depending on the concrete example.

- `CapCategorySwitchLogicOff` (on by default) or `CapCategorySwitchLogicPropagationOff` (off by default): see 1.5. This can either improve or degrade the performance depending on the concrete example.
- `DisableSanityChecks`: see 1.10.
- `DisableAddForCategoricalOperations`: see 1.12.
- `DeactivateToDoList`: see the package `ToolsForHomalg`.
- Use `CreateCapCategoryObjectWithAttributes` (2.2) instead of `AddObject` and `CreateCapCategoryMorphismWithAttributes` (3.2) instead of `AddMorphism`.
- Pass the option `overhead := false` to `CreateCapCategory`. Note: this may have unintended effects. Use with care!

1.14 LaTeX

1.14.1 LaTeXOutput (for IsCapCategoryCell)

▷ `LaTeXOutput(c)` (operation)

Returns: a string

The argument is a cell c. The output is a LaTeX string s (without enclosing dollar signs) that may be used to print out c nicely.

1.14.2 LaTeXOutput (for IsCapCategory)

▷ `LaTeXOutput(C)` (operation)

Returns: a string

The argument is a category C. The output is a LaTeX string s (without enclosing dollar signs) that may be used to print out C nicely.

Chapter 2

Objects

Any GAP object which is IsCapCategoryObject can be added to a category and then becomes an object in this category. Any object can belong to one or no category. After a GAP object is added to the category, it knows which things can be computed in its category and to which category it belongs. It knows categorial properties and attributes, and the functions for existential quantifiers can be applied to the object.

2.1 Attributes for the Type of Objects

2.1.1 CapCategory (for IsCapCategoryObject)

▷ `CapCategory(a)` (attribute)

Returns: a category

The argument is an object a. The output is the category **C** to which a was added.

2.2 Adding Objects to a Category

2.2.1 Add (for IsCapCategory, IsCapCategoryObject)

▷ `Add(category, object)` (operation)

Adds *object* as an object to *category*.

2.2.2 AddObject (for IsCapCategory, IsAttributeStoringRep)

▷ `AddObject(category, object)` (operation)

Adds *object* as an object to *category*. If *object* already lies in the filter `IsCapCategoryObject`, the operation `Add` (2.2.1) can be used instead.

2.2.3 CreateCapCategoryObjectWithAttributes

▷ `CreateCapCategoryObjectWithAttributes(category[, attribute1, value1, ...])` (function)

Returns: an object

Creates an object in *category* with the given attributes.

2.2.4 AsCapCategoryObject

▷ `AsCapCategoryObject(category, value)` (function)

Returns: an object

EXPERIMENTAL: This specification might change any time without prior notice. Views *value* as an object in *category*.

2.2.5 AsPrimitiveValue (for IsCapCategoryObject)

▷ `AsPrimitiveValue(object)` (attribute)
▷ `AsInteger(object)` (attribute)
▷ `AsHomalgMatrix(object)` (attribute)

Returns: a value

EXPERIMENTAL: This specification might change any time without prior notice. Views an object obtained via `AsCapCategoryObject` (2.2.4) as a primitive value again. Here, the word *primitive* means *primitive from the perspective of the category*. For example, from the perspective of an opposite category, objects of the underlying category are primitive values. The attribute is chosen according to the object datum type:

- For `IsInt`, the attribute `AsInteger` is used.
- For `IsHomalgMatrix`, the attribute `AsHomalgMatrix` is used.

In all other cases or if no object datum type is given, the attribute `AsPrimitiveValue` is used.

2.3 Equalities for Objects

2.3.1 IsEqualForObjects (for IsCapCategoryObject, IsCapCategoryObject)

▷ `IsEqualForObjects(a, b)` (operation)

Returns: a boolean

The arguments are two objects a and b. The output is `true` if $a = b$, otherwise the output is `false`.

2.3.2 IsIsomorphicForObjects (for IsCapCategoryObject, IsCapCategoryObject)

▷ `IsIsomorphicForObjects(a, b)` (operation)

Returns: a boolean

The arguments are two objects a and b. The output is `true` if a and b are isomorphic, that is, if there exists an isomorphism $a \to b$, otherwise the output is `false`.

2.3.3 SomeIsomorphismBetweenObjects (for IsCapCategoryObject, IsCapCategoryObject)

▷ `SomeIsomorphismBetweenObjects(a, b)` (operation)

Returns: an isomorphism in $\mathrm{Hom}(a,b)$

The arguments are two isomorphic objects a and b. The output is an isomorphism $a \to b$.

2.4 Categorical Properties of Objects

2.4.1 IsBijectiveObject (for IsCapCategoryObject)

▷ `IsBijectiveObject(a)` (property)

Returns: a boolean

The argument is an object a. The output is `true` if a is a bijective object, otherwise the output is `false`.

2.4.2 IsProjective (for IsCapCategoryObject)

▷ `IsProjective(a)` (property)

Returns: a boolean

The argument is an object a. The output is `true` if a is a projective object, otherwise the output is `false`.

2.4.3 IsInjective (for IsCapCategoryObject)

▷ `IsInjective(a)` (property)

Returns: a boolean

The argument is an object a. The output is `true` if a is an injective object, otherwise the output is `false`.

2.4.4 IsTerminal (for IsCapCategoryObject)

▷ `IsTerminal(a)` (property)

Returns: a boolean

The argument is an object a of a category $\mathbf{C}$. The output is `true` if a is isomorphic to the terminal object of $\mathbf{C}$, otherwise the output is `false`.

2.4.5 IsInitial (for IsCapCategoryObject)

▷ `IsInitial(a)` (property)

Returns: a boolean

The argument is an object a of a category $\mathbf{C}$. The output is `true` if a is isomorphic to the initial object of $\mathbf{C}$, otherwise the output is `false`.

2.4.6 IsZeroForObjects (for IsCapCategoryObject)

▷ `IsZeroForObjects(a)` (property)

Returns: a boolean

The argument is an object a of a category $\mathbf{C}$. The output is `true` if a is isomorphic to the zero object of $\mathbf{C}$, otherwise the output is `false`.

2.4.7 IsZero (for IsCapCategoryObject)

▷ `IsZero(a)` (property)

Returns: a boolean

The argument is an object a of a category **C**. The output is `true` if a is isomorphic to the zero object of **C**, otherwise the output is `false`.

2.5 Random Objects

CAP provides two principal methods to generate random objects:

- *By integers*: The integer is simply a parameter that can be used to create a random object.
- *By lists*: The list is used when creating a random object would need more than one parameter. Lists offer more flexibility at the expense of the genericity of the methods. This happens because lists that are valid as input in some category may be not valid for other categories. Hence, these operations are not thought to be used in generic categorical algorithms.

2.5.1 RandomObjectByInteger (for IsCapCategory, IsInt)

▷ `RandomObjectByInteger(C, n)` (operation)

Returns: an object in C

The arguments are a category C and an integer n. The output is a random object in C.

2.5.2 RandomObjectByList (for IsCapCategory, IsList)

▷ `RandomObjectByList(C, L)` (operation)

Returns: an object in C

The arguments are a category C and a list L. The output is a random object in C.

2.5.3 RandomObject (for IsCapCategory, IsInt)

▷ `RandomObject(C, n)` (operation)

These are convenient methods and they, depending on the input, delegate to one of the above methods.

2.5.4 RandomObject (for IsCapCategory, IsList)

▷ `RandomObject(C, L)` (operation)

2.6 Tool functions for caches

2.6.1 IsEqualForCacheForObjects (for IsCapCategoryObject, IsCapCategoryObject)

▷ `IsEqualForCacheForObjects(phi, psi)` (operation)

Returns: true or false

By default, CAP uses caches to store the values of Categorical operations. To get a value out of the cache, one needs to compare the input of a basic operation with its previous input. To compare objects in the category, IsEqualForCacheForObjects is used. By default, IsEqualForCacheForObjects falls back to IsEqualForCache (see ToolsForHomalg), which in turn defaults to recursive comparison for lists and `IsIdenticalObj` in all other cases. If you add a function via `AddIsEqualForCacheForObjects`, that function is used instead. A function $F : a,b \mapsto bool$ is expected there. The output has to be true or false. Fail is not allowed in this context.

2.7 Object constructors

2.7.1 ObjectConstructor (for IsCapCategory, IsObject)

▷ `ObjectConstructor(C, a)` (operation)

Returns: an object

The arguments are a category C and an object datum a (type and semantics of the object datum depend on the category). The output is an object of C defined by a. Note that by default this CAP operation is not cached. You can change this behaviour by calling `SetCachingToWeak( C, "ObjectConstructor" )` resp. `SetCachingToCrisp( C, "ObjectConstructor" )`.

2.7.2 / (for IsObject, IsCapCategory)

▷ `/(a, C)` (operation)

Returns: an object

Shorthand for `ObjectConstructor( C, a )`.

2.7.3 ObjectDatum (for IsCapCategoryObject)

▷ `ObjectDatum(obj)` (attribute)

Returns: depends on the category

The argument is a CAP category object *obj*. The output is a datum which can be used to construct *obj*, that is, `IsEqualForObjects( obj, ObjectConstructor( CapCategory( obj ), ObjectDatum( obj ) ) )`. Note that by default this CAP operation is not cached. You can change this behaviour by calling `SetCachingToWeak( C, "ObjectDatum" )` resp. `SetCachingToCrisp( C, "ObjectDatum" )`.

2.8 Well-Definedness of Objects

2.8.1 IsWellDefinedForObjects (for IsCapCategoryObject)

▷ `IsWellDefinedForObjects(a)` (operation)

Returns: a boolean

The argument is an object a. The output is `true` if a is well-defined, otherwise the output is `false`.

2.9 SetOfObjects

2.9.1 SetOfObjectsOfCategory (for IsCapCategory)

▷ `SetOfObjectsOfCategory(C)` (attribute)

Returns: a list of CAP category objects

Return a duplicate free list of objects of the category `C`. The ordering of the returned list must always be the same.

2.9.2 SetOfObjects (for IsCapCategory)

▷ `SetOfObjects(C)` (attribute)

Returns: a list of CAP category objects

Return a duplicate free list of objects of the category `C`. The ordering of the returned list must always be the same. The corresponding CAP operation is `SetOfObjectsOfCategory`.

2.10 Projectives

For a given object A in an abelian category having enough projectives, the following commands allow us to compute some projective object P together with an epimorphism $\pi : P \to A$.

2.10.1 SomeProjectiveObject (for IsCapCategoryObject)

▷ `SomeProjectiveObject(A)` (attribute)

Returns: an object

The argument is an object A. The output is some projective object P for which there exists an epimorphism $\pi : P \to A$.

2.10.2 EpimorphismFromSomeProjectiveObject (for IsCapCategoryObject)

▷ `EpimorphismFromSomeProjectiveObject(A)` (attribute)

Returns: a morphism in $\mathrm{Hom}(P,A)$

The argument is an object A. The output is an epimorphism $\pi : P \to A$ with P a projective object that equals the output of SomeProjectiveObject(A).

2.10.3 EpimorphismFromSomeProjectiveObjectWithGivenSomeProjectiveObject (for IsCapCategoryObject, IsCapCategoryObject)

▷ `EpimorphismFromSomeProjectiveObjectWithGivenSomeProjectiveObject(A, P)` (operation)

Returns: a morphism in $\mathrm{Hom}(P,A)$

The arguments are an object A and a projective object P that equals the output of SomeProjectiveObject(A). The output is an epimorphism $\pi : P \to A$.

2.10.4 ProjectiveLift (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `ProjectiveLift(pi, epsilon)` (operation)

Returns: a morphism in $\mathrm{Hom}(P,B)$

The arguments are a morphism $\pi : P \to A$ with P a projective, and an epimorphism $\varepsilon : B \to A$. The output is a morphism $\lambda : P \to B$ such that $\varepsilon \circ \lambda = \pi$.

2.11 Injectives

For a given object A in an abelian category having enough injectives, the following commands allow us to compute some injective object I together with a monomorphism $\iota : A \to I$.

2.11.1 SomeInjectiveObject (for IsCapCategoryObject)

▷ `SomeInjectiveObject(A)` (attribute)

Returns: an object

The argument is an object A. The output is some injective object I for which there exists a monomorphism $\iota : A \to I$.

2.11.2 MonomorphismIntoSomeInjectiveObject (for IsCapCategoryObject)

▷ `MonomorphismIntoSomeInjectiveObject(A)` (attribute)

Returns: a morphism in $\mathrm{Hom}(I,A)$

The argument is an object A. The output is a monomorphism $\iota : A \to I$ with I an injective object that equals the output of SomeInjectiveObject(A).

2.11.3 MonomorphismIntoSomeInjectiveObjectWithGivenSomeInjectiveObject (for IsCapCategoryObject, IsCapCategoryObject)

▷ `MonomorphismIntoSomeInjectiveObjectWithGivenSomeInjectiveObject(A, I)` (operation)

Returns: a morphism in $\mathrm{Hom}(I,A)$

The arguments are an object A and an injective object I that equals the output of SomeInjectiveObject(A). The output is a monomorphism $\iota : A \to I$.

2.11.4 InjectiveColift (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `InjectiveColift(iota, beta)` (operation)

Returns: a morphism in $\mathrm{Hom}(A,I)$

The arguments are a monomorphism $\iota : B \to A$ and a morphism $\beta : B \to I$ where I is an injective object. The output is a morphism $\lambda : A \to I$ such that $\lambda \circ \iota = \beta$.

2.12 Simplified Objects

Let i be a positive integer or ∞. For a given object A, an i-th simplified object of A consists of

- an object A_i,
- an isomorphism $\iota_A^i : A \to A_i$.

The idea is that the greater the i, the "simpler" the A_i (but this could mean the harder the computation) with ∞ as a possible value.

2.12.1 Simplify (for IsCapCategoryObject)

▷ `Simplify(A)` (attribute)

Returns: an object

The argument is an object A. The output is a simplified object A_∞.

2.12.2 SimplifyObject (for IsCapCategoryObject, IsObject)

▷ `SimplifyObject(A, i)` (operation)

Returns: an object

The arguments are an object A and a positive integer i or `infinity`. The output is a simplified object A_i.

2.12.3 SimplifyObject_IsoFromInputObject (for IsCapCategoryObject, IsObject)

▷ `SimplifyObject_IsoFromInputObject(A, i)` (operation)

Returns: a morphism in $\mathrm{Hom}(A, A_i)$

The arguments are an object A and a positive integer i or `infinity`. The output is an isomorphism to a simplified object $\iota_A^i : A \to A_i$.

2.12.4 SimplifyObject_IsoToInputObject (for IsCapCategoryObject, IsObject)

▷ `SimplifyObject_IsoToInputObject(A, i)` (operation)

Returns: a morphism in $\mathrm{Hom}(A_i, A)$

The arguments are an object A and a positive integer i or `infinity`. The output is an isomorphism from a simplified object $(\iota_A^i)^{-1} : A_i \to A$ which is the inverse of the output of `SimplifyObject_IsoFromInputObject`.

2.13 Dimensions

2.13.1 ProjectiveDimension (for IsCapCategoryObject)

▷ `ProjectiveDimension(A)` (attribute)

Returns: a nonnegative integer or infinity

The argument is an object A. The output is a the projective dimension of A.

2.13.2 InjectiveDimension (for IsCapCategoryObject)

▷ `InjectiveDimension(A)` (attribute)

Returns: a nonnegative integer or infinity

The argument is an object A. The output is a the injective dimension of A.

Chapter 3

Morphisms

Any GAP object satisfying `IsCapCategoryMorphism` can be added to a category and then becomes a morphism in this category. Any morphism can belong to one or no category. After a GAP object is added to the category, it knows which things can be computed in its category and to which category it belongs. It knows categorical properties and attributes, and the functions for existential quantifiers can be applied to the morphism.

3.1 Attributes for the Type of Morphisms

3.1.1 CapCategory (for IsCapCategoryMorphism)

▷ `CapCategory(`*`alpha`*`)` (attribute)

Returns: a category

The argument is a morphism α. The output is the category **C** to which α was added.

3.1.2 Source (for IsCapCategoryMorphism)

▷ `Source(`*`alpha`*`)` (attribute)

Returns: an object

The argument is a morphism $\alpha : a \rightarrow b$. The output is its source a.

3.1.3 Range (for IsCapCategoryMorphism)

▷ `Range(`*`alpha`*`)` (attribute)

Returns: an object

The argument is a morphism $\alpha : a \rightarrow b$. The output is its range b.

3.1.4 Target (for IsCapCategoryMorphism)

▷ `Target(`*`alpha`*`)` (attribute)

Returns: an object

The argument is a morphism $\alpha : a \rightarrow b$. The output is its target b.

3.2 Adding Morphisms to a Category

3.2.1 Add (for IsCapCategory, IsCapCategoryMorphism)

▷ `Add(category, morphism)` (operation)

Adds *morphism* as a morphism to *category*.

3.2.2 AddMorphism (for IsCapCategory, IsAttributeStoringRep)

▷ `AddMorphism(category, morphism)` (operation)

Adds *morphism* as a morphism to *category*. If *morphism* already lies in the filter `IsCapCategoryMorphism`, the operation `Add` (3.2.1) can be used instead.

3.2.3 CreateCapCategoryMorphismWithAttributes

▷ `CreateCapCategoryMorphismWithAttributes(category, source, range[, attr1, val1, attr2, val2, ...])` (function)

Returns: a morphism

Creates a morphism in *category* with the given attributes.

3.2.4 AsCapCategoryMorphism

▷ `AsCapCategoryMorphism(category, source, value, range)` (function)

Returns: a morphism

EXPERIMENTAL: This specification might change any time without prior notice. Views *value* as a morphism from *source* to *range* in *category*.

3.2.5 AsPrimitiveValue (for IsCapCategoryMorphism)

▷ `AsPrimitiveValue(morphism)` (attribute)
▷ `AsInteger(morphism)` (attribute)
▷ `AsHomalgMatrix(morphism)` (attribute)

Returns: a value

EXPERIMENTAL: This specification might change any time without prior notice. Views a morphism obtained via `AsCapCategoryMorphism` (3.2.4) as a primitive value again. Here, the word *primitive* means *primitive from the perspective of the category*. For example, from the perspective of an opposite category, morphisms of the underlying category are primitive values. The attribute is chosen according to the morphism datum type:

- For `IsInt`, the attribute `AsInteger` is used.
- For `IsHomalgMatrix`, the attribute `AsHomalgMatrix` is used.

In all other cases or if no morphism datum type is given, the attribute `AsPrimitiveValue` is used.

3.3 Morphism constructors

3.3.1 MorphismConstructor (for IsCapCategoryObject, IsObject, IsCapCategoryObject)

▷ `MorphismConstructor(S, a, T)` (operation)

Returns: a morphism in $\mathrm{Hom}(S,T)$

The arguments are two objects S and T in a category, and a morphism datum a (type and semantics of the morphism datum depend on the category). The output is a morphism in $\mathrm{Hom}(S,T)$ defined by a. Note that by default this CAP operation is not cached. You can change this behaviour by calling `SetCachingToWeak( C, "MorphismConstructor" )` resp. `SetCachingToCrisp( C, "MorphismConstructor" )`.

3.3.2 MorphismDatum (for IsCapCategoryMorphism)

▷ `MorphismDatum(mor)` (attribute)

Returns: depends on the category

The argument is a CAP category morphism *mor*. The output is a datum which can be used to construct *mor*, that is, `IsEqualForMorphisms( mor, MorphismConstructor( Source( mor ), MorphismDatum( mor ), Range( mor ) ) )`. Note that by default this CAP operation is not cached. You can change this behaviour by calling `SetCachingToWeak( C, "MorphismDatum" )` resp. `SetCachingToCrisp( C, "MorphismDatum" )`.

3.4 Categorical Properties of Morphisms

3.4.1 IsMonomorphism (for IsCapCategoryMorphism)

▷ `IsMonomorphism(alpha)` (property)

Returns: a boolean

The argument is a morphism α. The output is `true` if α is a monomorphism, otherwise the output is `false`.

3.4.2 IsEpimorphism (for IsCapCategoryMorphism)

▷ `IsEpimorphism(alpha)` (property)

Returns: a boolean

The argument is a morphism α. The output is `true` if α is an epimorphism, otherwise the output is `false`.

3.4.3 IsIsomorphism (for IsCapCategoryMorphism)

▷ `IsIsomorphism(alpha)` (property)

Returns: a boolean

The argument is a morphism α. The output is `true` if α is an isomorphism, otherwise the output is `false`.

3.4.4 IsSplitMonomorphism (for IsCapCategoryMorphism)

▷ `IsSplitMonomorphism(alpha)` (property)

Returns: a boolean

The argument is a morphism α. The output is `true` if α is a split monomorphism, otherwise the output is `false`.

3.4.5 IsSplitEpimorphism (for IsCapCategoryMorphism)

▷ `IsSplitEpimorphism(alpha)` (property)

Returns: a boolean

The argument is a morphism α. The output is `true` if α is a split epimorphism, otherwise the output is `false`.

3.4.6 IsOne (for IsCapCategoryMorphism)

▷ `IsOne(alpha)` (property)

Returns: a boolean

The argument is a morphism $\alpha : a \rightarrow a$. The output is `true` if α is congruent to the identity of a, otherwise the output is `false`.

3.4.7 IsIdempotent (for IsCapCategoryMorphism)

▷ `IsIdempotent(alpha)` (property)

Returns: a boolean

The argument is a morphism $\alpha : a \rightarrow a$. The output is `true` if $\alpha^2 \sim_{a,a} \alpha$, otherwise the output is `false`.

3.5 Random Morphisms

CAP provides two principal methods to generate random morphisms with or without fixed source and range:

- *By integers*: The integer is simply a parameter that can be used to create a random morphism.
- *By lists*: The list is used when creating a random morphism would need more than one parameter. Lists offer more flexibility at the expense of the genericity of the methods. This happens because lists that are valid as input in some category may be not valid for other categories. Hence, these operations are not thought to be used in generic categorical algorithms.

3.5.1 RandomMorphismWithFixedSourceByInteger (for IsCapCategoryObject, IsInt)

▷ `RandomMorphismWithFixedSourceByInteger(a, n)` (operation)

Returns: a morphism in $\mathrm{Hom}(a,b)$

The arguments are an object a in a category C and an integer n. The output is a random morphism $\alpha : a \rightarrow b$ for some object b in C. If C is equipped with the methods `RandomObjectByInteger` and `RandomMorphismWithFixedSourceAndRangeByInteger` and C is an Ab-category, then `RandomMorphismWithFixedSourceByInteger`(C,a,n) can be derived as `RandomMorphismWithFixedSourceAndRangeByInteger`$(C,a,b,1+$`Log2Int`$(n))$ where b is computed via `RandomObjectByInteger`(C,n).

3.5.2 RandomMorphismWithFixedSourceByList (for IsCapCategoryObject, IsList)

▷ `RandomMorphismWithFixedSourceByList(a, L)` (operation)

Returns: a morphism in $\mathrm{Hom}(a,b)$

The arguments are an object a in a category C and a list L. The output is a random morphism $\alpha : a \to b$ for some object b in C. If C is equipped with the methods `RandomObjectByList` and `RandomMorphismWithFixedSourceAndRangeByList` and C is an Ab-category, then `RandomMorphismWithFixedSourceByList`(C,a,L) can be derived as `RandomMorphismWithFixedSourceAndRangeByList`$(C,a,b,L[2])$ where b is computed via `RandomObjectByList`$(C,L[1])$.

3.5.3 RandomMorphismWithFixedRangeByInteger (for IsCapCategoryObject, IsInt)

▷ `RandomMorphismWithFixedRangeByInteger(b, n)` (operation)

Returns: a morphism in $\mathrm{Hom}(a,b)$

The arguments are an object b in a category C and an integer n. The output is a random morphism $\alpha : a \to b$ for some object a in C. If C is equipped with the methods `RandomObjectByInteger` and `RandomMorphismWithFixedSourceAndRangeByInteger` and C is an Ab-category, then `RandomMorphismWithFixedRangeByInteger`(C,b,n) can be derived as `RandomMorphismWithFixedSourceAndRangeByInteger`$(C,a,b,1+$`Log2Int`$(n))$ where a is computed via `RandomObjectByInteger`(C,n).

3.5.4 RandomMorphismWithFixedRangeByList (for IsCapCategoryObject, IsList)

▷ `RandomMorphismWithFixedRangeByList(b, L)` (operation)

Returns: a morphism in $\mathrm{Hom}(a,b)$

The arguments are an object b in a category C and a list L. The output is a random morphism $\alpha : a \to b$ for some object a in C. If C is equipped with the methods `RandomObjectByList` and `RandomMorphismWithFixedSourceAndRangeByList` and C is an Ab-category, then `RandomMorphismWithFixedRangeByList`(C,b,L) can be derived as `RandomMorphismWithFixedSourceAndRangeByList`$(C,a,b,L[2])$ where a is computed via `RandomObjectByList`$(C,L[1])$.

3.5.5 RandomMorphismWithFixedSourceAndRangeByInteger (for IsCapCategoryObject, IsCapCategoryObject, IsInt)

▷ `RandomMorphismWithFixedSourceAndRangeByInteger(a, b, n)` (operation)

Returns: a morphism in $\mathrm{Hom}(a,b)$

The arguments are two objects a and b in a category C and an integer n. The output is a random morphism $\alpha : a \to b$ in C.

3.5.6 RandomMorphismWithFixedSourceAndRangeByList (for IsCapCategoryObject, IsCapCategoryObject, IsList)

▷ `RandomMorphismWithFixedSourceAndRangeByList(a, b, L)` (operation)

Returns: a morphism in $\mathrm{Hom}(a,b)$

This operation is not a CAP basic operation The arguments are two objects a and b in a category C and a list L. The output is a random morphism $\alpha : a \rightarrow b$ in C.

3.5.7 RandomMorphismByInteger (for IsCapCategory, IsInt)

▷ `RandomMorphismByInteger(C, n)` (operation)

Returns: a morphism in C

The arguments are a category C and an integer n. The output is a random morphism in C. The operation can be derived in three different ways:

- If C is equipped with the methods `RandomObjectByInteger` and `RandomMorphismWithFixedSourceAndRangeByInteger` and C is an Ab-category, then `RandomMorphism`(C,n) can be derived as `RandomMorphismWithFixedSourceAndRangeByInteger`$(C,a,b,$1+`Log2Int`$(n))$ where a and b are computed via `RandomObjectByInteger`(C,n).
- If C is equipped with the methods `RandomObjectByInteger` and `RandomMorphismWithFixedSourceByInteger`, then `RandomMorphism`(C,n) can be derived as `RandomMorphismWithFixedSourceByInteger`$(C,a,$ 1+`Log2Int`$(n))$ where a is computed via `RandomObjectByInteger`(C,n).
- If C is equipped with the methods `RandomObjectByInteger` and `RandomMorphismWithFixedRangeByInteger`, then `RandomMorphism`(C,n) can be derived as `RandomMorphismWithFixedRangeByInteger`$(C,b,$ 1+`Log2Int`$(n))$ where b is computed via `RandomObjectByInteger`(C,n).

3.5.8 RandomMorphismByList (for IsCapCategory, IsList)

▷ `RandomMorphismByList(C, L)` (operation)

Returns: a morphism in C

The arguments are a category C and a list L. The output is a random morphism in C. The operation can be derived in three different ways:

- If C is equipped with the methods `RandomObjectByList` and `RandomMorphismWithFixedSourceAndRangeByList` and C is an Ab-category, then `RandomMorphism`(C,L) can be derived as `RandomMorphismWithFixedSourceAndRangeByList`$(C,a,b,L[3])$ where a and b are computed via `RandomObjectByList`$(C,L[i])$ for $i = 1,2$ respectively.
- If C is equipped with the methods `RandomObjectByList` and `RandomMorphismWithFixedSourceByList`, then `RandomMorphism`(C,L) can be derived as `RandomMorphismWithFixedSourceByList`$(C,a,L[2])$ where a is computed via `RandomObjectByList`$(C,L[1])$.

- If C is equipped with the methods `RandomObjectByList` and `RandomMorphismWithFixedRangeByList`, then `RandomMorphism`(C,L) can be derived as `RandomMorphismWithFixedRangeByList`$(C,b,L[2])$ where b is computed via `RandomObjectByList`$(C,L[1])$.

3.5.9 RandomMorphismWithFixedSource (for IsCapCategoryObject, IsInt)

▷ `RandomMorphismWithFixedSource(a, n)` (operation)
▷ `RandomMorphismWithFixedSource(a, L)` (operation)
▷ `RandomMorphismWithFixedRange(b, n)` (operation)
▷ `RandomMorphismWithFixedRange(b, L)` (operation)
▷ `RandomMorphismWithFixedSourceAndRange(a, b, n)` (operation)
▷ `RandomMorphismWithFixedSourceAndRange(a, b, L)` (operation)
▷ `RandomMorphism(a, b, n)` (operation)
▷ `RandomMorphism(a, b, L)` (operation)
▷ `RandomMorphism(C, n)` (operation)
▷ `RandomMorphism(C, L)` (operation)

These are convenient methods and they, depending on the input, delegate to one of the above methods.

3.6 Non-Categorical Properties of Morphisms

Non-categorical properties are not stable under equivalences of categories.

3.6.1 IsEqualToIdentityMorphism (for IsCapCategoryMorphism)

▷ `IsEqualToIdentityMorphism(alpha)` (property)

Returns: a boolean

The argument is a morphism $\alpha : a \to b$. The output is `true` if $\alpha = \mathrm{id}_a$, otherwise the output is `false`.

3.6.2 IsEqualToZeroMorphism (for IsCapCategoryMorphism)

▷ `IsEqualToZeroMorphism(alpha)` (property)

Returns: a boolean

The argument is a morphism $\alpha : a \to b$. The output is `true` if $\alpha = 0$, otherwise the output is `false`.

3.6.3 IsEndomorphism (for IsCapCategoryMorphism)

▷ `IsEndomorphism(alpha)` (property)

Returns: a boolean

The argument is a morphism α. The output is `true` if α is an endomorphism, otherwise the output is `false`.

3.6.4 IsAutomorphism (for IsCapCategoryMorphism)

▷ `IsAutomorphism(alpha)` (property)

Returns: a boolean

The argument is a morphism α. The output is `true` if α is an automorphism, otherwise the output is `false`.

3.7 Equality and Congruence for Morphisms

3.7.1 IsCongruentForMorphisms (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `IsCongruentForMorphisms(alpha, beta)` (operation)

Returns: a boolean

The arguments are two morphisms $\alpha, \beta : a \to b$. The output is `true` if $\alpha \sim_{a,b} \beta$, otherwise the output is `false`.

3.7.2 IsEqualForMorphisms (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `IsEqualForMorphisms(alpha, beta)` (operation)

Returns: a boolean

The arguments are two morphisms $\alpha, \beta : a \to b$. The output is `true` if $\alpha = \beta$, otherwise the output is `false`.

3.7.3 IsEqualForMorphismsOnMor (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `IsEqualForMorphismsOnMor(alpha, beta)` (operation)

Returns: a boolean

The arguments are two morphisms $\alpha : a \to b, \beta : c \to d$. The output is `true` if $\alpha = \beta$, otherwise the output is `false`.

3.8 Basic Operations for Morphisms in Ab-Categories

3.8.1 IsZeroForMorphisms (for IsCapCategoryMorphism)

▷ `IsZeroForMorphisms(alpha)` (property)

Returns: a boolean

The argument is a morphism $\alpha : a \to b$. The output is `true` if $\alpha \sim_{a,b} 0$, otherwise the output is `false`.

3.8.2 AdditionForMorphisms (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `AdditionForMorphisms(alpha, beta)` (operation)

Returns: a morphism in $\mathrm{Hom}(a,b)$

The arguments are two morphisms $\alpha, \beta : a \rightarrow b$. The output is the addition $\alpha + \beta$. Note: The addition has to be compatible with the congruence of morphisms.

3.8.3 SubtractionForMorphisms (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `SubtractionForMorphisms(alpha, beta)` (operation)

Returns: a morphism in $\mathrm{Hom}(a,b)$

The arguments are two morphisms $\alpha, \beta : a \rightarrow b$. The output is the addition $\alpha - \beta$. Note: The addition has to be compatible with the congruence of morphisms.

3.8.4 AdditiveInverseForMorphisms (for IsCapCategoryMorphism)

▷ `AdditiveInverseForMorphisms(alpha)` (attribute)

Returns: a morphism in $\mathrm{Hom}(a,b)$

The argument is a morphism $\alpha : a \rightarrow b$. The output is its additive inverse $-\alpha$. Note: The addition has to be compatible with the congruence of morphisms.

3.8.5 MultiplyWithElementOfCommutativeSemiringForMorphisms (for IsSemiringElement, IsCapCategoryMorphism)

▷ `MultiplyWithElementOfCommutativeSemiringForMorphisms(r, alpha)` (operation)

Returns: a morphism in $\mathrm{Hom}(a,b)$

The arguments are an element r of a commutative semiring and a morphism $\alpha : a \rightarrow b$. The output is the multiplication with the semiring element $r \cdot \alpha$. Note: The multiplication has to be compatible with the congruence of morphisms.

3.8.6 * (for IsSemiringElement, IsCapCategoryMorphism)

▷ `*(r, alpha)` (operation)

Returns: a morphism in $\mathrm{Hom}(a,b)$

This is a convenience method. It has two arguments. The first argument is either a rational number q or an element r of a commutative semiring R. The second argument is a morphism $\alpha : a \rightarrow b$ in a linear category over the commutative semiring R. In the case where the first element is a rational number, this method tries to interpret q as an element r of R via `R!.interpret_rationals_func`. If no such interpretation exists, this method throws an error. The output is the multiplication with the semiring element $r \cdot \alpha$.

3.8.7 ZeroMorphism (for IsCapCategoryObject, IsCapCategoryObject)

▷ `ZeroMorphism(a, b)` (operation)

Returns: a morphism in $\mathrm{Hom}(a,b)$

The arguments are two objects a and b. The output is the zero morphism $0 : a \rightarrow b$.

3.9 Subobject and Factorobject Operations

Subobjects of an object c are monomorphisms with range c and a special function for comparision. Similarly, factorobjects of an object c are epimorphisms with source c and a special function for comparision.

3.9.1 IsEqualAsSubobjects (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `IsEqualAsSubobjects(alpha, beta)` (operation)

Returns: a boolean

The arguments are two subobjects $\alpha : a \to c$, $\beta : b \to c$. The output is `true` if there exists an isomorphism $\iota : a \to b$ such that $\beta \circ \iota \sim_{a,c} \alpha$, otherwise the output is `false`.

3.9.2 IsEqualAsFactorobjects (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `IsEqualAsFactorobjects(alpha, beta)` (operation)

Returns: a boolean

The arguments are two factorobjects $\alpha : c \to a$, $\beta : c \to b$. The output is `true` if there exists an isomorphism $\iota : b \to a$ such that $\iota \circ \beta \sim_{c,a} \alpha$, otherwise the output is `false`.

3.9.3 IsDominating (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `IsDominating(alpha, beta)` (operation)

Returns: a boolean

$$\begin{tikzcd} a \arrow[dr, hook, "\alpha"] \arrow[dd, dashed, "\exists\iota"'] & \\ & c \\ b \arrow[ur, hook, "\beta"'] & \end{tikzcd}$$

In short: Returns `true` iff α is smaller than β. Full description: The arguments are two subobjects $\alpha : a \to c$, $\beta : b \to c$. The output is `true` if there exists a morphism $\iota : a \to b$ such that $\beta \circ \iota \sim_{a,c} \alpha$, otherwise the output is `false`.

3.9.4 IsCodominating (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `IsCodominating(alpha, beta)` (operation)

Returns: a boolean

$$\begin{tikzcd} & a \\ c \arrow[ur, two heads, "\alpha"] \arrow[dr, two heads, "\beta"'] & \\ & b \arrow[uu, dashed, "\exists\iota"'] \end{tikzcd}$$

In short: Returns `true` iff α is smaller than β. Full description: The arguments are two factorobjects $\alpha : c \to a$, $\beta : c \to b$. The output is `true` if there exists a morphism $\iota : b \to a$ such that $\iota \circ \beta \sim_{c,a} \alpha$, otherwise the output is `false`.

3.10 Identity Morphism and Composition of Morphisms

3.10.1 IdentityMorphism (for IsCapCategoryObject)

▷ `IdentityMorphism(a)` (attribute)

Returns: a morphism in $\mathrm{Hom}(a,a)$

The argument is an object a. The output is its identity morphism id_a.

3.10.2 PreCompose (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `PreCompose(`*`alpha, beta`*`)` (operation)

Returns: a morphism in $\mathrm{Hom}(a,c)$

The arguments are two morphisms $\alpha : a \to b$, $\beta : b \to c$. The output is the composition $\beta \circ \alpha : a \to c$.

3.10.3 PreCompose (for IsList)

▷ `PreCompose(`*`L`*`)` (operation)

Returns: a morphism in $\mathrm{Hom}(a_1,a_{n+1})$

This is a convenience method. The argument is a list of morphisms $L = (\alpha_1 : a_1 \to a_2, \alpha_2 : a_2 \to a_3, \dots, \alpha_n : a_n \to a_{n+1})$. The output is the composition $\alpha_n \circ (\alpha_{n-1} \circ (\dots(\alpha_2 \circ \alpha_1)))$.

3.10.4 PreComposeList (for IsCapCategoryObject, IsList, IsCapCategoryObject)

▷ `PreComposeList(`*`s, L, r`*`)` (operation)

Returns: a morphism in $\mathrm{Hom}(s,r)$

The arguments are two objects `s` $= a_1$, `r` $= a_{n+1}$, and a list of morphisms $L = (\alpha_1 : a_1 \to a_2, \alpha_2 : a_2 \to a_3, \dots, \alpha_n : a_n \to a_{n+1})$ in C. The output is the composition $\alpha_n \circ (\alpha_{n-1} \circ (\dots(\alpha_2 \circ \alpha_1)))$. If L is empty, then s must be equal to r and the output is congruent to the identity morphism of s.

3.10.5 PostCompose (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `PostCompose(`*`beta, alpha`*`)` (operation)

Returns: a morphism in $\mathrm{Hom}(a,c)$

The arguments are two morphisms $\beta : b \to c$, $\alpha : a \to b$. The output is the composition $\beta \circ \alpha : a \to c$.

3.10.6 PostCompose (for IsList)

▷ `PostCompose(`*`L`*`)` (operation)

Returns: a morphism in $\mathrm{Hom}(a_1,a_{n+1})$

This is a convenience method. The argument is a list of morphisms $L = (\alpha_n : a_n \to a_{n+1}, \alpha_{n-1} : a_{n-1} \to a_n, \dots, \alpha_1 : a_1 \to a_2)$. The output is the composition $((\alpha_n \circ \alpha_{n-1}) \circ \dots \alpha_2) \circ \alpha_1$.

3.10.7 PostComposeList (for IsCapCategoryObject, IsList, IsCapCategoryObject)

▷ `PostComposeList(s, L, r)` (operation)

Returns: a morphism in $\mathrm{Hom}(s,r)$

The arguments are two objects `s` $= a_1$, `r` $= a_{n+1}$, and a list of morphisms $L = (\alpha_n : a_n \to a_{n+1}, \alpha_{n-1} : a_{n-1} \to a_n, \dots, \alpha_1 : a_1 \to a_2)$. The output is the composition $((\alpha_n \circ \alpha_{n-1}) \circ \dots \alpha_2) \circ \alpha_1$. If L is empty, then s must be equal to r and the output is congruent to the identity morphism of s.

3.10.8 SumOfMorphisms (for IsCapCategoryObject, IsList, IsCapCategoryObject)

▷ `SumOfMorphisms(s, morphisms, r)` (operation)

Returns: a morphism in $\mathrm{Hom}(s,r)$

The arguments are two objects `s`, `r` and a list `morphisms` of morphisms from `s` to `r`. The output is the sum of all elements in `morphisms`, or the zero-morphism from `s` to `r` if `morphisms` is empty.

3.10.9 LinearCombinationOfMorphisms (for IsCapCategoryObject, IsList, IsList, IsCapCategoryObject)

▷ `LinearCombinationOfMorphisms(s, coeffs, mors, r)` (operation)

Returns: a morphism in $\mathrm{Hom}(s,r)$

The arguments are two objects `s`, `r` in some linear category over a semiring R, a list `coeffs` of semiring elements in R and a list `mors` of morphisms from `s` to `r`. The output is the linear combination of the morphisms in `mors` with respect to the coefficients list `coeffs`, or the zero morphism from `s` to `r` if `coeffs` and `mors` are the empty lists.

3.11 Well-Definedness of Morphisms

3.11.1 IsWellDefinedForMorphisms (for IsCapCategoryMorphism)

▷ `IsWellDefinedForMorphisms(alpha)` (operation)

Returns: a boolean

The argument is a morphism α. The output is `true` if α is well-defined, otherwise the output is `false`.

3.11.2 IsWellDefinedForMorphismsWithGivenSourceAndRange (for IsCapCategoryObject, IsCapCategoryMorphism, IsCapCategoryObject)

▷ `IsWellDefinedForMorphismsWithGivenSourceAndRange(source, alpha, range)` (operation)

Returns: a boolean

The arguments are two well-defined objects S and T and a morphism α. The output is `true` if α is a well-defined morphism from S to T, otherwise the output is `false`.

3.12 Lift/Colift

- For any pair of morphisms $\alpha : a \to c$, $\beta : b \to c$, we call each morphism $\alpha/\beta : a \to b$ such that $\beta \circ (\alpha/\beta) \sim_{a,c} \alpha$ a *lift of α along β*.

$$\begin{tikzcd} a \arrow[dr, "\alpha"] \arrow[dd, dashed, "\alpha/\beta"'] & \\ & c \\ b \arrow[ur, "\beta"'] & \end{tikzcd}$$

- For any pair of morphisms $\alpha : a \to c$, $\beta : a \to b$, we call each morphism $\alpha\backslash\beta : c \to b$ such that $(\alpha\backslash\beta) \circ \alpha \sim_{a,b} \beta$ a *colift of β along α*.

$$\begin{tikzcd} & c \arrow[dd, dashed, "\alpha\backslash\beta"] \\ a \arrow[ur, "\alpha"] \arrow[dr, "\beta"'] & \\ & b \end{tikzcd}$$

Note that such lifts (or colifts) do not have to be unique. So in general, we do not expect that algorithms computing lifts (or colifts) do this in a functorial way. Thus the operations `Lift` and `Colift` are not regarded as categorical operations, but only as set-theoretic operations.

3.12.1 LiftAlongMonomorphism (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `LiftAlongMonomorphism(iota, tau)` (operation)

Returns: a morphism in $\mathrm{Hom}(t,k)$

The arguments are a monomorphism $\iota : k \hookrightarrow a$ and a morphism $\tau : t \to a$ such that there is a morphism $u : t \to k$ with $\iota \circ u \sim_{t,a} \tau$. The output is such a u.

3.12.2 ColiftAlongEpimorphism (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `ColiftAlongEpimorphism(epsilon, tau)` (operation)

Returns: a morphism in $\mathrm{Hom}(c,t)$

The arguments are an epimorphism $\varepsilon : a \to c$ and a morphism $\tau : a \to t$ such that there is a morphism $u : c \to t$ with $u \circ \varepsilon \sim_{a,t} \tau$. The output is such a u.

3.12.3 IsLiftableAlongMonomorphism (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `IsLiftableAlongMonomorphism(iota, tau)` (operation)

Returns: a boolean

The arguments are a monomorphism $\iota : k \hookrightarrow a$ and a morphism $\tau : t \to a$. The output is `true` if there exists a morphism $u : t \to k$ with $\iota \circ u \sim_{t,a} \tau$. Otherwise, the output is `false`.

3.12.4 IsColiftableAlongEpimorphism (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `IsColiftableAlongEpimorphism(epsilon, tau)` (operation)

Returns: a boolean

The arguments are an epimorphism $\varepsilon : a \to c$ and a morphism $\tau : a \to t$. The output is `true` if there exists a morphism $u : c \to t$ with $u \circ \varepsilon \sim_{a,t} \tau$. Otherwise, the output is `false`.

3.12.5 Lift (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `Lift(alpha, beta)` (operation)

Returns: a morphism in $\mathrm{Hom}(a,b)$

The arguments are two morphisms $\alpha : a \to c$, $\beta : b \to c$ such that a lift $\alpha/\beta : a \to b$ of α along β exists. The output is such a lift $\alpha/\beta : a \to b$. Recall that a lift $\alpha/\beta : a \to b$ of α along β is a morphism such that $\beta \circ (\alpha/\beta) \sim_{a,c} \alpha$.

3.12.6 LiftOrFail (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `LiftOrFail(alpha, beta)` (operation)

Returns: a morphism in $\mathrm{Hom}(a,b) + \{\texttt{fail}\}$

This is a convenience operation. The arguments are two morphisms $\alpha : a \to c$, $\beta : b \to c$. The output is a lift $\alpha/\beta : a \to b$ of α along β if such a lift exists or `fail` if it doesn't. Recall that a lift $\alpha/\beta : a \to b$ of α along β is a morphism such that $\beta \circ (\alpha/\beta) \sim_{a,c} \alpha$.

3.12.7 IsLiftable (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `IsLiftable(alpha, beta)` (operation)

Returns: a boolean

The arguments are two morphisms $\alpha : a \to c$, $\beta : b \to c$. The output is `true` if there exists a lift $\alpha/\beta : a \to b$ of α along β, i.e., a morphism such that $\beta \circ (\alpha/\beta) \sim_{a,c} \alpha$. Otherwise, the output is `false`.

3.12.8 Colift (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `Colift(alpha, beta)` (operation)

Returns: a morphism in $\mathrm{Hom}(c,b)$

The arguments are two morphisms $\alpha : a \to c$, $\beta : a \to b$ such that a colift $\alpha\backslash\beta : c \to b$ of β along α exists. The output is such a colift $\alpha\backslash\beta : c \to b$. Recall that a colift $\alpha\backslash\beta : c \to b$ of β along α is a morphism such that $(\alpha\backslash\beta) \circ \alpha \sim_{a,b} \beta$.

3.12.9 ColiftOrFail (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `ColiftOrFail(alpha, beta)` (operation)

Returns: a morphism in $\mathrm{Hom}(c,b) + \{\texttt{fail}\}$

This is a convenience operation. The arguments are two morphisms $\alpha : a \to c$, $\beta : a \to b$. The output is a colift $\alpha\backslash\beta : c \to b$ of β along α if such a colift exists or `fail` if it doesn't. Recall that a colift $\alpha\backslash\beta : c \to b$ of β along α is a morphism such that $(\alpha\backslash\beta) \circ \alpha \sim_{a,b} \beta$.

3.12.10 IsColiftable (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `IsColiftable(alpha, beta)` (operation)

Returns: a boolean

The arguments are two morphisms $\alpha : a \to c$, $\beta : a \to b$. The output is `true` if there exists a colift $\alpha\backslash\beta : c \to b$ of β along α., i.e., a morphism such that $(\alpha\backslash\beta) \circ \alpha \sim_{a,b} \beta$. Otherwise, the output is `false`.

3.13 Inverses

Let $\alpha : a \to b$ be a morphism. An inverse of α is a morphism $\alpha^{-1} : b \to a$ such that $\alpha \circ \alpha^{-1} \sim_{b,b} \mathrm{id}_b$ and $\alpha^{-1} \circ \alpha \sim_{a,a} \mathrm{id}_a$.

$$\begin{tikzcd} a \arrow[loop left, "\mathrm{id}_a"] \arrow[r, "\alpha"] & b \arrow[l, bend left, "\alpha^{-1}"] \arrow[loop right, "\mathrm{id}_b"] \end{tikzcd}$$

3.13.1 InverseForMorphisms (for IsCapCategoryMorphism)

▷ `InverseForMorphisms(alpha)` (operation)

Returns: a morphism in $\mathrm{Hom}(b,a)$

The argument is an isomorphism $\alpha : a \to b$. The output is its inverse $\alpha^{-1} : b \to a$.

3.13.2 PreInverseForMorphisms (for IsCapCategoryMorphism)

▷ `PreInverseForMorphisms(alpha)` (operation)

Returns: a morphism in $\mathrm{Hom}(b,a)$

The argument is a split-epimorphism $\alpha : a \to b$. The output is a pre-inverse $\iota : b \to a$ of α, i.e., ι satisfies $\alpha \circ \iota \sim_{b,b} \mathrm{id}_b$. The morphism ι is also known as a section or a right-inverse of α.

3.13.3 PostInverseForMorphisms (for IsCapCategoryMorphism)

▷ `PostInverseForMorphisms(alpha)` (operation)

Returns: a morphism in $\mathrm{Hom}(b,a)$

The argument is a split-monomorphism $\alpha : a \to b$. The output is a post-inverse $\pi : b \to a$ of α, i.e., π satisfies $\pi \circ \alpha \sim_{a,a} \mathrm{id}_a$. The morphism π is also known as a contraction or a left-inverse of α.

3.14 Tool functions for caches

3.14.1 IsEqualForCacheForMorphisms (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `IsEqualForCacheForMorphisms(phi, psi)` (operation)

Returns: true or false

By default, CAP uses caches to store the values of Categorical operations. To get a value out of the cache, one needs to compare the input of a basic operation with its previous input. To compare morphisms in the category, IsEqualForCacheForMorphisms is used. By default, IsEqualForCacheForMorphisms falls back to IsEqualForCache (see ToolsForHomalg), which in turn defaults to recursive comparison for lists and `IsIdenticalObj` in all other cases. If you add a function via `AddIsEqualForCacheForMorphisms`, that function is used instead. A function $F : a,b \mapsto bool$ is expected there. The output has to be true or false. Fail is not allowed in this context.

3.15 IsHomSetInhabited

3.15.1 IsHomSetInhabited (for IsCapCategoryObject, IsCapCategoryObject)

▷ `IsHomSetInhabited(A, B)` (operation)

Returns: a boolean

The arguments are two objects `A` and `B`. The output is `true` if there exists a morphism from `A` to `B`, otherwise the output is `false`.

3.16 SetOfMorphisms

3.16.1 SetOfMorphismsOfFiniteCategory (for IsCapCategory)

▷ `SetOfMorphismsOfFiniteCategory(C)` (attribute)

Returns: a list of a CAP category morphisms

Return a duplicate free list of morphisms of the finite category `C`.

3.16.2 SetOfMorphisms (for IsCapCategory)

▷ `SetOfMorphisms(C)` (attribute)

Returns: a list of CAP category objects

Return a duplicate free list of morphisms of the finite category `C`. The corresponding CAP operation is `SetOfMorphismsOfFiniteCategory`.

3.17 SetOfGeneratingMorphisms

3.17.1 SetOfGeneratingMorphismsOfCategory (for IsCapCategory)

▷ `SetOfGeneratingMorphismsOfCategory(C)` (attribute)

Returns: a list of a CAP category morphisms

Return a list of generating morphisms of the finitely generated category `C`.

3.17.2 SetOfGeneratingMorphisms (for IsCapCategory)

▷ `SetOfGeneratingMorphisms(C)` (attribute)

Returns: a list of a CAP category morphisms

Return a list of generating morphisms of the finitely generated category `C`. The corresponding CAP operation is `SetOfGeneratingMorphismsOfCategory`.

3.18 Homomorphism structures

Homomorphism structures are way to "oversee" the homomorphisms between two given objects. Let C, D be categories. A D-homomorphism structure for C consists of the following data:

- a functor $H : C^{\mathrm{op}} \times C \to D$ (when C and D are Ab-categories, H is assumed to be bilinear).
- an object $1 \in D$, called the distinguished object,
- a bijection $\nu : \mathrm{Hom}_C(a,b) \simeq \mathrm{Hom}_D(1, H(a,b))$ natural in $a,b \in C$.

3.18.1 HomomorphismStructureOnObjects (for IsCapCategoryObject, IsCapCategoryObject)

▷ `HomomorphismStructureOnObjects(a, b)` (operation)

Returns: an object in D

The arguments are two objects a,b in C. The output is the value of the homomorphism structure on objects $H(a,b)$.

3.18.2 HomomorphismStructureOnMorphisms (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `HomomorphismStructureOnMorphisms(alpha, beta)` (operation)

Returns: a morphism in $\mathrm{Hom}_D(H(a',b),H(a,b'))$

The arguments are two morphisms $\alpha : a \to a', \beta : b \to b'$ in C. The output is the value of the homomorphism structure on morphisms $H(\alpha,\beta)$.

3.18.3 HomomorphismStructureOnMorphismsWithGivenObjects (for IsCapCategoryObject, IsCapCategoryMorphism, IsCapCategoryMorphism, IsCapCategoryObject)

▷ `HomomorphismStructureOnMorphismsWithGivenObjects(s, alpha, beta, r)` (operation)

Returns: a morphism in $\mathrm{Hom}_D(H(a',b),H(a,b'))$

The arguments are an object $s = H(a',b)$ in D, two morphisms $\alpha : a \to a', \beta : b \to b'$ in C, and an object $r = H(a,b')$ in D. The output is the value of the homomorphism structure on morphisms $H(\alpha,\beta)$.

3.18.4 DistinguishedObjectOfHomomorphismStructure (for IsCapCategory)

▷ `DistinguishedObjectOfHomomorphismStructure(C)` (attribute)

Returns: an object in D

The argument is a category C. The output is the distinguished object 1 in D of the homomorphism structure.

3.18.5 InterpretMorphismAsMorphismFromDistinguishedObjectToHomomorphismStructure (for IsCapCategoryMorphism)

▷ `InterpretMorphismAsMorphismFromDistinguishedObjectToHomomorphismStructure(alpha)` (attribute)

Returns: a morphism in $\mathrm{Hom}_D(1,H(a,a'))$

The argument is a morphism $\alpha : a \to a'$ in C. The output is the corresponding morphism $\nu(\alpha) : 1 \to H(a,a')$ in D of the homomorphism structure.

3.18.6 InterpretMorphismAsMorphismFromDistinguishedObjectToHomomorphismStructureWithG (for IsCapCategoryObject, IsCapCategoryMorphism, IsCapCategoryObject)

▷ `InterpretMorphismAsMorphismFromDistinguishedObjectToHomomorphismStructureWithGivenObjects( alpha, r)` (operation)

Returns: a morphism in $\mathrm{Hom}_D(1,r)$

The arguments are the distinguished object 1, a morphism $\alpha : a \to a'$, and the object $r = H(a,a')$. The output is the corresponding morphism $\nu(\alpha) : 1 \to r$ in D of the homomorphism structure.

3.18.7 InterpretMorphismFromDistinguishedObjectToHomomorphismStructureAsMorphism (for IsCapCategoryObject, IsCapCategoryObject, IsCapCategoryMorphism)

▷ `InterpretMorphismFromDistinguishedObjectToHomomorphismStructureAsMorphism(a, a', iota)` (operation)

Returns: a morphism in $\mathrm{Hom}_C(a,a')$

The arguments are objects a,a' in C and a morphism $\iota : 1 \to H(a,a')$ in D. The output is the corresponding morphism $\nu^{-1}(\iota) : a \to a'$ in C of the homomorphism structure.

3.18.8 SolveLinearSystemInAbCategory (for IsList, IsList, IsList)

▷ `SolveLinearSystemInAbCategory(alpha, beta, gamma)` (operation)

Returns: a list of morphisms $[X_1,\dots,X_n]$

The arguments are three lists α, β, and γ. The first list α (the left coefficients) is a list of list of morphisms $\alpha_{ij} : A_i \to B_j$, where $i = 1\dots m$ and $j = 1\dots n$ for integers $m,n \geq 1$. The second list β (the right coefficients) is a list of list of morphisms $\beta_{ij} : C_j \to D_i$, where $i = 1\dots m$ and $j = 1\dots n$. The third list γ (the right side) is a list of morphisms $\gamma_i : A_i \to D_i$, where $i = 1,\dots,m$. Assumes that a solution to the linear system defined by α, β, γ exists, i.e., there exist morphisms $X_j : B_j \to C_j$ for $j = 1\dots n$ such that $\sum_{j=1}^n \alpha_{ij} \cdot X_j \cdot \beta_{ij} = \gamma_i$ for all $i = 1\dots m$. The output is list of such morphisms $X_j : B_j \to C_j$ for $j = 1\dots n$.

3.18.9 SolveLinearSystemInAbCategoryOrFail (for IsList, IsList, IsList)

▷ `SolveLinearSystemInAbCategoryOrFail(alpha, beta, gamma)` (operation)

Returns: a list of morphisms $[X_1,\dots,X_n]$ or `fail`

This is a convenience operation. Like `SolveLinearSystemInAbCategory`, but without the assumption that a solution exists. If no solution exists, `fail` is returned.

3.18.10 MereExistenceOfSolutionOfLinearSystemInAbCategory (for IsList, IsList, IsList)

▷ `MereExistenceOfSolutionOfLinearSystemInAbCategory(alpha, beta, gamma)` (operation)

Returns: a boolean

Like `SolveLinearSystemInAbCategory`, but the output is simply `true` if a solution exists, `false` otherwise.

3.18.11 MereExistenceOfUniqueSolutionOfLinearSystemInAbCategory (for IsList, IsList, IsList)

▷ `MereExistenceOfUniqueSolutionOfLinearSystemInAbCategory(alpha, beta, gamma)` (operation)

Returns: a boolean

Like `SolveLinearSystemInAbCategory`, but the output is simply `true` if a unique solution exists, and `false` otherwise.

3.18.12 MereExistenceOfUniqueSolutionOfHomogeneousLinearSystemInAbCategory (for IsList, IsList)

▷ `MereExistenceOfUniqueSolutionOfHomogeneousLinearSystemInAbCategory(alpha, beta)` (operation)

Returns: a boolean

Like `SolveLinearSystemInAbCategory` but the output is `true` if the homogeneous system has only the trivial zero solution, and `false` otherwise.

3.18.13 BasisOfSolutionsOfHomogeneousLinearSystemInLinearCategory (for IsList, IsList)

▷ `BasisOfSolutionsOfHomogeneousLinearSystemInLinearCategory(alpha, beta)` (operation)

Returns: a list of lists of morphisms $[X^1,\dots,X^t]$

The arguments are two lists of lists α and β of morphisms in a linear category over a commutative ring k. The first list α (the left coefficients) is a list of list of morphisms $\alpha_{ij}: A_i \to B_j$, where $i = 1,\dots,m$ and $j = 1,\dots,n$ for integers $m,n \geq 1$. The second list β (the right coefficients) is a list of list of morphisms $\beta_{ij}: C_j \to D_i$, where $i = 1,\dots,m$ and $j = 1,\dots,n$. The output is a generating set $[X^1,\dots,X^t]$ for the solutions of the homogeneous linear system: $\sum_{j=1}^{n} \alpha_{ij}\cdot X_j \cdot \beta_{ij} = 0,\ i = 1,\dots,m.$ Particularly, each X^k is a list (of length n) of morphisms $X_j^k: B_j \to C_j, j = 1,\dots,n.$

3.18.14 BasisOfSolutionsOfHomogeneousDoubleLinearSystemInLinearCategory (for IsList, IsList, IsList, IsList)

▷ `BasisOfSolutionsOfHomogeneousDoubleLinearSystemInLinearCategory(alpha, beta, gamma, delta)` (operation)

Returns: a list of lists of morphisms $[X^1,\dots,X^t]$

The arguments are four lists of lists α, β, γ, δ of morphisms in some linear category over commutative ring. Each of α and γ is a list of list of morphisms $\alpha_{ij},\gamma_{ij}: A_i \to B_j$, where $i = 1,\dots,m$ and $j = 1,\dots,n$ for integers $m,n \geq 1$. Each of β and δ is also a list of list of morphisms $\beta_{ij},\delta_{ij}: C_j \to D_i$, where $i = 1,\dots,m$ and $j = 1,\dots,n$. The output is a generating set $[X^1,\dots,X^t]$ for the solutions of the homogeneous linear system defined by α, β, γ and δ, i.e., $\sum_{j=1}^{n} \alpha_{ij}\cdot X_j^k \cdot \beta_{ij} = \sum_{j=1}^{n} \gamma_{ij}\cdot X_j^k \cdot \delta_{ij}$ for all $i = 1,\dots,m$ and all $k = 1,\dots,t$.

3.18.15 BasisOfSolutionsOfHomogeneousDoubleLinearSystemInLinearCategory (for IsList, IsList)

▷ `BasisOfSolutionsOfHomogeneousDoubleLinearSystemInLinearCategory(alpha, delta)` (operation)

Returns: a list of lists of morphisms $[X^1,\dots,X^t]$

The arguments are two lists of lists α, δ morphisms in some linear category over commutative ring. α is a list of list of morphisms $\alpha_{ij}: A_i \to B_j$ and δ is a list of list of morphisms $\delta_{ij}: C_j \to D_i$, where $i = 1,\dots,m$ and $j = 1,\dots,n$ for integers $m,n \geq 1$. The method delegates to `BasisOfSolutionsOfHomogeneousDoubleLinearSystemInLinearCategory` applied on α, β, γ, δ where

- β_{ij} equals IdentityMorphism(Source(δ_{ij})) if δ_{ij} is an endomorphism, and ZeroMorphism(Source(δ_{ij}), Range(δ_{ij})) otherwise,
- γ_{ij} equals IdentityMorphism(Source(α_{ij})) if α_{ij} is an endomorphism, and ZeroMorphism(Source(α_{ij}), Range(α_{ij})) otherwise;

for all $i = 1,\dots,m$ and $j = 1,\dots,n$.

3.18.16 HomStructure (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `HomStructure(alpha, beta)` (operation)

Returns: a morphism in $\mathrm{Hom}_D(H(a',b),H(a,b'))$

This is a convenience method. The arguments are two morphisms $\alpha : a \to a', \beta : b \to b'$ in C. The output is `HomomorphismStructureOnMorphisms` called on α, β.

3.18.17 HomStructure (for IsCapCategoryMorphism, IsCapCategoryObject)

▷ `HomStructure(alpha, b)` (operation)

Returns: a morphism in $\mathrm{Hom}_D(H(a',b),H(a,b))$

This is a convenience method. The arguments are a morphism $\alpha : a \to a'$ and an object b in C. The output is `HomomorphismStructureOnMorphisms` called on α, id_b.

3.18.18 HomStructure (for IsCapCategoryObject, IsCapCategoryMorphism)

▷ `HomStructure(a, beta)` (operation)

Returns: a morphism in $\mathrm{Hom}_D(H(a,b),H(a,b'))$

This is a convenience method. The arguments are an object a and a morphism $\beta : b \to b'$ in C. The output is `HomomorphismStructureOnMorphisms` called on id_a, β.

3.18.19 HomStructure (for IsCapCategoryObject, IsCapCategoryObject)

▷ `HomStructure(a, b)` (operation)

Returns: an object

This is a convenience method. The arguments are two objects a and b in C. The output is `HomomorphismStructureOnObjects` called on a,b.

3.18.20 HomStructure (for IsCapCategoryMorphism)

▷ `HomStructure(arg)` (operation)

This is a convenience method for `InterpretMorphismAsMorphismFromDistinguishedObjectToHomomorphism`

3.18.21 HomStructure (for IsCapCategoryObject, IsCapCategoryObject, IsCapCategoryMorphism)

▷ `HomStructure(arg1, arg2, arg3)` (operation)

This is a convenience method for InterpretMorphismFromDistinguishedObjectToHomomorphismStructureA

3.18.22 HomStructure (for IsCapCategory)

▷ HomStructure(*arg*) (operation)

This is a convenience method for DistinguishedObjectOfHomomorphismStructure.

3.18.23 ExtendRangeOfHomomorphismStructureByFullEmbedding (for IsCapCategory, IsCapCategory, IsFunction, IsFunction, IsFunction, IsFunction)

▷ ExtendRangeOfHomomorphismStructureByFullEmbedding(*C*, *E*, *object_function*, *morphism_function*, *object_function_inverse*, *morphism_function_inverse*) (operation)
▷ HomomorphismStructureOnObjectsExtendedByFullEmbedding(*C*, *E*, *a*, *b*) (operation)
▷ HomomorphismStructureOnMorphismsExtendedByFullEmbedding(*C*, *E*, *alpha*, *beta*) (operation)
▷ HomomorphismStructureOnMorphismsWithGivenObjectsExtendedByFullEmbedding(*C*, *E*, *s*, *alpha*, *beta*, *r*) (operation)
▷ DistinguishedObjectOfHomomorphismStructureExtendedByFullEmbedding(*C*, *E*) (operation)
▷ InterpretMorphismAsMorphismFromDistinguishedObjectToHomomorphismStructureExtendedByFullEmb *E*, *alpha*) (operation)
▷ InterpretMorphismAsMorphismFromDistinguishedObjectToHomomorphismStructureWithGivenObjectsE *E*, *distinguished_object*, *alpha*, *r*) (operation)
▷ InterpretMorphismFromDistinguishedObjectToHomomorphismStructureAsMorphismExtendedByFullEmb *E*, *a*, *a'*, *iota*) (operation)

Returns: nothing

If $\iota : D \to E$ is a full embedding of categories, every D-homomorphism structure for a category C extends to a E-homomorphism structure for C. This operations accepts four functions and installs operations DistinguishedObjectOfHomomorphismStructureExtendedByFullEmbedding, HomomorphismStructureOnObjectsExtendedByFullEmbedding etc. which correspond to the E-homomorphism structure for C. Note: To distinguish embeddings in different categories, in addition to C also E is passed to the operations. When using this with different embeddings with the range category E, only the last embedding will be used. The arguments are:

- object_function gets the categories C and E and an object in D.
- morphism_function gets the categories C and E, an object in E, a morphism in D and another object in E. The objects are the results of object_function applied to the source and range of the morphism.
- object_function_inverse gets the categories C and E and an object in E.
- morphism_function_inverse gets the categories C and E, an object in D, a morphism in E and another object in D. The objects are the results of object_function_inverse applied to the source and range of the morphism.

object_function and morphism_function define the embedding. object_function_inverse and morphism_function_inverse define the inverse of the embedding on its image.

3.18.24 ExtendRangeOfHomomorphismStructureByIdentityAsFullEmbedding (for IsCapCategory)

▷ `ExtendRangeOfHomomorphismStructureByIdentityAsFullEmbedding(C)` (operation)

Returns: nothing

Chooses the identity on D as the full embedding in `ExtendRangeOfHomomorphismStructureByFullEmbedding` (3.18.23). This is useful to handle this case as a degenerate case of `ExtendRangeOfHomomorphismStructureByFullEmbedding` (3.18.23).

3.18.25 MorphismsOfExternalHom (for IsCapCategoryObject, IsCapCategoryObject)

▷ `MorphismsOfExternalHom(a, b)` (operation)

Returns: a list of morphisms in $\mathrm{Hom}(a,b)$

The argument are two objects `a`, `b`. The output is a list of all morphisms from `a` to `b`.

3.18.26 BasisOfExternalHom (for IsCapCategoryObject, IsCapCategoryObject)

▷ `BasisOfExternalHom(a, b)` (operation)

Returns: a list of morphisms in $\mathrm{Hom}_C(a,b)$

The arguments are objects a,b in a k-linear category C. The output is a list L of morphisms which is a basis of $\mathrm{Hom}_C(a,b)$ in the sense that any given morphism $\alpha : a \to b$ can uniquely be written as a linear combination of L with the coefficients in `CoefficientsOfMorphism`(α).

3.18.27 CoefficientsOfMorphism (for IsCapCategoryMorphism)

▷ `CoefficientsOfMorphism(alpha)` (attribute)

Returns: a list of elements in k

This is a convenience method. The argument is a morphism $\alpha : a \to b$ in a k-linear category C. The output is a list of coefficients of α with respect to the list `BasisOfExternalHom(a,b)`.

3.19 Simplified Morphisms

Let $\phi : A \to B$ be a morphism. There are several different natural ways to look at ϕ as an object in an ambient category:

- $\mathrm{Hom}(A,B)$, the set of homomorphisms with the equivalence relation `IsCongruentForMorphisms` regarded as a category,
- $\sum_A \mathrm{Hom}(A,B)$, the category of morphisms where the range is fixed,
- $\sum_B \mathrm{Hom}(A,B)$, the category of morphisms where the source is fixed,
- $\sum_{A,B} \mathrm{Hom}(A,B)$, the category of morphisms where neither source nor range is fixed,

and furthermore, if ϕ happens to be an endomorphism $A \to A$, we also have

- $\sum_A \mathrm{Hom}(A,A)$, the category of endomorphisms.

Let **C** be one of the categories above in which ϕ may reside as an object, and let i be a non-negative integer or ∞. CAP provides commands for passing from ϕ to ϕ_i, where ϕ_i is isomorphic to ϕ in **C**, but "simpler". The idea is that the greater the i, the "simpler" the ϕ_i (but this could mean the harder the computation), with ∞ as a possible value. The case $i = 0$ defaults to the identity operator for all simplifications. For the Add-operatations, only the cases $i \geq 1$ have to be given as functions.

If we regard ϕ as an object in the category $\mathrm{Hom}(A,B)$, ϕ_i is again in $\mathrm{Hom}(A,B)$ such that $\phi \sim_{A,B} \phi_i$. This case is handled by the following commands:

3.19.1 SimplifyMorphism (for IsCapCategoryMorphism, IsObject)

▷ `SimplifyMorphism(phi, i)` (operation)

Returns: a morphism in $\mathrm{Hom}(A,B)$

The arguments are a morphism $\phi : A \to B$ and a non-negative integer i or `infinity`. The output is a simplified morphism ϕ_i.

If we regard ϕ as an object in the category $\sum_A \mathrm{Hom}(A,B)$, then ϕ_i is a morphism of type $A_i \to B$ and there is an isomorphism $\sigma_i : A \to A_i$ such that $\phi_i \circ \sigma_i \sim_{A,B} \phi$. This case is handled by the following commands:

3.19.2 SimplifySource (for IsCapCategoryMorphism, IsObject)

▷ `SimplifySource(phi, i)` (operation)

Returns: a morphism in $\mathrm{Hom}(A_i,B)$

The arguments are a morphism $\phi : A \to B$ and a non-negative integer i or `infinity`. The output is a simplified morphism with simplified source $\phi_i : A_i \to B$.

3.19.3 SimplifySource_IsoToInputObject (for IsCapCategoryMorphism, IsObject)

▷ `SimplifySource_IsoToInputObject(phi, i)` (operation)

Returns: a morphism in $\mathrm{Hom}(A_i,A)$

The arguments are a morphism $\phi : A \to B$ and a non-negative integer i or `infinity`. The output is the isomorphism $(\sigma_i)^{-1} : A_i \to A$.

3.19.4 SimplifySource_IsoFromInputObject (for IsCapCategoryMorphism, IsObject)

▷ `SimplifySource_IsoFromInputObject(phi, i)` (operation)

Returns: a morphism in $\mathrm{Hom}(A,A_i)$

The arguments are a morphism $\phi : A \to B$ and a non-negative integer i or `infinity`. The output is the isomorphism $\sigma_i : A \to A_i$.

If we regard ϕ as an object in the category $\sum_B \mathrm{Hom}(A,B)$, then ϕ_i is a morphism of type $A \to B_i$ and there is an isomorphism $\rho_i : B \to B_i$ such that $\rho_i^{-1} \circ \phi_i \sim_{A,B} \phi$. This case is handled by the following commands:

3.19.5 SimplifyRange (for IsCapCategoryMorphism, IsObject)

▷ `SimplifyRange(phi, i)` (operation)

Returns: a morphism in $\mathrm{Hom}(A,B_i)$

The arguments are a morphism $\phi : A \to B$ and a non-negative integer i or `infinity`. The output is a simplified morphism with simplified range $\phi_i : A \to B_i$.

3.19.6 SimplifyRange_IsoToInputObject (for IsCapCategoryMorphism, IsObject)

▷ `SimplifyRange_IsoToInputObject(phi, i)` (operation)

Returns: a morphism in $\mathrm{Hom}(B_i,B)$

The arguments are a morphism $\phi : A \to B$ and a non-negative integer i or `infinity`. The output is the isomorphism $(\rho_i)^{-1} : B_i \to B$.

3.19.7 SimplifyRange_IsoFromInputObject (for IsCapCategoryMorphism, IsObject)

▷ `SimplifyRange_IsoFromInputObject(phi, i)` (operation)

Returns: a morphism in $\mathrm{Hom}(B,B_i)$

The arguments are a morphism $\phi : A \to B$ and a non-negative integer i or `infinity`. The output is the isomorphism $\rho_i : B \to B_i$.

If we regard ϕ as an object in the category $\sum_{A,B} \mathrm{Hom}(A,B)$, then ϕ_i is a morphism of type $A_i \to B_i$ and there is are isomorphisms $\sigma_i : A \to A_i$ and $\rho_i : B \to B_i$ such that $\rho_i^{-1} \circ \phi_i \circ \sigma_i \sim_{A,B} \phi$. This case is handled by the following commands:

3.19.8 SimplifySourceAndRange (for IsCapCategoryMorphism, IsObject)

▷ `SimplifySourceAndRange(phi, i)` (operation)

Returns: a morphism in $\mathrm{Hom}(A_i,B_i)$

The arguments are a morphism $\phi : A \to B$ and a non-negative integer i or `infinity`. The output is a simplified morphism with simplified source and range $\phi_i : A_i \to B_i$.

3.19.9 SimplifySourceAndRange_IsoToInputRange (for IsCapCategoryMorphism, IsObject)

▷ `SimplifySourceAndRange_IsoToInputRange(phi, i)` (operation)

Returns: a morphism in $\mathrm{Hom}(B_i,B)$

The arguments are a morphism $\phi : A \to B$ and a non-negative integer i or `infinity`. The output is the isomorphism $(\rho_i)^{-1} : B_i \to B$.

3.19.10 SimplifySourceAndRange_IsoFromInputRange (for IsCapCategoryMorphism, IsObject)

▷ `SimplifySourceAndRange_IsoFromInputRange(phi, i)` (operation)

Returns: a morphism in $\mathrm{Hom}(B,B_i)$

The arguments are a morphism $\phi : A \to B$ and a non-negative integer i or `infinity`. The output is the isomorphism $\rho_i : B \to B_i$.

3.19.11 SimplifySourceAndRange_IsoToInputSource (for IsCapCategoryMorphism, IsObject)

▷ `SimplifySourceAndRange_IsoToInputSource(phi, i)` (operation)

Returns: a morphism in $\mathrm{Hom}(A_i, A)$

The arguments are a morphism $\phi : A \to B$ and a non-negative integer i or `infinity`. The output is the isomorphism $(\sigma_i)^{-1} : A_i \to A$.

3.19.12 SimplifySourceAndRange_IsoFromInputSource (for IsCapCategoryMorphism, IsObject)

▷ `SimplifySourceAndRange_IsoFromInputSource(phi, i)` (operation)

Returns: a morphism in $\mathrm{Hom}(A, A_i)$

The arguments are a morphism $\phi : A \to B$ and a non-negative integer i or `infinity`. The output is the isomorphism $\sigma_i : A \to A_i$.

If $\phi : A \to A$ is an endomorphism, we may regard it as an object in the category $\sum_A \mathrm{Hom}(A, A)$. In this case ϕ_i is a morphism of type $A_i \to A_i$ and there is an isomorphism $\sigma_i : A \to A_i$ such that $\sigma_i^{-1} \circ \phi_i \circ \sigma_i \sim_{A,A} \phi$. This case is handled by the following commands:

3.19.13 SimplifyEndo (for IsCapCategoryMorphism, IsObject)

▷ `SimplifyEndo(phi, i)` (operation)

Returns: a morphism in $\mathrm{Hom}(A_i, A_i)$

The arguments are an endomorphism $\phi : A \to A$ and a non-negative integer i or `infinity`. The output is a simplified endomorphism $\phi_i : A_i \to A_i$.

3.19.14 SimplifyEndo_IsoToInputObject (for IsCapCategoryMorphism, IsObject)

▷ `SimplifyEndo_IsoToInputObject(phi, i)` (operation)

Returns: a morphism in $\mathrm{Hom}(A_i, A)$

The arguments are an endomorphism $\phi : A \to A$ and a non-negative integer i or `infinity`. The output is the isomorphism $(\sigma_i)^{-1} : A_i \to A$.

3.19.15 SimplifyEndo_IsoFromInputObject (for IsCapCategoryMorphism, IsObject)

▷ `SimplifyEndo_IsoFromInputObject(phi, i)` (operation)

Returns: a morphism in $\mathrm{Hom}(A, A_i)$

The arguments are an endomorphism $\phi : A \to A$ and a non-negative integer i or `infinity`. The output is the isomorphism $\sigma_i : A \to A_i$.

3.19.16 Simplify (for IsCapCategoryMorphism)

▷ `Simplify(phi)` (attribute)

Returns: a morphism in $\mathrm{Hom}(A_\infty, B_\infty)$

This is a convenient method. The argument is a morphism $\phi : A \to B$. The output is a "simplified" version of ϕ that may change the source and range of ϕ (up to isomorphism). To be precise, the output is an ∞-th simplified morphism of $(\iota_A^\infty)^{-1} \circ \phi \circ \iota_A^\infty$.

3.20 Reduction by split epi summands

Let $\alpha : A \to B$ be a morphism in an additive category. Suppose we are given direct sum decompositions of $A \simeq A' \oplus A''$ and $B \simeq B' \oplus B''$ such that

$$\begin{tikzcd} A' \oplus A'' \arrow[r, "\alpha' \oplus \alpha''"] & B' \oplus B'' \\ A \arrow[u] \arrow[r, "\alpha"] & B \arrow[u] \end{tikzcd}$$

If α'' is a split epimorphism, then we call $\alpha' : A' \to B'$ *some reduction of* α *by split epi summands*. The inclusions/projections of the decompositions into direct sums induce commutative diagrams

$$\begin{tikzcd} A' \arrow[r, "\alpha'"] & B' \\ A \arrow[u] \arrow[r, "\alpha"] & B \arrow[u, "\beta"'] \end{tikzcd}$$

and

$$\begin{tikzcd} A' \arrow[r, "\alpha'"] \arrow[d] & B' \arrow[d, "\beta'"] \\ A \arrow[r, "\alpha"] & B \end{tikzcd}$$

3.20.1 SomeReductionBySplitEpiSummand (for IsCapCategoryMorphism)

▷ `SomeReductionBySplitEpiSummand(alpha)` (attribute)

Returns: a morphism in $\mathrm{Hom}(A', B')$

The argument is a morphism $\alpha : A \to B$. The output is some reduction of α by split epi summands $\alpha' : A' \to B'$.

3.20.2 SomeReductionBySplitEpiSummand_MorphismToInputRange (for IsCapCategoryMorphism)

▷ `SomeReductionBySplitEpiSummand_MorphismToInputRange(alpha)` (attribute)

Returns: a morphism in $\mathrm{Hom}(B', B)$

The argument is a morphism $\alpha : A \to B$. The output is the morphism $\beta' : B' \to B$ linking α with some reduction by split epi summands.

3.20.3 SomeReductionBySplitEpiSummand_MorphismFromInputRange (for IsCapCategoryMorphism)

▷ `SomeReductionBySplitEpiSummand_MorphismFromInputRange(alpha)` (attribute)

Returns: a morphism in $\mathrm{Hom}(B,B')$

The argument is a morphism $\alpha : A \to B$. The output is the morphism $\beta : B \to B'$ linking α with some reduction by split epi summands.

Chapter 4

Category 2-Cells

4.1 Attributes for the Type of 2-Cells

4.1.1 Source (for IsCapCategoryTwoCell)

▷ `Source(c)` (attribute)

Returns: a morphism

The argument is a 2-cell $c : \alpha \to \beta$. The output is its source α.

4.1.2 Range (for IsCapCategoryTwoCell)

▷ `Range(c)` (attribute)

Returns: a morphism

The argument is a 2-cell $c : \alpha \to \beta$. The output is its range β.

4.1.3 Target (for IsCapCategoryTwoCell)

▷ `Target(c)` (attribute)

Returns: a morphism

The argument is a 2-cell $c : \alpha \to \beta$. The output is its target β.

4.2 Adding 2-Cells to a Category

4.2.1 Add (for IsCapCategory, IsCapCategoryTwoCell)

▷ `Add(category, twocell)` (operation)

Adds *twocell* as a 2-cell to *category*.

4.2.2 AddTwoCell (for IsCapCategory, IsAttributeStoringRep)

▷ `AddTwoCell(category, twocell)` (operation)

Adds *twocell* as a 2-cell to *category*. If *twocell* already lies in the filter `IsCapCategoryTwoCell`, the operation `Add` (4.2.1) can be used instead.

4.2.3 CreateCapCategoryTwoCellWithAttributes

▷ `CreateCapCategoryTwoCellWithAttributes(`*`category, source, range[, attr1, val1, attr2, val2, ...]`*`)` (function)

Returns: a twocell

Creates a 2-cell in `category` with the given attributes.

4.3 Identity 2-Cell and Composition of 2-Cells

4.3.1 IdentityTwoCell (for IsCapCategoryMorphism)

▷ `IdentityTwoCell(`*`alpha`*`)` (attribute)

Returns: a 2-cell

The argument is a morphism α. The output is its identity 2-cell $\mathrm{id}_\alpha : \alpha \rightarrow \alpha$.

4.3.2 HorizontalPreCompose (for IsCapCategoryTwoCell, IsCapCategoryTwoCell)

▷ `HorizontalPreCompose(`*`c, d`*`)` (operation)

Returns: a 2-cell

The arguments are two 2-cells $c : \alpha \rightarrow \beta$, $d : \gamma \rightarrow \delta$ between morphisms $\alpha, \beta : a \rightarrow b$ and $\gamma, \delta : b \rightarrow c$. The output is their horizontal composition $d * c : (\gamma \circ \alpha) \rightarrow (\delta \circ \beta)$.

4.3.3 HorizontalPostCompose (for IsCapCategoryTwoCell, IsCapCategoryTwoCell)

▷ `HorizontalPostCompose(`*`d, c`*`)` (operation)

Returns: a 2-cell

The arguments are two 2-cells $d : \gamma \rightarrow \delta$, $c : \alpha \rightarrow \beta$ between morphisms $\alpha, \beta : a \rightarrow b$ and $\gamma, \delta : b \rightarrow c$. The output is their horizontal composition $d * c : (\gamma \circ \alpha) \rightarrow (\delta \circ \beta)$.

4.3.4 VerticalPreCompose (for IsCapCategoryTwoCell, IsCapCategoryTwoCell)

▷ `VerticalPreCompose(`*`c, d`*`)` (operation)

Returns: a 2-cell

The arguments are two 2-cells $c : \alpha \rightarrow \beta$, $d : \beta \rightarrow \gamma$ between morphisms $\alpha, \beta, \gamma : a \rightarrow b$. The output is their vertical composition $d \circ c : \alpha \rightarrow \gamma$.

4.3.5 VerticalPostCompose (for IsCapCategoryTwoCell, IsCapCategoryTwoCell)

▷ `VerticalPostCompose(`*`d, c`*`)` (operation)

Returns: a 2-cell

The arguments are two 2-cells $d : \beta \rightarrow \gamma$, $c : \alpha \rightarrow \beta$ between morphisms $\alpha, \beta, \gamma : a \rightarrow b$. The output is their vertical composition $d \circ c : \alpha \rightarrow \gamma$.

4.4 Well-Definedness for 2-Cells

4.4.1 IsWellDefinedForTwoCells (for IsCapCategoryTwoCell)

▷ `IsWellDefinedForTwoCells(c)` (operation)

Returns: a boolean

The argument is a 2-cell c. The output is `true` if c is well-defined, otherwise the output is `false`.

Chapter 5

Category of Categories

Categories itself with functors as morphisms form a category Cat. So the data structure of `CapCategorys` is designed to be objects in a category. This category is implemented in `CapCat`. For every category, the corresponding object in Cat can be obtained via `AsCatObject`. The implemetation of the category of categories offers a data structure for functors. Those are implemented as morphisms in this category, so functors can be handled like morphisms in a category. Also convenience functions to install functors as methods are implemented (in order to avoid `ApplyFunctor`).

5.1 The Category Cat

5.1.1 CapCat

▷ `CapCat` (global variable)

This variable stores the category of categories. Every category object is constructed as an object in this category, so Cat is constructed when loading the package.

5.2 Categories

5.2.1 IsCapCategoryAsCatObject (for IsCapCategoryObject)

▷ `IsCapCategoryAsCatObject(`*object*`)` (Category)

Returns: `true` or `false`

The GAP category of CAP categories seen as object in Cat.

5.2.2 IsCapFunctor (for IsCapCategoryMorphism)

▷ `IsCapFunctor(`*object*`)` (Category)

Returns: `true` or `false`

The GAP category of functors.

5.2.3 IsCapNaturalTransformation (for IsCapCategoryTwoCell)

▷ `IsCapNaturalTransformation(`*object*`)` (Category)

Returns: `true` or `false`

The GAP category of natural transformations.

5.3 Constructors

5.3.1 AsCatObject (for IsCapCategory)

▷ `AsCatObject(C)` (attribute)

Given a CAP category C, this method returns the corresponding object in Cat. For technical reasons, the filter `IsCapCategory` must not imply the filter `IsCapCategoryObject`. For example, if `InitialObject` is applied to an object, it returns the initial object of its category. If it is applied to a category, it returns the initial object of the category. If a CAP category would be a category object itself, this would be ambiguous. So categories must be wrapped in a CatObject to be an object in Cat. This method returns the wrapper object. The category can be reobtained by `AsCapCategory`.

5.3.2 AsCapCategory (for IsCapCategoryAsCatObject)

▷ `AsCapCategory(C)` (attribute)

For an object C in Cat, this method returns the underlying CAP category. This method is inverse to `AsCatObject`, i.e. AsCapCategory(AsCatObject(A)) = A.

5.4 Functors

Functors are morphisms in Cat, thus they have source and target which are categories. A multivariate functor can be constructed via a product category as source, a presheaf is constructed via the opposite category as source. However, the user can explicitly decide the arity of a functor (which will only have technical implications). Thus, it is for example possible to consider a functor $A \times B \to C$ either as a unary functor with source category $A \times B$ or as a binary functor. Moreover, an object and a morphism function can be added to a functor, to apply it to objects or morphisms in the source category.

5.4.1 CapFunctor (for IsString, IsCapCategory, IsCapCategory)

▷ `CapFunctor(name, A, B)` (operation)
▷ `CapFunctor(name, A, B)` (operation)
▷ `CapFunctor(name, A, B)` (operation)
▷ `CapFunctor(name, A, B)` (operation)

These methods construct a unary CAP functor. The first argument is a string for the functor's name. `A` and `B` are the source and target of the functor, and they can be given as objects in `CapCat` or as a CAP-category.

5.4.2 CapFunctor (for IsString, IsList, IsCapCategory)

▷ `CapFunctor(name, list, B)` (operation)
▷ `CapFunctor(name, list, B)` (operation)

These methods construct a possible multivariate CAP functor. The first argument is a string for the functor's name. The second argument is a list encoding the input signature of the functor. It can be given as a list of pairs $[[A_1,b_1],\dots,[A_n,b_n]]$ where a pair consists of a category A_i (given as an object in `CapCat` or as a CAP-category) and a boolean b_i for $i = 1,\dots,n$. Instead of a pair $[A_i,b_i]$, you can also give simply A_i, which will be interpreted as the pair $[A_i,\texttt{false}]$. The third argument is the target `B` of the functor, and it can be given as an object in `CapCat` or as a CAP-category. The output is a functor with source given by the product category $D_1 \times ... \times D_n$, where $D_i = A_i$ if $b_i = \texttt{false}$, and $D_i = A_i^{\mathrm{op}}$ otherwise.

5.4.3 SourceOfFunctor (for IsCapFunctor)

▷ `SourceOfFunctor(F)` (attribute)

The argument is a functor F. The output is its source as CAP category.

5.4.4 RangeOfFunctor (for IsCapFunctor)

▷ `RangeOfFunctor(F)` (attribute)

The argument is a functor F. The output is its range as CAP category.

5.4.5 AddObjectFunction (for IsCapFunctor, IsFunction)

▷ `AddObjectFunction(F, f)` (operation)

This operation adds a function f to the functor F which can then be applied to objects in the source. The given function f has to take arguments according to the `InputSignature` of F, i.e., if the input signature is given by $[[A_1,b_1],\dots,[A_n,b_n]]$, then f must take n arguments, where the i-th argument is an object in the category A_i (the boolean b_i is ignored). The function should return an object in the range of the functor, except when the automatic call of `AddObject` was enabled via `EnableAddForCategoricalOperations`. In this case the output only has to be a GAP object in `IsAttributeStoringRep`, which will be automatically added as an object to the range of the functor.

5.4.6 FunctorObjectOperation (for IsCapFunctor)

▷ `FunctorObjectOperation(F)` (attribute)

Returns: a GAP operation

The argument is a functor F. The output is the GAP operation realizing the action of F on objects.

5.4.7 AddMorphismFunction (for IsCapFunctor, IsFunction)

▷ `AddMorphismFunction(F, f)` (operation)

This operation adds a function f to the functor F which can then be applied to morphisms in the source. The given function f has to take as its first argument an object s that is equal (via `IsEqualForObjects`) to the source of the result of applying F to the input morphisms. The next arguments of f have to morphisms according to the `InputSignature` of F, i.e., if the input signature is given by $[[A_1,b_1],\dots,[A_n,b_n]]$, then f must take n arguments, where the i-th argument is a morphism in the category A_i (the boolean b_i is ignored). The last argument of f must be an object r that is equal (via `IsEqualForObjects`) to the range of the result of applying F to the input morphisms. The function should return a morphism in the range of the functor, except when the automatic call of `AddMorphism` was enabled via `EnableAddForCategoricalOperations`. In this case the output only has to be a GAP object in `IsAttributeStoringRep` (with attributes `Source` and `Range` containing also GAP objects in `IsAttributeStoringRep`), which will be automatically added as a morphism to the range of the functor.

5.4.8 FunctorMorphismOperation (for IsCapFunctor)

▷ `FunctorMorphismOperation(F)` (attribute)

Returns: a GAP operation

The argument is a functor F. The output is the GAP operation realizing the action of F on morphisms.

5.4.9 ApplyFunctor

▷ `ApplyFunctor(func, A[, B, ...])` (function)

Returns: IsCapCategoryCell

Applies the functor *func* either to

- an object or morphism *A* in the source of *func* or
- to objects or morphisms belonging to the categories in the input signature of *func*.

5.4.10 InputSignature (for IsCapFunctor)

▷ `InputSignature(F)` (attribute)

Returns: IsList

The argument is a functor F. The output is a list of pairs $[[A_1,b_1],\dots,[A_n,b_n]]$ where a pair consists of a CAP-category A_i and a boolean b_i for $i = 1,\dots,n$. The source of F is given by the product category $D_1 \times ... \times D_n$, where $D_i = A_i$ if $b_i =$ `false`, and $D_i = A_i^{\mathrm{op}}$ otherwise.

5.4.11 InstallFunctor (for IsCapFunctor, IsString)

▷ `InstallFunctor(F, s)` (operation)

Returns: nothing

The arguments are a functor F and a string s. To simplify the description of this operation, we let $[[A_1,b_1],\dots,[A_n,b_n]]$ denote the input signature of F. This method tries to install 3 operations: an operation ω_1 with the name s, an operation ω_2 with the name s`OnObjects`, and an operation ω_3 with the name s`OnMorphisms`. The operation ω_1 takes as input either n- objects/morphisms in A_i or a single object/morphism in the source of F, and outputs the result of applying F to this input. ω_2 and ω_3 are the corresponding variants for objects or morphisms only. This function can only be called once for each functor, every further call will be ignored.

5.4.12 IdentityFunctor (for IsCapCategory)

▷ `IdentityFunctor(cat)` (attribute)

Returns: a functor

Returns the identity functor of the category *cat* viewed as an object in the category of categories.

5.4.13 FunctorCanonicalizeZeroObjects (for IsCapCategory)

▷ `FunctorCanonicalizeZeroObjects(cat)` (attribute)

Returns: a functor

Returns the endofunctor of the category *cat* with zero which maps each (object isomorphic to the) zero object to `ZeroObject(cat)` and to itself otherwise. This functor is equivalent to the identity functor.

5.4.14 NaturalIsomorphismFromIdentityToCanonicalizeZeroObjects (for IsCapCategory)

▷ `NaturalIsomorphismFromIdentityToCanonicalizeZeroObjects(cat)` (attribute)

Returns: a natural transformation

Returns the natural isomorphism from the identity functor to `FunctorCanonicalizeZeroObjects(cat)`.

5.4.15 FunctorCanonicalizeZeroMorphisms (for IsCapCategory)

▷ `FunctorCanonicalizeZeroMorphisms(cat)` (attribute)

Returns: a functor

Returns the endofunctor of the category *cat* with zero which maps each object to itself, each morphism ϕ to itself, unless it is congruent to the zero morphism; in this case it is mapped to `ZeroMorphism(Source(`ϕ`), Range(`ϕ`))`. This functor is equivalent to the identity functor.

5.4.16 NaturalIsomorphismFromIdentityToCanonicalizeZeroMorphisms (for IsCapCategory)

▷ `NaturalIsomorphismFromIdentityToCanonicalizeZeroMorphisms(cat)` (attribute)

Returns: a natural transformation

Returns the natural isomorphism from the identity functor to `FunctorCanonicalizeZeroMorphisms(cat)`.

5.4.17 AdditiveFunctorDataFromValuesOnBasisMorphisms (for IsCapCategory, IsCapCategory, IsFunction, IsFunction)

▷ `AdditiveFunctorDataFromValuesOnBasisMorphisms(source, range, F_o, F_m[, options])` (operation)

Returns: a list consisting of two or four functions together with a final record storing cache data

The arguments are two linear categories *source* and *range* over the same commutative ring, together with a function *F_o* on objects and a function *F_m* on basis morphisms. The operation returns the data needed to define the additive functor determined by these values, extending *F_m* from basis morphisms to arbitrary morphisms by linearity.

The result is a list whose first two entries are the induced object function and morphism function. These functions cache values on objects and lazily cache values on basis morphisms via `LazyHList`.

Two options can be passed. The option `known_values_on_objects` initializes the object cache with a pair of parallel lists. The option `is_full` controls whether two additional entries are returned, namely preimage functions on objects and morphisms.

The last entry is always a record containing the internal cache data. If `is_full := true`, this record also contains the cached data for the preimage functions.

5.5 Natural transformations

Natural transformations form the 2-cells of Cat. As such, it is possible to compose them vertically and horizontally, see Section 4.3.

5.5.1 Name (for IsCapNaturalTransformation)

▷ `Name(arg)` (attribute)

Returns: a string

As every functor, every natural transformation has a name attribute. It has to be a string and will be set by the Constructor.

5.5.2 NaturalTransformation (for IsCapFunctor, IsCapFunctor)

▷ `NaturalTransformation([name, ]F, G)` (operation)

Returns: a natural transformation

Constructs a natural transformation between the functors $F\colon A \rightarrow B$ and $G\colon A \rightarrow B$. The string *name* is optional, and, if not given, set automatically from the names of the functors

5.5.3 AddNaturalTransformationFunction (for IsCapNaturalTransformation, IsFunction)

▷ `AddNaturalTransformationFunction(N, func)` (operation)

Adds the function *func* to the natural transformation N. The function should take three arguments. If $N : F \rightarrow G$, the arguments should be $F(A), A, G(A)$. The ouptput should be a morphism, $F(A) \rightarrow G(A)$.

5.5.4 ApplyNaturalTransformation

▷ `ApplyNaturalTransformation(N, A)` (function)

Returns: a morphism

Given a natural transformation $N\colon F \rightarrow G$ and an object A, this function should return the morphism $F(A) \rightarrow G(A)$, corresponding to N.

5.5.5 InstallNaturalTransformation (for IsCapNaturalTransformation, IsString)

▷ `InstallNaturalTransformation(N, name)` (operation)

Installs the natural transformation *N* as operation with the name `name`. Argument for this operation is an object, output is a morphism.

5.5.6 HorizontalPreComposeNaturalTransformationWithFunctor (for IsCapNaturalTransformation, IsCapFunctor)

▷ `HorizontalPreComposeNaturalTransformationWithFunctor(N, F)` (operation)

Returns: a natural transformation

Computes the horizontal composition of the natural transformation *N* and the functor *F*.

5.5.7 HorizontalPreComposeFunctorWithNaturalTransformation (for IsCapFunctor, IsCapNaturalTransformation)

▷ `HorizontalPreComposeFunctorWithNaturalTransformation(F, N)` (operation)

Returns: a natural transformation

Computes the horizontal composition of the functor *F* and the natural transformation *N*.

Chapter 6

Universal Objects

6.1 Kernel

For a given morphism $\alpha : A \to B$, a kernel of α consists of three parts:

- an object K,
- a morphism $\iota : K \to A$ such that $\alpha \circ \iota \sim_{K,B} 0$,
- a dependent function u mapping each morphism $\tau : T \to A$ satisfying $\alpha \circ \tau \sim_{T,B} 0$ to a morphism $u(\tau) : T \to K$ such that $\iota \circ u(\tau) \sim_{T,A} \tau$.

The triple (K, ι, u) is called a *kernel* of α if the morphisms $u(\tau)$ are uniquely determined up to congruence of morphisms. We denote the object K of such a triple by KernelObject(α). We say that the morphism $u(\tau)$ is induced by the *universal property of the kernel*.
KernelObject is a functorial operation. This means: for $\mu : A \to A'$, $\nu : B \to B'$, $\alpha : A \to B$, $\alpha' : A' \to B'$ such that $\nu \circ \alpha \sim_{A,B'} \alpha' \circ \mu$, we obtain a morphism KernelObject$(\alpha) \to$ KernelObject(α').

$$\begin{tikzcd}
T \arrow[d, dashed, "\exists ! u(\tau)"'] \arrow[dr, "\tau"] \arrow[drr, dotted, bend left, "\alpha \circ \tau \sim_{T,B} 0"] & & \\
K \arrow[r, "\iota"] \arrow[rr, dotted, bend right, "\alpha \circ \iota \sim_{K,B} 0"'] & A \arrow[r, "\alpha"] & B
\end{tikzcd}$$

6.1.1 KernelObject (for IsCapCategoryMorphism)

▷ `KernelObject(alpha)` (attribute)

Returns: an object

The argument is a morphism α. The output is the kernel K of α.

6.1.2 KernelEmbedding (for IsCapCategoryMorphism)

▷ `KernelEmbedding(alpha)` (attribute)

Returns: a morphism in $\mathrm{Hom}(\mathrm{KernelObject}(\alpha),A)$

The argument is a morphism $\alpha : A \to B$. The output is the kernel embedding $\iota : \mathrm{KernelObject}(\alpha) \to A$.

6.1.3 KernelEmbeddingWithGivenKernelObject (for IsCapCategoryMorphism, IsCapCategoryObject)

▷ `KernelEmbeddingWithGivenKernelObject(alpha, K)` (operation)

Returns: a morphism in $\mathrm{Hom}(K,A)$

The arguments are a morphism $\alpha : A \to B$ and an object $K = \mathrm{KernelObject}(\alpha)$. The output is the kernel embedding $\iota : K \to A$.

6.1.4 MorphismFromKernelObjectToSink (for IsCapCategoryMorphism)

▷ `MorphismFromKernelObjectToSink(alpha)` (operation)

Returns: the zero morphism in $\mathrm{Hom}(\mathrm{KernelObject}(\alpha),B)$

The argument is a morphism $\alpha : A \to B$. The output is the zero morphism $0 : \mathrm{KernelObject}(\alpha) \to B$.

6.1.5 MorphismFromKernelObjectToSinkWithGivenKernelObject (for IsCapCategoryMorphism, IsCapCategoryObject)

▷ `MorphismFromKernelObjectToSinkWithGivenKernelObject(alpha, K)` (operation)

Returns: the zero morphism in $\mathrm{Hom}(K,B)$

The arguments are a morphism $\alpha : A \to B$ and an object $K = \mathrm{KernelObject}(\alpha)$. The output is the zero morphism $0 : K \to B$.

6.1.6 KernelLift (for IsCapCategoryMorphism, IsCapCategoryObject, IsCapCategoryMorphism)

▷ `KernelLift(alpha, T, tau)` (operation)

Returns: a morphism in $\mathrm{Hom}(T,\mathrm{KernelObject}(\alpha))$

The arguments are a morphism $\alpha : A \to B$, a test object T, and a test morphism $\tau : T \to A$ satisfying $\alpha \circ \tau \sim_{T,B} 0$. For convenience, the test object `T` can be omitted and is automatically derived from `tau` in that case. The output is the morphism $u(\tau) : T \to \mathrm{KernelObject}(\alpha)$ given by the universal property of the kernel.

6.1.7 KernelLiftWithGivenKernelObject (for IsCapCategoryMorphism, IsCapCategoryObject, IsCapCategoryMorphism, IsCapCategoryObject)

▷ `KernelLiftWithGivenKernelObject(alpha, T, tau, K)` (operation)

Returns: a morphism in $\mathrm{Hom}(T,K)$

The arguments are a morphism $\alpha : A \to B$, a test object T, a test morphism $\tau : T \to A$ satisfying $\alpha \circ \tau \sim_{T,B} 0$, and an object $K = \mathrm{KernelObject}(\alpha)$. For convenience, the test object `T` can be omitted and is automatically derived from `tau` in that case. The output is the morphism $u(\tau): T \rightarrow K$ given by the universal property of the kernel.

6.1.8 KernelObjectFunctorial (for IsList)

▷ `KernelObjectFunctorial(L)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{KernelObject}(\alpha), \mathrm{KernelObject}(\alpha'))$

The argument is a list $L = [\alpha : A \rightarrow B, [\mu : A \rightarrow A', \nu : B \rightarrow B'], \alpha' : A' \rightarrow B']$ of morphisms. The output is the morphism $\mathrm{KernelObject}(\alpha) \rightarrow \mathrm{KernelObject}(\alpha')$ given by the functoriality of the kernel.

6.1.9 KernelObjectFunctorial (for IsCapCategoryMorphism, IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `KernelObjectFunctorial(alpha, mu, alpha_prime)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{KernelObject}(\alpha), \mathrm{KernelObject}(\alpha'))$

The arguments are three morphisms $\alpha : A \rightarrow B$, $\mu : A \rightarrow A'$, $\alpha' : A' \rightarrow B'$. The output is the morphism $\mathrm{KernelObject}(\alpha) \rightarrow \mathrm{KernelObject}(\alpha')$ given by the functoriality of the kernel.

6.1.10 KernelObjectFunctorialWithGivenKernelObjects (for IsCapCategoryObject, IsCapCategoryMorphism, IsCapCategoryMorphism,IsCapCategoryMorphism, IsCapCategoryObject)

▷ `KernelObjectFunctorialWithGivenKernelObjects(s, alpha, mu, alpha_prime, r)` (operation)

Returns: a morphism in $\mathrm{Hom}(s, r)$

The arguments are an object $s = \mathrm{KernelObject}(\alpha)$, three morphisms $\alpha : A \rightarrow B$, $\mu : A \rightarrow A'$, $\alpha' : A' \rightarrow B'$, and an object $r = \mathrm{KernelObject}(\alpha')$. The output is the morphism $\mathrm{KernelObject}(\alpha) \rightarrow \mathrm{KernelObject}(\alpha')$ given by the functoriality of the kernel.

6.1.11 KernelObjectFunctorialWithGivenKernelObjects (for IsCapCategoryObject, IsCapCategoryMorphism, IsCapCategoryMorphism,IsCapCategoryMorphism, IsCapCategoryMorphism, IsCapCategoryObject)

▷ `KernelObjectFunctorialWithGivenKernelObjects(s, alpha, mu, nu, alpha_prime, r)` (operation)

Returns: a morphism in $\mathrm{Hom}(s, r)$

The arguments are an object $s = \mathrm{KernelObject}(\alpha)$, four morphisms $\alpha : A \rightarrow B$, $\mu : A \rightarrow A'$, $\nu : B \rightarrow B'$, $\alpha' : A' \rightarrow B'$, and an object $r = \mathrm{KernelObject}(\alpha')$. The output is the morphism $\mathrm{KernelObject}(\alpha) \rightarrow \mathrm{KernelObject}(\alpha')$ given by the functoriality of the kernel.

6.2 Cokernel

For a given morphism $\alpha : A \rightarrow B$, a cokernel of α consists of three parts:

- an object K,
- a morphism $\varepsilon : B \rightarrow K$ such that $\varepsilon \circ \alpha \sim_{A,K} 0$,

- a dependent function u mapping each $\tau : B \to T$ satisfying $\tau \circ \alpha \sim_{A,T} 0$ to a morphism $u(\tau) : K \to T$ such that $u(\tau) \circ \varepsilon \sim_{B,T} \tau$.

The triple (K, ε, u) is called a *cokernel* of α if the morphisms $u(\tau)$ are uniquely determined up to congruence of morphisms. We denote the object K of such a triple by $\mathrm{CokernelObject}(\alpha)$. We say that the morphism $u(\tau)$ is induced by the *universal property of the cokernel*.
CokernelObject is a functorial operation. This means: for $\mu : A \to A'$, $\nu : B \to B'$, $\alpha : A \to B$, $\alpha' : A' \to B'$ such that $\nu \circ \alpha \sim_{A,B'} \alpha' \circ \mu$, we obtain a morphism $\mathrm{CokernelObject}(\alpha) \to \mathrm{CokernelObject}(\alpha')$.

$$\begin{tikzcd}
 & & T \\
A \arrow[r, "\alpha"] \arrow[urr, bend left, dotted, "\tau \circ \alpha \sim_{A,T} 0"] \arrow[rr, bend right, dotted, "\varepsilon \circ \alpha \sim_{A,K} 0"'] & B \arrow[r, "\varepsilon"] \arrow[ur, "\tau"] & K \arrow[u, dashed, "\exists! u(\tau)"']
\end{tikzcd}$$

6.2.1 CokernelObject (for IsCapCategoryMorphism)

▷ `CokernelObject(alpha)` (attribute)

Returns: an object

The argument is a morphism $\alpha : A \to B$. The output is the cokernel K of α.

6.2.2 CokernelProjection (for IsCapCategoryMorphism)

▷ `CokernelProjection(alpha)` (attribute)

Returns: a morphism in $\mathrm{Hom}(B, \mathrm{CokernelObject}(\alpha))$

The argument is a morphism $\alpha : A \to B$. The output is the cokernel projection $\varepsilon : B \to \mathrm{CokernelObject}(\alpha)$.

6.2.3 CokernelProjectionWithGivenCokernelObject (for IsCapCategoryMorphism, IsCapCategoryObject)

▷ `CokernelProjectionWithGivenCokernelObject(alpha, K)` (operation)

Returns: a morphism in $\mathrm{Hom}(B, K)$

The arguments are a morphism $\alpha : A \to B$ and an object $K = \mathrm{CokernelObject}(\alpha)$. The output is the cokernel projection $\varepsilon : B \to \mathrm{CokernelObject}(\alpha)$.

6.2.4 MorphismFromSourceToCokernelObject (for IsCapCategoryMorphism)

▷ `MorphismFromSourceToCokernelObject(alpha)` (operation)

Returns: the zero morphism in $\mathrm{Hom}(A, \mathrm{CokernelObject}(\alpha))$.

The argument is a morphism $\alpha : A \to B$. The output is the zero morphism $0 : A \to \mathrm{CokernelObject}(\alpha)$.

6.2.5 MorphismFromSourceToCokernelObjectWithGivenCokernelObject (for IsCapCategoryMorphism, IsCapCategoryObject)

▷ `MorphismFromSourceToCokernelObjectWithGivenCokernelObject(alpha, K)` (operation)

Returns: the zero morphism in $\mathrm{Hom}(A,K)$.

The argument is a morphism $\alpha : A \rightarrow B$ and an object $K = \mathrm{CokernelObject}(\alpha)$. The output is the zero morphism $0 : A \rightarrow K$.

6.2.6 CokernelColift (for IsCapCategoryMorphism, IsCapCategoryObject, IsCapCategoryMorphism)

▷ `CokernelColift(alpha, T, tau)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{CokernelObject}(\alpha),T)$

The arguments are a morphism $\alpha : A \rightarrow B$, a test object T, and a test morphism $\tau : B \rightarrow T$ satisfying $\tau \circ \alpha \sim_{A,T} 0$. For convenience, the test object `T` can be omitted and is automatically derived from `tau` in that case. The output is the morphism $u(\tau) : \mathrm{CokernelObject}(\alpha) \rightarrow T$ given by the universal property of the cokernel.

6.2.7 CokernelColiftWithGivenCokernelObject (for IsCapCategoryMorphism, IsCapCategoryObject, IsCapCategoryMorphism, IsCapCategoryObject)

▷ `CokernelColiftWithGivenCokernelObject(alpha, T, tau, K)` (operation)

Returns: a morphism in $\mathrm{Hom}(K,T)$

The arguments are a morphism $\alpha : A \rightarrow B$, a test object T, a test morphism $\tau : B \rightarrow T$ satisfying $\tau \circ \alpha \sim_{A,T} 0$, and an object $K = \mathrm{CokernelObject}(\alpha)$. For convenience, the test object `T` can be omitted and is automatically derived from `tau` in that case. The output is the morphism $u(\tau) : K \rightarrow T$ given by the universal property of the cokernel.

6.2.8 CokernelObjectFunctorial (for IsList)

▷ `CokernelObjectFunctorial(L)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{CokernelObject}(\alpha),\mathrm{CokernelObject}(\alpha'))$

The argument is a list $L = [\alpha : A \rightarrow B, [\mu : A \rightarrow A', \nu : B \rightarrow B'], \alpha' : A' \rightarrow B']$. The output is the morphism $\mathrm{CokernelObject}(\alpha) \rightarrow \mathrm{CokernelObject}(\alpha')$ given by the functoriality of the cokernel.

6.2.9 CokernelObjectFunctorial (for IsCapCategoryMorphism, IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `CokernelObjectFunctorial(alpha, nu, alpha_prime)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{CokernelObject}(\alpha),\mathrm{CokernelObject}(\alpha'))$

The arguments are three morphisms $\alpha : A \rightarrow B, \nu : B \rightarrow B', \alpha' : A' \rightarrow B'$. The output is the morphism $\mathrm{CokernelObject}(\alpha) \rightarrow \mathrm{CokernelObject}(\alpha')$ given by the functoriality of the cokernel.

6.2.10 CokernelObjectFunctorialWithGivenCokernelObjects (for IsCapCategoryObject, IsCapCategoryMorphism, IsCapCategoryMorphism,IsCapCategoryMorphism, IsCapCategoryObject)

▷ `CokernelObjectFunctorialWithGivenCokernelObjects(s, alpha, nu, alpha_prime, r)` (operation)

Returns: a morphism in $\mathrm{Hom}(s,r)$

The arguments are an object $s = \mathrm{CokernelObject}(\alpha)$, three morphisms $\alpha : A \rightarrow B, \nu : B \rightarrow B', \alpha' : A' \rightarrow B'$, and an object $r = \mathrm{CokernelObject}(\alpha')$. The output is the morphism $\mathrm{CokernelObject}(\alpha) \rightarrow \mathrm{CokernelObject}(\alpha')$ given by the functoriality of the cokernel.

6.2.11 CokernelObjectFunctorialWithGivenCokernelObjects (for IsCapCategoryObject, IsCapCategoryMorphism, IsCapCategoryMorphism,IsCapCategoryMorphism, IsCapCategoryMorphism, IsCapCategoryObject)

▷ `CokernelObjectFunctorialWithGivenCokernelObjects(s, alpha, mu, nu, alpha_prime, r)` (operation)

Returns: a morphism in $\mathrm{Hom}(s,r)$

The arguments are an object $s = \mathrm{CokernelObject}(\alpha)$, four morphisms $\alpha : A \rightarrow B, \mu : A \rightarrow A', \nu : B \rightarrow B', \alpha' : A' \rightarrow B'$, and an object $r = \mathrm{CokernelObject}(\alpha')$. The output is the morphism $\mathrm{CokernelObject}(\alpha) \rightarrow \mathrm{CokernelObject}(\alpha')$ given by the functoriality of the cokernel.

6.3 Zero Object

A zero object consists of three parts:

- an object Z,
- a function u_{in} mapping each object A to a morphism $u_{\mathrm{in}}(A) : A \rightarrow Z$,
- a function u_{out} mapping each object A to a morphism $u_{\mathrm{out}}(A) : Z \rightarrow A$.

The triple $(Z, u_{\mathrm{in}}, u_{\mathrm{out}})$ is called a *zero object* if the morphisms $u_{\mathrm{in}}(A)$, $u_{\mathrm{out}}(A)$ are uniquely determined up to congruence of morphisms. We denote the object Z of such a triple by ZeroObject. We say that the morphisms $u_{\mathrm{in}}(A)$ and $u_{\mathrm{out}}(A)$ are induced by the *universal property of the zero object*.

6.3.1 ZeroObject (for IsCapCategory)

▷ `ZeroObject(C)` (attribute)

Returns: an object

The argument is a category C. The output is a zero object Z of C.

6.3.2 ZeroObject (for IsCapCategoryCell)

▷ `ZeroObject(c)` (attribute)

Returns: an object

This is a convenience method. The argument is a cell c. The output is a zero object Z of the category C for which $c \in C$.

6.3.3 UniversalMorphismFromZeroObject (for IsCapCategoryObject)

▷ `UniversalMorphismFromZeroObject(A)` (attribute)

Returns: a morphism in $\mathrm{Hom}(\mathrm{ZeroObject}, A)$

The argument is an object A. The output is the universal morphism $u_{\mathrm{out}} : \mathrm{ZeroObject} \to A$.

6.3.4 UniversalMorphismFromZeroObjectWithGivenZeroObject (for IsCapCategoryObject, IsCapCategoryObject)

▷ `UniversalMorphismFromZeroObjectWithGivenZeroObject(A, Z)` (operation)

Returns: a morphism in $\mathrm{Hom}(Z, A)$

The arguments are an object A, and a zero object $Z = \mathrm{ZeroObject}$. The output is the universal morphism $u_{\mathrm{out}} : Z \to A$.

6.3.5 UniversalMorphismIntoZeroObject (for IsCapCategoryObject)

▷ `UniversalMorphismIntoZeroObject(A)` (attribute)

Returns: a morphism in $\mathrm{Hom}(A, \mathrm{ZeroObject})$

The argument is an object A. The output is the universal morphism $u_{\mathrm{in}} : A \to \mathrm{ZeroObject}$.

6.3.6 UniversalMorphismIntoZeroObjectWithGivenZeroObject (for IsCapCategoryObject, IsCapCategoryObject)

▷ `UniversalMorphismIntoZeroObjectWithGivenZeroObject(A, Z)` (operation)

Returns: a morphism in $\mathrm{Hom}(A, Z)$

The arguments are an object A, and a zero object $Z = \mathrm{ZeroObject}$. The output is the universal morphism $u_{\mathrm{in}} : A \to Z$.

6.3.7 MorphismFromZeroObject (for IsCapCategoryObject)

▷ `MorphismFromZeroObject(A)` (attribute)

Returns: a morphism in $\mathrm{Hom}(\mathrm{ZeroObject}, A)$

This is a synonym for `UniversalMorphismFromZeroObject`.

6.3.8 MorphismIntoZeroObject (for IsCapCategoryObject)

▷ `MorphismIntoZeroObject(A)` (attribute)

Returns: a morphism in $\mathrm{Hom}(A, \mathrm{ZeroObject})$

This is a synonym for `UniversalMorphismIntoZeroObject`.

6.3.9 IsomorphismFromZeroObjectToInitialObject (for IsCapCategory)

▷ `IsomorphismFromZeroObjectToInitialObject(C)` (attribute)

Returns: a morphism in $\mathrm{Hom}(\mathrm{ZeroObject}, \mathrm{InitialObject})$

The argument is a category C. The output is the unique isomorphism $\mathrm{ZeroObject} \to \mathrm{InitialObject}$.

6.3.10 IsomorphismFromInitialObjectToZeroObject (for IsCapCategory)

▷ `IsomorphismFromInitialObjectToZeroObject(C)` (attribute)

Returns: a morphism in Hom(InitialObject, ZeroObject)

The argument is a category C. The output is the unique isomorphism InitialObject $\to$ ZeroObject.

6.3.11 IsomorphismFromZeroObjectToTerminalObject (for IsCapCategory)

▷ `IsomorphismFromZeroObjectToTerminalObject(C)` (attribute)

Returns: a morphism in Hom(ZeroObject, TerminalObject)

The argument is a category C. The output is the unique isomorphism ZeroObject $\to$ TerminalObject.

6.3.12 IsomorphismFromTerminalObjectToZeroObject (for IsCapCategory)

▷ `IsomorphismFromTerminalObjectToZeroObject(C)` (attribute)

Returns: a morphism in Hom(TerminalObject, ZeroObject)

The argument is a category C. The output is the unique isomorphism TerminalObject $\to$ ZeroObject.

6.3.13 ZeroObjectFunctorial (for IsCapCategory)

▷ `ZeroObjectFunctorial(C)` (attribute)

Returns: a morphism in Hom(ZeroObject, ZeroObject)

The argument is a category C. The output is the unique morphism ZeroObject $\to$ ZeroObject.

6.3.14 ZeroObjectFunctorialWithGivenZeroObjects (for IsCapCategoryObject, IsCapCategoryObject)

▷ `ZeroObjectFunctorialWithGivenZeroObjects(C, zero_object1, zero_object2)` (operation)

Returns: a morphism in Hom$(zero_object1, zero_object2)$

The argument is a category C and a zero object ZeroObject(C) twice (for compatibility with other functorials). The output is the unique morphism $zero_object1 \to zero_object2$.

6.4 Terminal Object

A terminal object consists of two parts:

- an object T,
- a function u mapping each object A to a morphism $u(A) : A \to T$.

The pair (T, u) is called a *terminal object* if the morphisms $u(A)$ are uniquely determined up to congruence of morphisms. We denote the object T of such a pair by TerminalObject. We say that the morphism $u(A)$ is induced by the *universal property of the terminal object*.
TerminalObject is a functorial operation. This just means: There exists a unique morphism $T \to T$.

6.4.1 TerminalObject (for IsCapCategory)

▷ `TerminalObject(C)` (attribute)

Returns: an object

The argument is a category C. The output is a terminal object T of C.

6.4.2 TerminalObject (for IsCapCategoryCell)

▷ `TerminalObject(c)` (attribute)

Returns: an object

This is a convenience method. The argument is a cell c. The output is a terminal object T of the category C for which $c \in C$.

6.4.3 UniversalMorphismIntoTerminalObject (for IsCapCategoryObject)

▷ `UniversalMorphismIntoTerminalObject(A)` (attribute)

Returns: a morphism in $\mathrm{Hom}(A, \mathrm{TerminalObject})$

The argument is an object A. The output is the universal morphism $u(A) : A \rightarrow \mathrm{TerminalObject}$.

6.4.4 UniversalMorphismIntoTerminalObjectWithGivenTerminalObject (for IsCapCategoryObject, IsCapCategoryObject)

▷ `UniversalMorphismIntoTerminalObjectWithGivenTerminalObject(A, T)` (operation)

Returns: a morphism in $\mathrm{Hom}(A, T)$

The argument are an object A, and an object $T = \mathrm{TerminalObject}$. The output is the universal morphism $u(A) : A \rightarrow T$.

6.4.5 TerminalObjectFunctorial (for IsCapCategory)

▷ `TerminalObjectFunctorial(C)` (attribute)

Returns: a morphism in $\mathrm{Hom}(\mathrm{TerminalObject}, \mathrm{TerminalObject})$

The argument is a category C. The output is the unique morphism $\mathrm{TerminalObject} \rightarrow \mathrm{TerminalObject}$.

6.4.6 TerminalObjectFunctorialWithGivenTerminalObjects (for IsCapCategoryObject, IsCapCategoryObject)

▷ `TerminalObjectFunctorialWithGivenTerminalObjects(C, terminal_object1, terminal_object2)` (operation)

Returns: a morphism in $\mathrm{Hom}(terminal_object1, terminal_object2)$

The argument is a category C and a terminal object $\mathrm{TerminalObject}(C)$ twice (for compatibility with other functorials). The output is the unique morphism $terminal_object1 \rightarrow terminal_object2$.

6.5 Initial Object

An initial object consists of two parts:

- an object I,

- a function u mapping each object A to a morphism $u(A) : I \to A$.

The pair (I,u) is called a *initial object* if the morphisms $u(A)$ are uniquely determined up to congruence of morphisms. We denote the object I of such a triple by InitialObject. We say that the morphism $u(A)$ is induced by the *universal property of the initial object*.
InitialObject is a functorial operation. This just means: There exists a unique morphisms $I \to I$.

6.5.1 InitialObject (for IsCapCategory)

▷ `InitialObject(C)` (attribute)

Returns: an object

The argument is a category C. The output is an initial object I of C.

6.5.2 InitialObject (for IsCapCategoryCell)

▷ `InitialObject(c)` (attribute)

Returns: an object

This is a convenience method. The argument is a cell c. The output is an initial object I of the category C for which $c \in C$.

6.5.3 UniversalMorphismFromInitialObject (for IsCapCategoryObject)

▷ `UniversalMorphismFromInitialObject(A)` (attribute)

Returns: a morphism in Hom(InitialObject,A).

The argument is an object A. The output is the universal morphism $u(A)$: InitialObject $\to A$.

6.5.4 UniversalMorphismFromInitialObjectWithGivenInitialObject (for IsCapCategoryObject, IsCapCategoryObject)

▷ `UniversalMorphismFromInitialObjectWithGivenInitialObject(A, I)` (operation)

Returns: a morphism in Hom(I,A).

The arguments are an object A, and an object $I =$ InitialObject. The output is the universal morphism $u(A)$: InitialObject $\to A$.

6.5.5 InitialObjectFunctorial (for IsCapCategory)

▷ `InitialObjectFunctorial(C)` (attribute)

Returns: a morphism in Hom(InitialObject, InitialObject)

The argument is a category C. The output is the unique morphism InitialObject $\to$ InitialObject.

6.5.6 InitialObjectFunctorialWithGivenInitialObjects (for IsCapCategoryObject, IsCapCategoryObject)

▷ `InitialObjectFunctorialWithGivenInitialObjects(C, initial_object1, initial_object2)` (operation)

Returns: a morphism in Hom$(initial_object1, initial_object2)$

The argument is a category C and an initial object InitialObject(C) twice (for compatibility with other functorials). The output is the unique morphism $initial_object1 \to initial_object2$.

6.6 Direct Sum

For an integer $n \geq 1$ and a given list $D = (S_1, \dots, S_n)$ in an Ab-category, a direct sum consists of five parts:

- an object S,
- a list of morphisms $\pi = (\pi_i : S \to S_i)_{i=1\dots n}$,
- a list of morphisms $\iota = (\iota_i : S_i \to S)_{i=1\dots n}$,
- a dependent function u_{in} mapping every list $\tau = (\tau_i : T \to S_i)_{i=1\dots n}$ to a morphism $u_{\mathrm{in}}(\tau) : T \to S$ such that $\pi_i \circ u_{\mathrm{in}}(\tau) \sim_{T,S_i} \tau_i$ for all $i = 1, \dots, n$.
- a dependent function u_{out} mapping every list $\tau = (\tau_i : S_i \to T)_{i=1\dots n}$ to a morphism $u_{\mathrm{out}}(\tau) : S \to T$ such that $u_{\mathrm{out}}(\tau) \circ \iota_i \sim_{S_i,T} \tau_i$ for all $i = 1, \dots, n$,

such that

- $\sum_{i=1}^{n} \iota_i \circ \pi_i \sim_{S,S} \mathrm{id}_S$,
- $\pi_j \circ \iota_i \sim_{S_i,S_j} \delta_{i,j}$,

where $\delta_{i,j} \in \mathrm{Hom}(S_i, S_j)$ is the identity if $i = j$, and 0 otherwise. The 5-tuple $(S, \pi, \iota, u_{\mathrm{in}}, u_{\mathrm{out}})$ is called a *direct sum* of D. We denote the object S of such a 5-tuple by $\bigoplus_{i=1}^{n} S_i$. We say that the morphisms $u_{\mathrm{in}}(\tau), u_{\mathrm{out}}(\tau)$ are induced by the *universal property of the direct sum*.
DirectSum is a functorial operation. This means: For $(\mu_i : S_i \to S'_i)_{i=1\dots n}$, we obtain a morphism $\bigoplus_{i=1}^{n} S_i \to \bigoplus_{i=1}^{n} S'_i$.

$$\begin{tikzcd}
 & T \arrow[dl, bend right, "\tau_1"'] \arrow[d, dashed, "\exists u_{in}(\tau)"'] \arrow[dr, bend left, "\tau_2"] & \\
S_1 \arrow[r, bend right, "\iota_1"'] & S \arrow[l, bend right, "\pi_1"'] \arrow[r, bend left, "\pi_2"] & S_2 \arrow[l, bend left, "\iota_2"]
\end{tikzcd}$$

$$\begin{tikzcd}
 & T & \\
S_1 \arrow[ur, bend left, "\tau_1"] \arrow[r, bend right, "\iota_1"'] & S \arrow[u, dashed, "\exists u_{out}(\tau)"] \arrow[l, bend right, "\pi_1"'] \arrow[r, bend left, "\pi_2"] & S_2 \arrow[ul, bend right, "\tau_2"'] \arrow[l, bend left, "\iota_2"]
\end{tikzcd}$$

6.6.1 DirectSum

▷ `DirectSum(arg)` (function)

Returns: an object

This is a convenience method. There are two different ways to use this method:

- The argument is a list of objects $D = (S_1, \dots, S_n)$.
- The arguments are objects $S_1, \dots, S_n$.

The output is the direct sum $\bigoplus_{i=1}^{n} S_i$.

6.6.2 DirectSumOp (for IsList)

▷ `DirectSumOp(D)` (operation)

Returns: an object

The argument is a list of objects $D = (S_1, \dots, S_n)$. The output is the direct sum $\bigoplus_{i=1}^n S_i$.

6.6.3 ProjectionInFactorOfDirectSum (for IsList, IsInt)

▷ `ProjectionInFactorOfDirectSum(D, k)` (operation)

Returns: a morphism in $\mathrm{Hom}(\bigoplus_{i=1}^n S_i, S_k)$

The arguments are a list of objects $D = (S_1, \dots, S_n)$ and an integer k. The output is the k-th projection $\pi_k : \bigoplus_{i=1}^n S_i \rightarrow S_k$.

6.6.4 ProjectionInFactorOfDirectSumWithGivenDirectSum (for IsList, IsInt, IsCapCategoryObject)

▷ `ProjectionInFactorOfDirectSumWithGivenDirectSum(D, k, S)` (operation)

Returns: a morphism in $\mathrm{Hom}(S, S_k)$

The arguments are a list of objects $D = (S_1, \dots, S_n)$, an integer k, and an object $S = \bigoplus_{i=1}^n S_i$. The output is the k-th projection $\pi_k : S \rightarrow S_k$.

6.6.5 InjectionOfCofactorOfDirectSum (for IsList, IsInt)

▷ `InjectionOfCofactorOfDirectSum(D, k)` (operation)

Returns: a morphism in $\mathrm{Hom}(S_k, \bigoplus_{i=1}^n S_i)$

The arguments are a list of objects $D = (S_1, \dots, S_n)$ and an integer k. The output is the k-th injection $\iota_k : S_k \rightarrow \bigoplus_{i=1}^n S_i$.

6.6.6 InjectionOfCofactorOfDirectSumWithGivenDirectSum (for IsList, IsInt, IsCapCategoryObject)

▷ `InjectionOfCofactorOfDirectSumWithGivenDirectSum(D, k, S)` (operation)

Returns: a morphism in $\mathrm{Hom}(S_k, S)$

The arguments are a list of objects $D = (S_1, \dots, S_n)$, an integer k, and an object $S = \bigoplus_{i=1}^n S_i$. The output is the k-th injection $\iota_k : S_k \rightarrow S$.

6.6.7 UniversalMorphismIntoDirectSum (for IsList, IsCapCategoryObject, IsList)

▷ `UniversalMorphismIntoDirectSum(D, T, tau)` (operation)

Returns: a morphism in $\mathrm{Hom}(T, \bigoplus_{i=1}^n S_i)$

The arguments are a list of objects $D = (S_1, \dots, S_n)$, a test object T, and a list of morphisms $\tau = (\tau_i : T \rightarrow S_i)_{i=1\dots n}$. For convenience, the diagram `D` and/or the test object `T` can be omitted and are automatically derived from `tau` in that case. The output is the morphism $u_{\mathrm{in}}(\tau) : T \rightarrow \bigoplus_{i=1}^n S_i$ given by the universal property of the direct sum.

6.6.8 UniversalMorphismIntoDirectSumWithGivenDirectSum (for IsList, IsCapCategoryObject, IsList, IsCapCategoryObject)

▷ `UniversalMorphismIntoDirectSumWithGivenDirectSum(D, T, tau, S)` (operation)

Returns: a morphism in $\mathrm{Hom}(T,S)$

The arguments are a list of objects $D=(S_1,\dots,S_n)$, a test object T, a list of morphisms $\tau=(\tau_i : T\rightarrow S_i)_{i=1\dots n}$, and an object $S=\bigoplus_{i=1}^n S_i$. For convenience, the test object `T` can be omitted and is automatically derived from `tau` in that case. The output is the morphism $u_{\mathrm{in}}(\tau): T\rightarrow S$ given by the universal property of the direct sum.

6.6.9 UniversalMorphismFromDirectSum (for IsList, IsCapCategoryObject, IsList)

▷ `UniversalMorphismFromDirectSum(D, T, tau)` (operation)

Returns: a morphism in $\mathrm{Hom}(\bigoplus_{i=1}^n S_i, T)$

The arguments are a list of objects $D=(S_1,\dots,S_n)$, a test object T, and a list of morphisms $\tau=(\tau_i : S_i\rightarrow T)_{i=1\dots n}$. For convenience, the diagram `D` and/or the test object `T` can be omitted and are automatically derived from `tau` in that case. The output is the morphism $u_{\mathrm{out}}(\tau): \bigoplus_{i=1}^n S_i\rightarrow T$ given by the universal property of the direct sum.

6.6.10 UniversalMorphismFromDirectSumWithGivenDirectSum (for IsList, IsCapCategoryObject, IsList, IsCapCategoryObject)

▷ `UniversalMorphismFromDirectSumWithGivenDirectSum(D, T, tau, S)` (operation)

Returns: a morphism in $\mathrm{Hom}(S,T)$

The arguments are a list of objects $D=(S_1,\dots,S_n)$, a test object T, a list of morphisms $\tau=(\tau_i : S_i\rightarrow T)_{i=1\dots n}$, and an object $S=\bigoplus_{i=1}^n S_i$. For convenience, the test object `T` can be omitted and is automatically derived from `tau` in that case. The output is the morphism $u_{\mathrm{out}}(\tau): S\rightarrow T$ given by the universal property of the direct sum.

6.6.11 IsomorphismFromDirectSumToDirectProduct (for IsList)

▷ `IsomorphismFromDirectSumToDirectProduct(D)` (operation)

Returns: a morphism in $\mathrm{Hom}(\bigoplus_{i=1}^n S_i, \prod_{i=1}^n S_i)$

The argument is a list of objects $D=(S_1,\dots,S_n)$. The output is the canonical isomorphism $\bigoplus_{i=1}^n S_i\rightarrow\prod_{i=1}^n S_i$.

6.6.12 IsomorphismFromDirectProductToDirectSum (for IsList)

▷ `IsomorphismFromDirectProductToDirectSum(D)` (operation)

Returns: a morphism in $\mathrm{Hom}(\prod_{i=1}^n S_i, \bigoplus_{i=1}^n S_i)$

The argument is a list of objects $D=(S_1,\dots,S_n)$. The output is the canonical isomorphism $\prod_{i=1}^n S_i\rightarrow\bigoplus_{i=1}^n S_i$.

6.6.13 IsomorphismFromDirectSumToCoproduct (for IsList)

▷ `IsomorphismFromDirectSumToCoproduct(D)` (operation)

Returns: a morphism in $\mathrm{Hom}(\bigoplus_{i=1}^n S_i, \bigsqcup_{i=1}^n S_i)$

The argument is a list of objects $D = (S_1, \dots, S_n)$. The output is the canonical isomorphism $\bigoplus_{i=1}^n S_i \to \bigsqcup_{i=1}^n S_i$.

6.6.14 IsomorphismFromCoproductToDirectSum (for IsList)

▷ `IsomorphismFromCoproductToDirectSum(`*D*`)` (operation)

Returns: a morphism in $\mathrm{Hom}(\bigsqcup_{i=1}^n S_i, \bigoplus_{i=1}^n S_i)$

The argument is a list of objects $D = (S_1, \dots, S_n)$. The output is the canonical isomorphism $\bigsqcup_{i=1}^n S_i \to \bigoplus_{i=1}^n S_i$.

6.6.15 MorphismBetweenDirectSums (for IsList, IsList, IsList)

▷ `MorphismBetweenDirectSums(`*diagram_S, M, diagram_T*`)` (operation)

Returns: a morphism in $\mathrm{Hom}(\bigoplus_{i=1}^m A_i, \bigoplus_{j=1}^n B_j)$

The arguments are given as follows:

- *diagram_S* is a list of objects $(A_i)_{i=1\dots m}$,
- *diagram_T* is a list of objects $(B_j)_{j=1\dots n}$,
- *M* is a list of lists of morphisms $((\phi_{i,j} : A_i \to B_j)_{j=1\dots n})_{i=1\dots m}$.

The output is the morphism $\bigoplus_{i=1}^m A_i \to \bigoplus_{j=1}^n B_j$ defined by the matrix M.

6.6.16 MorphismBetweenDirectSums (for IsList)

▷ `MorphismBetweenDirectSums(`*M*`)` (operation)

Returns: a morphism in $\mathrm{Hom}(\bigoplus_{i=1}^m A_i, \bigoplus_{j=1}^n B_j)$

This is a convenience method. The argument $M = ((\phi_{i,j} : A_i \to B_j)_{j=1\dots n})_{i=1\dots m}$ is a (non-empty) list of (non-empty) lists of morphisms. The output is the morphism $\bigoplus_{i=1}^m A_i \to \bigoplus_{j=1}^n B_j$ defined by the matrix M.

6.6.17 MorphismBetweenDirectSumsWithGivenDirectSums (for IsCapCategoryObject, IsList, IsList, IsList, IsCapCategoryObject)

▷ `MorphismBetweenDirectSumsWithGivenDirectSums(`*S, diagram_S, M, diagram_T, T*`)` (operation)

Returns: a morphism in $\mathrm{Hom}(\bigoplus_{i=1}^m A_i, \bigoplus_{j=1}^n B_j)$

The arguments are given as follows:

- *diagram_S* is a list of objects $(A_i)_{i=1\dots m}$,
- *diagram_T* is a list of objects $(B_j)_{j=1\dots n}$,
- *S* is the direct sum $\bigoplus_{i=1}^m A_i$,
- *T* is the direct sum $\bigoplus_{j=1}^n B_j$,
- *M* is a list of lists of morphisms $((\phi_{i,j} : A_i \to B_j)_{j=1\dots n})_{i=1\dots m}$.

The output is the morphism $\bigoplus_{i=1}^m A_i \to \bigoplus_{j=1}^n B_j$ defined by the matrix M.

6.6.18 ComponentOfMorphismIntoDirectSum (for IsCapCategoryMorphism, IsList, IsInt)

▷ `ComponentOfMorphismIntoDirectSum(alpha, D, k)` (operation)

Returns: a morphism in $\mathrm{Hom}(A, S_k)$

The arguments are a morphism $\alpha : A \to S$, a list $D = (S_1, \dots, S_n)$ of objects with $S = \bigoplus_{j=1}^{n} S_j$, and an integer k. The output is the component morphism $A \to S_k$.

6.6.19 ComponentOfMorphismFromDirectSum (for IsCapCategoryMorphism, IsList, IsInt)

▷ `ComponentOfMorphismFromDirectSum(alpha, D, k)` (operation)

Returns: a morphism in $\mathrm{Hom}(S_k, A)$

The arguments are a morphism $\alpha : S \to A$, a list $D = (S_1, \dots, S_n)$ of objects with $S = \bigoplus_{j=1}^{n} S_j$, and an integer k. The output is the component morphism $S_k \to A$.

6.6.20 DirectSumFunctorial (for IsList, IsList, IsList)

▷ `DirectSumFunctorial(source_diagram, L, range_diagram)` (operation)

Returns: a morphism in $\mathrm{Hom}(\bigoplus_{i=1}^{n} S_i, \bigoplus_{i=1}^{n} S_i')$

The arguments are a list of objects $(S_i)_{i=1\dots n}$, a list of morphisms $L = (\mu_1 : S_1 \to S_1', \dots, \mu_n : S_n \to S_n')$, and a list of objects $(S_i')_{i=1\dots n}$. For convenience, *source_diagram* and *range_diagram* can be omitted and are automatically derived from `L` in that case. The output is a morphism $\bigoplus_{i=1}^{n} S_i \to \bigoplus_{i=1}^{n} S_i'$ given by the functoriality of the direct sum.

6.6.21 DirectSumFunctorialWithGivenDirectSums (for IsCapCategoryObject, IsList, IsList, IsList, IsCapCategoryObject)

▷ `DirectSumFunctorialWithGivenDirectSums(d_1, source_diagram, L, range_diagram, d_2)` (operation)

Returns: a morphism in $\mathrm{Hom}(d_1, d_2)$

The arguments are an object $d_1 = \bigoplus_{i=1}^{n} S_i$, a list of objects $(S_i)_{i=1\dots n}$, a list of morphisms $L = (\mu_1 : S_1 \to S_1', \dots, \mu_n : S_n \to S_n')$, a list of objects $(S_i')_{i=1\dots n}$, and an object $d_2 = \bigoplus_{i=1}^{n} S_i'$. For convenience, *source_diagram* and *range_diagram* can be omitted and are automatically derived from `L` in that case. The output is a morphism $d_1 \to d_2$ given by the functoriality of the direct sum.

6.7 Coproduct

For an integer $n \geq 1$ and a given list of objects $D = (I_1, \dots, I_n)$, a coproduct of D consists of three parts:

- an object I,
- a list of morphisms $\iota = (\iota_i : I_i \to I)_{i=1\dots n}$
- a dependent function u mapping each list of morphisms $\tau = (\tau_i : I_i \to T)$ to a morphism $u(\tau) : I \to T$ such that $u(\tau) \circ \iota_i \sim_{I_i,T} \tau_i$ for all $i = 1, \dots, n$.

The triple (I, ι, u) is called a *coproduct* of D if the morphisms $u(\tau)$ are uniquely determined up to congruence of morphisms. We denote the object I of such a triple by $\bigsqcup_{i=1}^{n} I_i$. We say that the morphism $u(\tau)$ is induced by the *universal property of the coproduct*.
Coproduct is a functorial operation. This means: For $(\mu_i : I_i \to I_i')_{i=1\dots n}$, we obtain a morphism $\bigsqcup_{i=1}^{n} I_i \to \bigsqcup_{i=1}^{n} I_i'$.

$$\begin{tikzcd} & T & \\ I_1 \arrow[r, "\iota_1"'] \arrow[ur, bend left, "\tau_1"] & I \arrow[u, dashed, "\exists ! u(\tau)"'] & I_2 \arrow[l, "\iota_2"] \arrow[ul, bend right, "\tau_2"'] \end{tikzcd}$$

6.7.1 Coproduct (for IsList)

▷ `Coproduct(D)` (operation)

Returns: an object

The argument is a list of objects $D = (I_1, \dots, I_n)$. The output is the coproduct $\bigsqcup_{i=1}^{n} I_i$.

6.7.2 Coproduct (for IsCapCategoryObject, IsCapCategoryObject)

▷ `Coproduct(I1, I2)` (operation)

Returns: an object

This is a convenience method. The arguments are two objects I_1, I_2. The output is the coproduct $I_1 \bigsqcup I_2$.

6.7.3 Coproduct (for IsCapCategoryObject, IsCapCategoryObject, IsCapCategoryObject)

▷ `Coproduct(I1, I2)` (operation)

Returns: an object

This is a convenience method. The arguments are three objects I_1, I_2, I_3. The output is the coproduct $I_1 \bigsqcup I_2 \bigsqcup I_3$.

6.7.4 InjectionOfCofactorOfCoproduct (for IsList, IsInt)

▷ `InjectionOfCofactorOfCoproduct(D, k)` (operation)

Returns: a morphism in $\mathrm{Hom}(I_k, \bigsqcup_{i=1}^{n} I_i)$

The arguments are a list of objects $D = (I_1, \dots, I_n)$ and an integer k. The output is the k-th injection $\iota_k : I_k \to \bigsqcup_{i=1}^{n} I_i$.

6.7.5 InjectionOfCofactorOfCoproductWithGivenCoproduct (for IsList, IsInt, IsCapCategoryObject)

▷ `InjectionOfCofactorOfCoproductWithGivenCoproduct(D, k, I)` (operation)

Returns: a morphism in $\mathrm{Hom}(I_k, I)$

The arguments are a list of objects $D = (I_1, \dots, I_n)$, an integer k, and an object $I = \bigsqcup_{i=1}^{n} I_i$. The output is the k-th injection $\iota_k : I_k \to I$.

6.7.6 UniversalMorphismFromCoproduct (for IsList, IsCapCategoryObject, IsList)

▷ `UniversalMorphismFromCoproduct(D, T, tau)` (operation)

Returns: a morphism in $\mathrm{Hom}(\bigsqcup_{i=1}^{n} I_i, T)$

The arguments are a list of objects $D = (I_1, \dots, I_n)$, a test object T, and a list of morphisms $\tau = (\tau_i : I_i \rightarrow T)$. For convenience, the diagram `D` and/or the test object `T` can be omitted and are automatically derived from `tau` in that case. The output is the morphism $u(\tau) : \bigsqcup_{i=1}^{n} I_i \rightarrow T$ given by the universal property of the coproduct.

6.7.7 UniversalMorphismFromCoproductWithGivenCoproduct (for IsList, IsCapCategoryObject, IsList, IsCapCategoryObject)

▷ `UniversalMorphismFromCoproductWithGivenCoproduct(D, T, tau, I)` (operation)

Returns: a morphism in $\mathrm{Hom}(I, T)$

The arguments are a list of objects $D = (I_1, \dots, I_n)$, a test object T, a list of morphisms $\tau = (\tau_i : I_i \rightarrow T)$, and an object $I = \bigsqcup_{i=1}^{n} I_i$. For convenience, the test object `T` can be omitted and is automatically derived from `tau` in that case. The output is the morphism $u(\tau) : I \rightarrow T$ given by the universal property of the coproduct.

6.7.8 CoproductFunctorial (for IsList, IsList, IsList)

▷ `CoproductFunctorial(source_diagram, L, range_diagram)` (operation)

Returns: a morphism in $\mathrm{Hom}(\bigsqcup_{i=1}^{n} I_i, \bigsqcup_{i=1}^{n} I_i')$

The arguments are a list of objects $(I_i)_{i=1\dots n}$, a list $L = (\mu_1 : I_1 \rightarrow I_1', \dots, \mu_n : I_n \rightarrow I_n')$, and a list of objects $(I_i')_{i=1\dots n}$. For convenience, `source_diagram` and `range_diagram` can be omitted and are automatically derived from `L` in that case. The output is a morphism $\bigsqcup_{i=1}^{n} I_i \rightarrow \bigsqcup_{i=1}^{n} I_i'$ given by the functoriality of the coproduct.

6.7.9 CoproductFunctorialWithGivenCoproducts (for IsCapCategoryObject, IsList, IsList, IsList, IsCapCategoryObject)

▷ `CoproductFunctorialWithGivenCoproducts(s, source_diagram, L, range_diagram, r)` (operation)

Returns: a morphism in $\mathrm{Hom}(s, r)$

The arguments are an object $s = \bigsqcup_{i=1}^{n} I_i$, a list of objects $(I_i)_{i=1\dots n}$, a list $L = (\mu_1 : I_1 \rightarrow I_1', \dots, \mu_n : I_n \rightarrow I_n')$, a list of objects $(I_i')_{i=1\dots n}$, and an object $r = \bigsqcup_{i=1}^{n} I_i'$. For convenience, `source_diagram` and `range_diagram` can be omitted and are automatically derived from `L` in that case. The output is a morphism $\bigsqcup_{i=1}^{n} I_i \rightarrow \bigsqcup_{i=1}^{n} I_i'$ given by the functoriality of the coproduct.

6.7.10 ComponentOfMorphismFromCoproduct (for IsCapCategoryMorphism, IsList, IsInt)

▷ `ComponentOfMorphismFromCoproduct(alpha, D, k)` (operation)

Returns: a morphism in $\mathrm{Hom}(I_k, A)$

The arguments are a morphism $\alpha : I \rightarrow A$, a list $D = (I_1, \dots, I_n)$ of objects with $I = \bigsqcup_{j=1}^{n} I_j$, and an integer k. The output is the component morphism $I_k \rightarrow A$.

6.8 Direct Product

For an integer $n \geq 1$ and a given list of objects $D = (P_1, \ldots, P_n)$, a direct product of D consists of three parts:

- an object P,
- a list of morphisms $\pi = (\pi_i : P \to P_i)_{i=1\ldots n}$
- a dependent function u mapping each list of morphisms $\tau = (\tau_i : T \to P_i)_{i=1,\ldots,n}$ to a morphism $u(\tau) : T \to P$ such that $\pi_i \circ u(\tau) \sim_{T,P_i} \tau_i$ for all $i = 1, \ldots, n$.

The triple (P, π, u) is called a *direct product* of D if the morphisms $u(\tau)$ are uniquely determined up to congruence of morphisms. We denote the object P of such a triple by $\prod_{i=1}^n P_i$. We say that the morphism $u(\tau)$ is induced by the *universal property of the direct product*.
DirectProduct is a functorial operation. This means: For $(\mu_i : P_i \to P_i')_{i=1\ldots n}$, we obtain a morphism $\prod_{i=1}^n P_i \to \prod_{i=1}^n P_i'$.

$$
\begin{tikzcd}
 & T \arrow[dl, bend right, "\tau_1"'] \arrow[dr, bend left, "\tau_2"] \arrow[d, dashed, "\exists ! u(\tau)"'] & \\
P_1 & P \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & P_2
\end{tikzcd}
$$

6.8.1 DirectProduct

▷ `DirectProduct(arg)` (function)

Returns: an object

This is a convenience method. There are two different ways to use this method:

- The argument is a list of objects $D = (P_1, \ldots, P_n)$.
- The arguments are objects $P_1, \ldots, P_n$.

The output is the direct product $\prod_{i=1}^n P_i$.

6.8.2 DirectProductOp (for IsList)

▷ `DirectProductOp(D)` (operation)

Returns: an object

The argument is a list of objects $D = (P_1, \ldots, P_n)$. The output is the direct product $\prod_{i=1}^n P_i$.

6.8.3 ProjectionInFactorOfDirectProduct (for IsList, IsInt)

▷ `ProjectionInFactorOfDirectProduct(D, k)` (operation)

Returns: a morphism in $\mathrm{Hom}(\prod_{i=1}^n P_i, P_k)$

The arguments are a list of objects $D = (P_1, \ldots, P_n)$ and an integer k. The output is the k-th projection $\pi_k : \prod_{i=1}^n P_i \to P_k$.

6.8.4 ProjectionInFactorOfDirectProductWithGivenDirectProduct (for IsList, IsInt, IsCapCategoryObject)

▷ `ProjectionInFactorOfDirectProductWithGivenDirectProduct(D, k, P)` (operation)

Returns: a morphism in $\mathrm{Hom}(P, P_k)$

The arguments are a list of objects $D = (P_1, \dots, P_n)$, an integer k, and an object $P = \prod_{i=1}^n P_i$. The output is the k-th projection $\pi_k : P \to P_k$.

6.8.5 UniversalMorphismIntoDirectProduct (for IsList, IsCapCategoryObject, IsList)

▷ `UniversalMorphismIntoDirectProduct(D, T, tau)` (operation)

Returns: a morphism in $\mathrm{Hom}(T, \prod_{i=1}^n P_i)$

The arguments are a list of objects $D = (P_1, \dots, P_n)$, a test object T, and a list of morphisms $\tau = (\tau_i : T \to P_i)_{i=1,\dots,n}$. For convenience, the diagram `D` and/or the test object `T` can be omitted and are automatically derived from `tau` in that case. The output is the morphism $u(\tau) : T \to \prod_{i=1}^n P_i$ given by the universal property of the direct product.

6.8.6 UniversalMorphismIntoDirectProductWithGivenDirectProduct (for IsList, IsCapCategoryObject, IsList, IsCapCategoryObject)

▷ `UniversalMorphismIntoDirectProductWithGivenDirectProduct(D, T, tau, P)` (operation)

Returns: a morphism in $\mathrm{Hom}(T, \prod_{i=1}^n P_i)$

The arguments are a list of objects $D = (P_1, \dots, P_n)$, a test object T, a list of morphisms $\tau = (\tau_i : T \to P_i)_{i=1,\dots,n}$, and an object $P = \prod_{i=1}^n P_i$. For convenience, the test object `T` can be omitted and is automatically derived from `tau` in that case. The output is the morphism $u(\tau) : T \to \prod_{i=1}^n P_i$ given by the universal property of the direct product.

6.8.7 DirectProductFunctorial (for IsList, IsList, IsList)

▷ `DirectProductFunctorial(source_diagram, L, range_diagram)` (operation)

Returns: a morphism in $\mathrm{Hom}(\prod_{i=1}^n P_i, \prod_{i=1}^n P_i')$

The arguments are a list of objects $(P_i)_{i=1\dots n}$, a list of morphisms $L = (\mu_i : P_i \to P_i')_{i=1\dots n}$, and a list of objects $(P_i')_{i=1\dots n}$. For convenience, `source_diagram` and `range_diagram` can be omitted and are automatically derived from `L` in that case. The output is a morphism $\prod_{i=1}^n P_i \to \prod_{i=1}^n P_i'$ given by the functoriality of the direct product.

6.8.8 DirectProductFunctorialWithGivenDirectProducts (for IsCapCategoryObject, IsList, IsList, IsList, IsCapCategoryObject)

▷ `DirectProductFunctorialWithGivenDirectProducts(s, source_diagram, L, range_diagram, r)` (operation)

Returns: a morphism in $\mathrm{Hom}(s, r)$

The arguments are an object $s = \prod_{i=1}^n P_i$, a list of objects $(P_i)_{i=1\dots n}$, a list of morphisms $L = (\mu_i : P_i \to P_i')_{i=1\dots n}$, a list of objects $(P_i')_{i=1\dots n}$, and an object $r = \prod_{i=1}^n P_i'$. For convenience, `source_diagram` and `range_diagram` can be omitted and are automatically derived from `L` in that case. The output is a morphism $\prod_{i=1}^n P_i \to \prod_{i=1}^n P_i'$ given by the functoriality of the direct product.

6.8.9 ComponentOfMorphismIntoDirectProduct (for IsCapCategoryMorphism, IsList, IsInt)

▷ `ComponentOfMorphismIntoDirectProduct(alpha, D, k)` (operation)

Returns: a morphism in $\mathrm{Hom}(A, P_k)$

The arguments are a morphism $\alpha : A \to P$, a list $D = (P_1, \dots, P_n)$ of objects with $P = \prod_{j=1}^{n} P_j$, and an integer k. The output is the component morphism $A \to P_k$.

6.9 Equalizer

For an integer $n \geq 0$ and a given list of morphisms $D = (\beta_i : A \to B)_{i=1\dots n}$, an equalizer of D consists of three parts:

- an object E,
- a morphism $\iota : E \to A$ such that $\beta_i \circ \iota \sim_{E,B} \beta_j \circ \iota$ for all pairs i, j.
- a dependent function u mapping each morphism $\tau = (\tau : T \to A)$ such that $\beta_i \circ \tau \sim_{T,B} \beta_j \circ \tau$ for all pairs i, j to a morphism $u(\tau) : T \to E$ such that $\iota \circ u(\tau) \sim_{T,A} \tau$.

The triple (E, ι, u) is called an *equalizer* of D if the morphisms $u(\tau)$ are uniquely determined up to congruence of morphisms. We denote the object E of such a triple by $\mathrm{Equalizer}(D)$. We say that the morphism $u(\tau)$ is induced by the *universal property of the equalizer*.
Equalizer is a functorial operation. This means: For a second diagram $D' = (\beta'_i : A' \to B')_{i=1\dots n}$ and a natural morphism between equalizer diagrams (i.e., a collection of morphisms $\mu : A \to A'$ and $\beta : B \to B'$ such that $\beta'_i \circ \mu \sim_{A,B'} \beta \circ \beta_i$ for $i = 1, \dots, n$) we obtain a morphism $\mathrm{Equalizer}(D) \to \mathrm{Equalizer}(D')$.

$$\begin{tikzcd}
T \arrow[d, dashed, "\exists ! u(\tau)"'] \arrow[dr, "\tau"] \arrow[drr, dotted, bend left, "\beta_2 \circ \tau \sim_{T,B} \beta_1 \circ \tau"] & & \\
E \arrow[r, "\iota"] \arrow[rr, dotted, bend right, "\beta_2 \circ \iota \sim_{E,B} \beta_1 \circ \iota"'] & A \arrow[r, bend left, "\beta_1"] \arrow[r, bend right, "\beta_2"'] & B
\end{tikzcd}$$

6.9.1 Equalizer

▷ `Equalizer(arg)` (function)

Returns: an object

This is a convenience method. There are three different ways to use this method:

- The arguments are an object A and a list of morphisms $D = (\beta_i : A \to B)_{i=1\dots n}$.
- The argument is a nonempty list of morphisms $D = (\beta_i : A \to B)_{i=1\dots n}$.
- The arguments are morphisms $\beta_1 : A \to B, \dots, \beta_n : A \to B$ with $n \geq 1$.

The output is the equalizer $\mathrm{Equalizer}(D)$.

6.9.2 EqualizerOp (for IsCapCategoryObject, IsList)

▷ `EqualizerOp(A, D)` (operation)

Returns: an object

The arguments are an object A and list of morphisms $D = (\beta_i : A \to B)_{i=1\dots n}$. For convenience, the object A can be omitted and is automatically derived from D, which has to be nonempty. The output is the equalizer $\mathrm{Equalizer}(D)$.

6.9.3 EmbeddingOfEqualizer (for IsCapCategoryObject, IsList)

▷ `EmbeddingOfEqualizer(A, D)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{Equalizer}(D), A)$

The arguments are an object A and a list of morphisms $D = (\beta_i : A \to B)_{i=1\dots n}$. For convenience, the object A can be omitted and is automatically derived from D, which has to be nonempty. The output is the equalizer embedding $\iota : \mathrm{Equalizer}(D) \to A$.

6.9.4 EmbeddingOfEqualizerWithGivenEqualizer (for IsCapCategoryObject, IsList, IsCapCategoryObject)

▷ `EmbeddingOfEqualizerWithGivenEqualizer(A, D, E)` (operation)

Returns: a morphism in $\mathrm{Hom}(E, A)$

The arguments are an object A, a list of morphisms $D = (\beta_i : A \to B)_{i=1\dots n}$, and an object $E = \mathrm{Equalizer}(D)$. For convenience, the object A can be omitted and is automatically derived from D, which has to be nonempty. The output is the equalizer embedding $\iota : E \to A$.

6.9.5 MorphismFromEqualizerToSink (for IsCapCategoryObject, IsList)

▷ `MorphismFromEqualizerToSink(A, D)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{Equalizer}(D), B)$

The arguments are an object A and a list of morphisms $D = (\beta_i : A \to B)_{i=1\dots n}$. For convenience, the object A can be omitted and is automatically derived from D, which has to be nonempty. The output is the composition $\mu : \mathrm{Equalizer}(D) \to B$ of the embedding $\iota : \mathrm{Equalizer}(D) \to A$ and β_1.

6.9.6 MorphismFromEqualizerToSinkWithGivenEqualizer (for IsCapCategoryObject, IsList, IsCapCategoryObject)

▷ `MorphismFromEqualizerToSinkWithGivenEqualizer(A, D, E)` (operation)

Returns: a morphism in $\mathrm{Hom}(E, B)$

The arguments are an object A, a list of morphisms $D = (\beta_i : A \to B)_{i=1\dots n}$ and an object $E = \mathrm{Equalizer}(D)$. For convenience, the object A can be omitted and is automatically derived from D, which has to be nonempty. The output is the composition $\mu : E \to B$ of the embedding $\iota : E \to A$ and β_1.

6.9.7 UniversalMorphismIntoEqualizer (for IsCapCategoryObject, IsList, IsCapCategoryObject, IsCapCategoryMorphism)

▷ `UniversalMorphismIntoEqualizer(A, D, T, tau)` (operation)

Returns: a morphism in $\mathrm{Hom}(T, \mathrm{Equalizer}(D))$

The arguments are an object A, a list of morphisms $D = (\beta_i : A \rightarrow B)_{i=1\dots n}$, a test object T, and a morphism $\tau : T \rightarrow A$ such that $\beta_i \circ \tau \sim_{T,B} \beta_j \circ \tau$ for all pairs i, j. For convenience, the object A can be omitted and is automatically derived from D, which has to be nonempty. For convenience, the test object `T` can be omitted and is automatically derived from `tau` in that case. The output is the morphism $u(\tau) : T \rightarrow \mathrm{Equalizer}(D)$ given by the universal property of the equalizer.

6.9.8 UniversalMorphismIntoEqualizerWithGivenEqualizer (for IsCapCategoryObject, IsList, IsCapCategoryObject, IsCapCategoryMorphism, IsCapCategoryObject)

▷ `UniversalMorphismIntoEqualizerWithGivenEqualizer(A, D, T, tau, E)` (operation)

Returns: a morphism in $\mathrm{Hom}(T,E)$

The arguments are an object A, a list of morphisms $D = (\beta_i : A \rightarrow B)_{i=1\dots n}$, a test object T, a morphism $\tau : T \rightarrow A)$ such that $\beta_i \circ \tau \sim_{T,B} \beta_j \circ \tau$ for all pairs i, j, and an object $E = \mathrm{Equalizer}(D)$. For convenience, the object A can be omitted and is automatically derived from D, which has to be nonempty. For convenience, the test object `T` can be omitted and is automatically derived from `tau` in that case. The output is the morphism $u(\tau) : T \rightarrow E$ given by the universal property of the equalizer.

6.9.9 EqualizerFunctorial (for IsList, IsCapCategoryMorphism, IsList)

▷ `EqualizerFunctorial(Ls, mu, Lr)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{Equalizer}((\beta_i)_{i=1\dots n}), \mathrm{Equalizer}((\beta_i')_{i=1\dots n}))$

The arguments are a list of morphisms $L_s = (\beta_i : A \rightarrow B)_{i=1\dots n}$, a morphism $\mu : A \rightarrow A'$, and a list of morphisms $L_r = (\beta_i' : A' \rightarrow B')_{i=1\dots n}$ such that there exists a morphism $\beta : B \rightarrow B'$ such that $\beta_i' \circ \mu \sim_{A,B'} \beta \circ \beta_i$ for $i = 1, \dots, n$. The output is the morphism $\mathrm{Equalizer}((\beta_i)_{i=1\dots n}) \rightarrow \mathrm{Equalizer}((\beta_i')_{i=1\dots n})$ given by the functorality of the equalizer.

6.9.10 EqualizerFunctorialWithGivenEqualizers (for IsCapCategoryObject, IsList, IsCapCategoryMorphism, IsList, IsCapCategoryObject)

▷ `EqualizerFunctorialWithGivenEqualizers(s, Ls, mu, Lr, r)` (operation)

Returns: a morphism in $\mathrm{Hom}(s,r)$

The arguments are an object $s = \mathrm{Equalizer}((\beta_i)_{i=1\dots n})$, a list of morphisms $L_s = (\beta_i : A \rightarrow B)_{i=1\dots n}$, a morphism $\mu : A \rightarrow A'$, and a list of morphisms $L_r = (\beta_i' : A' \rightarrow B')_{i=1\dots n}$ such that there exists a morphism $\beta : B \rightarrow B'$ such that $\beta_i' \circ \mu \sim_{A,B'} \beta \circ \beta_i$ for $i = 1, \dots, n$, and an object $r = \mathrm{Equalizer}((\beta_i')_{i=1\dots n})$. The output is the morphism $s \rightarrow r$ given by the functorality of the equalizer.

6.9.11 JointPairwiseDifferencesOfMorphismsIntoDirectProduct (for IsCapCategoryObject, IsList)

▷ `JointPairwiseDifferencesOfMorphismsIntoDirectProduct(A, D)` (operation)

Returns: a morphism in $\mathrm{Hom}(A, \prod_{i=1}^{n-1} B)$

The arguments are an object A and a list of morphisms $D = (\beta_i : A \rightarrow B)_{i=1\dots n}$. The output is a morphism $A \rightarrow \prod_{i=1}^{n-1} B$ such that its kernel equalizes the β_i.

6.9.12 IsomorphismFromEqualizerToKernelOfJointPairwiseDifferencesOfMorphismsIntoDirectPro (for IsCapCategoryObject, IsList)

▷ `IsomorphismFromEqualizerToKernelOfJointPairwiseDifferencesOfMorphismsIntoDirectProduct(A, D)` (operation)

Returns: a morphism in Hom(Equalizer(D), Δ)

The arguments are an object A and a list of morphisms $D = (\beta_i : A \rightarrow B)_{i=1\dots n}$. The output is a morphism Equalizer(D) $\rightarrow \Delta$, where Δ denotes the kernel object equalizing the morphisms β_i.

6.9.13 IsomorphismFromKernelOfJointPairwiseDifferencesOfMorphismsIntoDirectProductToEqua (for IsCapCategoryObject, IsList)

▷ `IsomorphismFromKernelOfJointPairwiseDifferencesOfMorphismsIntoDirectProductToEqualizer(A, D)` (operation)

Returns: a morphism in Hom(Δ, Equalizer(D))

The arguments are an object A and a list of morphisms $D = (\beta_i : A \rightarrow B)_{i=1\dots n}$. The output is a morphism $\Delta \rightarrow$ Equalizer(D), where Δ denotes the kernel object equalizing the morphisms β_i.

6.10 Coequalizer

For an integer $n \geq 0$ and a given list of morphisms $D = (\beta_i : B \rightarrow A)_{i=1\dots n}$, a coequalizer of D consists of three parts:

- an object C,
- a morphism $\pi : A \rightarrow C$ such that $\pi \circ \beta_i \sim_{B,C} \pi \circ \beta_j$ for all pairs i, j,
- a dependent function u mapping the morphism $\tau : A \rightarrow T$ such that $\tau \circ \beta_i \sim_{B,T} \tau \circ \beta_j$ to a morphism $u(\tau) : C \rightarrow T$ such that $u(\tau) \circ \pi \sim_{A,T} \tau$.

The triple (C, π, u) is called a *coequalizer* of D if the morphisms $u(\tau)$ are uniquely determined up to congruence of morphisms. We denote the object C of such a triple by Coequalizer(D). We say that the morphism $u(\tau)$ is induced by the *universal property of the coequalizer*.
Coequalizer is a functorial operation. This means: For a second diagram $D' = (\beta_i' : B' \rightarrow A')_{i=1\dots n}$ and a natural morphism between coequalizer diagrams (i.e., a collection of morphisms $\mu : A \rightarrow A'$ and $\beta : B \rightarrow B'$ such that $\beta_i' \circ \beta \sim_{B,A'} \mu \circ \beta_i$ for $i = 1, \dots n$) we obtain a morphism Coequalizer(D) $\rightarrow$ Coequalizer(D').

$$\begin{tikzcd}
 & & T \\
B \arrow[r, bend left, "\beta_1"] \arrow[r, bend right, "\beta_2"'] \arrow[urr, dotted, bend left, "\tau \circ \beta_1 \sim_{B,T} \tau \circ \beta_2"] \arrow[rr, dotted, bend right=50, "\pi \circ \beta_1 \sim_{B,C} \pi \circ \beta_2"'] & A \arrow[ur, "\tau"] \arrow[r, "\pi"] & C \arrow[u, dashed, "\exists ! u(\tau)"']
\end{tikzcd}$$

6.10.1 Coequalizer

▷ `Coequalizer(arg)` (function)

Returns: an object

This is a convenience method. There are three different ways to use this method:

- The arguments are an object A and a list of morphisms $D = (\beta_i : B \to A)_{i=1...n}$.
- The argument is a nonempty list of morphisms $D = (\beta_i : B \to A)_{i=1...n}$.
- The arguments are morphisms $\beta_1 : B \to A, \dots, \beta_n : B \to A$ with $n \geq 1$.

The output is the coequalizer $\mathrm{Coequalizer}(D)$.

6.10.2 CoequalizerOp (for IsCapCategoryObject, IsList)

▷ `CoequalizerOp(A, D)` (operation)

Returns: an object

The arguments are an object A and a list of morphisms $D = (\beta_i : B \to A)_{i=1...n}$. For convenience, the object A can be omitted and is automatically derived from D, which has to be nonempty. The output is the coequalizer $\mathrm{Coequalizer}(D)$.

6.10.3 ProjectionOntoCoequalizer (for IsCapCategoryObject, IsList)

▷ `ProjectionOntoCoequalizer(A, D)` (operation)

Returns: a morphism in $\mathrm{Hom}(A, \mathrm{Coequalizer}(D))$.

The arguments are an object A and a list of morphisms $D = (\beta_i : B \to A)_{i=1...n}$. For convenience, the object A can be omitted and is automatically derived from D, which has to be nonempty. The output is the projection $\pi : A \to \mathrm{Coequalizer}(D)$.

6.10.4 ProjectionOntoCoequalizerWithGivenCoequalizer (for IsCapCategoryObject, IsList, IsCapCategoryObject)

▷ `ProjectionOntoCoequalizerWithGivenCoequalizer(A, D, C)` (operation)

Returns: a morphism in $\mathrm{Hom}(A, C)$.

The arguments are an object A, a list of morphisms $D = (\beta_i : B \to A)_{i=1...n}$, and an object $C = \mathrm{Coequalizer}(D)$. For convenience, the object A can be omitted and is automatically derived from D, which has to be nonempty. The output is the projection $\pi : A \to C$.

6.10.5 MorphismFromSourceToCoequalizer (for IsCapCategoryObject, IsList)

▷ `MorphismFromSourceToCoequalizer(A, D)` (operation)

Returns: a morphism in $\mathrm{Hom}(B, \mathrm{Coequalizer}(D))$.

The arguments are an object A and a list of morphisms $D = (\beta_i : B \to A)_{i=1...n}$. For convenience, the object A can be omitted and is automatically derived from D, which has to be nonempty. The output is the composition $\mu : B \to \mathrm{Coequalizer}(D)$ of β_1 and the projection $\pi : A \to \mathrm{Coequalizer}(D)$.

6.10.6 MorphismFromSourceToCoequalizerWithGivenCoequalizer (for IsCapCategoryObject, IsList, IsCapCategoryObject)

▷ `MorphismFromSourceToCoequalizerWithGivenCoequalizer(A, D, C)` (operation)

Returns: a morphism in $\mathrm{Hom}(B,C)$.

The arguments are an object A, a list of morphisms $D = (\beta_i : B \rightarrow A)_{i=1\dots n}$ and an object $C = \mathrm{Coequalizer}(D)$. For convenience, the object A can be omitted and is automatically derived from D, which has to be nonempty. The output is the composition $\mu : B \rightarrow C$ of β_1 and the projection $\pi : A \rightarrow C$.

6.10.7 UniversalMorphismFromCoequalizer (for IsCapCategoryObject, IsList, IsCapCategoryObject, IsCapCategoryMorphism)

▷ `UniversalMorphismFromCoequalizer(A, D, T, tau)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{Coequalizer}(D),T)$

The arguments are an object A, a list of morphisms $D = (\beta_i : B \rightarrow A)_{i=1\dots n}$, a test object T, and a morphism $\tau : A \rightarrow T$ such that $\tau \circ \beta_i \sim_{B,T} \tau \circ \beta_j$ for all pairs i, j. For convenience, the object A can be omitted and is automatically derived from D, which has to be nonempty. For convenience, the test object T can be omitted and is automatically derived from `tau` in that case. The output is the morphism $u(\tau) : \mathrm{Coequalizer}(D) \rightarrow T$ given by the universal property of the coequalizer.

6.10.8 UniversalMorphismFromCoequalizerWithGivenCoequalizer (for IsCapCategoryObject, IsList, IsCapCategoryObject, IsCapCategoryMorphism, IsCapCategoryObject)

▷ `UniversalMorphismFromCoequalizerWithGivenCoequalizer(A, D, T, tau, C)` (operation)

Returns: a morphism in $\mathrm{Hom}(C,T)$

The arguments are an object A, a list of morphisms $D = (\beta_i : B \rightarrow A)_{i=1\dots n}$, a test object T, a morphism $\tau : A \rightarrow T$ such that $\tau \circ \beta_i \sim_{B,T} \tau \circ \beta_j$, and an object $C = \mathrm{Coequalizer}(D)$. For convenience, the object A can be omitted and is automatically derived from D, which has to be nonempty. For convenience, the test object T can be omitted and is automatically derived from `tau` in that case. The output is the morphism $u(\tau) : C \rightarrow T$ given by the universal property of the coequalizer.

6.10.9 CoequalizerFunctorial (for IsList, IsCapCategoryMorphism, IsList)

▷ `CoequalizerFunctorial(Ls, mu, Lr)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{Coequalizer}((\beta_i)_{i=1\dots n}),\mathrm{Coequalizer}((\beta_i')_{i=1\dots n}))$

The arguments are a list of morphisms $L_s = (\beta_i : B \rightarrow A)_{i=1\dots n}$, a morphism $\mu : A \rightarrow A'$, and a list of morphisms $L_r = (\beta_i' : B' \rightarrow A')_{i=1\dots n}$ such that there exists a morphism $\beta : B \rightarrow B'$ such that $\beta_i' \circ \beta \sim_{B,A'} \mu \circ \beta_i$ for $i = 1,\dots n$. The output is the morphism $\mathrm{Coequalizer}((\beta_i)_{i=1}^n) \rightarrow \mathrm{Coequalizer}((\beta_i')_{i=1}^n)$ given by the functorality of the coequalizer.

6.10.10 CoequalizerFunctorialWithGivenCoequalizers (for IsCapCategoryObject, IsList, IsCapCategoryMorphism, IsList, IsCapCategoryObject)

▷ `CoequalizerFunctorialWithGivenCoequalizers(s, Ls, mu, Lr, r)` (operation)

Returns: a morphism in $\mathrm{Hom}(s,r)$

The arguments are an object $s = \mathrm{Coequalizer}((\beta_i)_{i=1}^n)$, a list of morphisms $L_s = (\beta_i : B \to A)_{i=1\dots n}$, a morphism $\mu : A \to A'$, and a list of morphisms $L_r = (\beta_i' : B' \to A')_{i=1\dots n}$ such that there exists a morphism $\beta : B \to B'$ such that $\beta_i' \circ \beta \sim_{B,A'} \mu \circ \beta_i$ for $i = 1, \dots n$, and an object $r = \mathrm{Coequalizer}((\beta_i')_{i=1}^n)$. The output is the morphism $s \to r$ given by the functorality of the coequalizer.

6.10.11 JointPairwiseDifferencesOfMorphismsFromCoproduct (for IsCapCategoryObject, IsList)

▷ `JointPairwiseDifferencesOfMorphismsFromCoproduct(A, D)` (operation)

Returns: a morphism in $\mathrm{Hom}(\bigsqcup_{i=1}^{n-1} B, A)$

The arguments are an object A and a list of morphisms $D = (\beta_i : B \to A)_{i=1\dots n}$. The output is a morphism $\bigsqcup_{i=1}^{n-1} B \to A$ such that its cokernel coequalizes the β_i.

6.10.12 IsomorphismFromCoequalizerToCokernelOfJointPairwiseDifferencesOfMorphismsFromCo (for IsCapCategoryObject, IsList)

▷ `IsomorphismFromCoequalizerToCokernelOfJointPairwiseDifferencesOfMorphismsFromCoproduct(A, D)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{Coequalizer}(D), \Delta)$

The arguments are an object A and a list of morphisms $D = (\beta_i : B \to A)_{i=1\dots n}$. The output is a morphism $\mathrm{Coequalizer}(D) \to \Delta$, where Δ denotes the cokernel object coequalizing the morphisms β_i.

6.10.13 IsomorphismFromCokernelOfJointPairwiseDifferencesOfMorphismsFromCoproductToCoe (for IsCapCategoryObject, IsList)

▷ `IsomorphismFromCokernelOfJointPairwiseDifferencesOfMorphismsFromCoproductToCoequalizer(A, D)` (operation)

Returns: a morphism in $\mathrm{Hom}(\Delta, \mathrm{Coequalizer}(D))$

The arguments are an object A and a list of morphisms $D = (\beta_i : B \to A)_{i=1\dots n}$. The output is a morphism $\Delta \to \mathrm{Coequalizer}(D)$, where Δ denotes the cokernel object coequalizing the morphisms β_i.

6.11 CoequalizerOfIdentityAndAutomorphisms

For an integer $n \geq 0$ and a given list of automorphisms $D = (\alpha_i : A \to A)_{i=1\dots n}$, a coequalizer of the identity morphism on A and D consists of three parts:

- an object C,
- a morphism $\pi : A \to C$ such that $\pi \circ \alpha_i \sim_{A,C} \pi \circ \alpha_j$ for all pairs i, j,
- a dependent function u mapping the morphism $\tau : A \to T$ such that $\tau \circ \alpha_i \sim_{A,T} \tau \circ \alpha_j$ to a morphism $u(\tau) : C \to T$ such that $u(\tau) \circ \pi \sim_{A,T} \tau$.

The triple (C, π, u) is called a *coequalizer* of D if the morphisms $u(\tau)$ are uniquely determined up to congruence of morphisms. We denote the object C of such a triple by $\mathrm{CoequalizerOfIdentityAndAutomorphisms}(D)$. We say that the morphism $u(\tau)$ is induced by the *universal property of the coequalizer*.

CoequalizerOfIdentityAndAutomorphisms is a functorial operation. This means: For a second diagram $D' = (\alpha_i' : A' \to A')_{i=1\ldots n}$ and a natural morphism between coequalizer diagrams (i.e., a collection of morphisms $\mu : A \to A'$ and $\alpha : A \to A'$ such that $\alpha_i' \circ \alpha \sim_{A,A'} \mu \circ \alpha_i$ for $i = 1,\ldots n$) we obtain a morphism CoequalizerOfIdentityAndAutomorphisms$(D) \to$ CoequalizerOfIdentityAndAutomorphisms(D').

Given a category $\mathbf{C}$ and a functor $F : G \to \mathbf{C}$ from a group G (viewed as a groupoid on one object). The operation CoequalizerOfIdentityAndAutomorphisms is needed in $\mathbf{C}$ to lift F to a functor $\widetilde{F} : \mathrm{T}(G) \to \mathbf{C}$, where $\mathrm{T}(G)$ is the coequalizer completion of G, namely the category of transitive (left) G-sets.

$$\begin{tikzcd}
 & & T \\
A \arrow[r, shift left, "\alpha_1"] \arrow[r, shift right, "\alpha_2"'] \arrow[urr, dotted, bend left, "\tau \circ \alpha_1 \sim_{A,T} \tau \circ \alpha_2"] \arrow[rr, dotted, bend right, "\pi \circ \alpha_1 \sim_{A,C} \pi \circ \alpha_2"'] & A \arrow[r, "\pi"] \arrow[ur, "\tau"] & C \arrow[u, dashed, "\exists ! u(\tau)"']
\end{tikzcd}$$

6.11.1 CoequalizerOfIdentityAndAutomorphisms

▷ `CoequalizerOfIdentityAndAutomorphisms(arg)` (function)

Returns: an object

This is a convenience method. There are three different ways to use this method:

- The arguments are an object A and a list of automorphisms $D = (\alpha_i : A \to A)_{i=1\ldots n}$.
- The argument is a nonempty list of automorphisms $D = (\alpha_i : A \to A)_{i=1\ldots n}$.
- The arguments are morphisms $\alpha_1 : A \to A, \ldots, \alpha_n : A \to A$ with $n \geq 1$.

The output is the coequalizer CoequalizerOfIdentityAndAutomorphisms(D).

6.11.2 CoequalizerOfIdentityAndAutomorphismsOp (for IsCapCategoryObject, IsList)

▷ `CoequalizerOfIdentityAndAutomorphismsOp(A, D)` (operation)

Returns: an object

The arguments are an object A and a list of automorphisms $D = (\alpha_i : A \to A)_{i=1\ldots n}$. For convenience, the object A can be omitted and is automatically derived from D, which has to be nonempty. The output is the coequalizer CoequalizerOfIdentityAndAutomorphisms(D).

6.11.3 ProjectionOntoCoequalizerOfIdentityAndAutomorphisms (for IsCapCategoryObject, IsList)

▷ `ProjectionOntoCoequalizerOfIdentityAndAutomorphisms(A, D)` (operation)

Returns: a morphism in Hom$(A,$CoequalizerOfIdentityAndAutomorphisms$(D))$.

The arguments are an object A and a list of automorphisms $D = (\alpha_i : A \to A)_{i=1\ldots n}$. For convenience, the object A can be omitted and is automatically derived from D, which has to be nonempty. The output is the projection $\pi : A \to$ CoequalizerOfIdentityAndAutomorphisms(D).

6.11.4 ProjectionOntoCoequalizerOfIdentityAndAutomorphismsWithGivenCoequalizer (for IsCapCategoryObject, IsList, IsCapCategoryObject)

▷ `ProjectionOntoCoequalizerOfIdentityAndAutomorphismsWithGivenCoequalizer(A, D, C)` (operation)

Returns: a morphism in $\mathrm{Hom}(A,C)$.

The arguments are an object A, a list of automorphisms $D = (\alpha_i : A \rightarrow A)_{i=1\dots n}$, and an object $C = \mathrm{CoequalizerOfIdentityAndAutomorphisms}(D)$. For convenience, the object A can be omitted and is automatically derived from D, which has to be nonempty. The output is the projection $\pi : A \rightarrow C$.

6.11.5 UniversalMorphismFromCoequalizerOfIdentityAndAutomorphisms (for IsCapCategoryObject, IsList, IsCapCategoryObject, IsCapCategoryMorphism)

▷ `UniversalMorphismFromCoequalizerOfIdentityAndAutomorphisms(A, D, T, tau)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{CoequalizerOfIdentityAndAutomorphisms}(D),T)$

The arguments are an object A, a list of automorphisms $D = (\alpha_i : A \rightarrow A)_{i=1\dots n}$, a test object T, and a morphism $\tau : A \rightarrow T$ such that $\tau \circ \alpha_i \sim_{A,T} \tau \circ \alpha_j$ for all pairs i,j. For convenience, the object A can be omitted and is automatically derived from D, which has to be nonempty. For convenience, the test object T can be omitted and is automatically derived from `tau` in that case. The output is the morphism $u(\tau) : \mathrm{CoequalizerOfIdentityAndAutomorphisms}(D) \rightarrow T$ given by the universal property of the coequalizer.

6.11.6 UniversalMorphismFromCoequalizerOfIdentityAndAutomorphismsWithGivenCoequalizer (for IsCapCategoryObject, IsList, IsCapCategoryObject, IsCapCategoryMorphism, IsCapCategoryObject)

▷ `UniversalMorphismFromCoequalizerOfIdentityAndAutomorphismsWithGivenCoequalizer(A, D, T, tau, C)` (operation)

Returns: a morphism in $\mathrm{Hom}(C,T)$

The arguments are an object A, a list of automorphisms $D = (\alpha_i : A \rightarrow A)_{i=1\dots n}$, a test object T, a morphism $\tau : A \rightarrow T$ such that $\tau \circ \alpha_i \sim_{A,T} \tau \circ \alpha_j$, and an object $C = \mathrm{CoequalizerOfIdentityAndAutomorphisms}(D)$. For convenience, the object A can be omitted and is automatically derived from D, which has to be nonempty. For convenience, the test object T can be omitted and is automatically derived from `tau` in that case. The output is the morphism $u(\tau) : C \rightarrow T$ given by the universal property of the coequalizer.

6.11.7 CoequalizerOfIdentityAndAutomorphismsFunctorial (for IsList, IsCapCategoryMorphism, IsList)

▷ `CoequalizerOfIdentityAndAutomorphismsFunctorial(Ls, mu, Lr)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{CoequalizerOfIdentityAndAutomorphisms}((\alpha_i)_{i=1\dots n}),\mathrm{CoequalizerOfIdentityAndA$

The arguments are a list of automorphisms $L_s = (\alpha_i : A \rightarrow A)_{i=1\dots n}$, a morphism $\mu : A \rightarrow A'$, and a list of automorphisms $L_r = (\alpha_i' : A' \rightarrow A')_{i=1\dots n}$ such that there exists a morphism $\alpha : A \rightarrow A'$ such that $\alpha_i' \circ \alpha \sim_{A,A'} \mu \circ \alpha_i$ for $i = 1,\dots n$. The output is the morphism $\mathrm{CoequalizerOfIdentityAndAutomorphisms}((\alpha_i)_{i=1}^n) \rightarrow \mathrm{CoequalizerOfIdentityAndAutomorphisms}((\alpha_i')_{i=1}^n)$ given by the functorality of the coequalizer.

6.11.8 CoequalizerOfIdentityAndAutomorphismsFunctorialWithGivenCoequalizers (for IsCapCategoryObject, IsList, IsCapCategoryMorphism, IsList, IsCapCategoryObject)

▷ `CoequalizerOfIdentityAndAutomorphismsFunctorialWithGivenCoequalizers(s, Ls, mu, Lr, r)` (operation)

Returns: a morphism in $\mathrm{Hom}(s,r)$

The arguments are an object $s = \mathrm{CoequalizerOfIdentityAndAutomorphisms}((\alpha_i)_{i=1}^n)$, a list of automorphisms $L_s = (\alpha_i : A \to A)_{i=1\dots n}$, a morphism $\mu : A \to A'$, and a list of automorphisms $L_r = (\alpha_i' : A' \to A')_{i=1\dots n}$ such that there exists a morphism $\alpha : A \to A'$ such that $\alpha_i' \circ \alpha \sim_{A,A'} \mu \circ \alpha_i$ for $i = 1,\dots n$, and an object $r = \mathrm{CoequalizerOfIdentityAndAutomorphisms}((\alpha_i')_{i=1}^n)$. The output is the morphism $s \to r$ given by the functorality of the coequalizer.

6.11.9 IsomorphismFromCoequalizerOfIdentityAndAutomorphismsToCoequalizer (for IsCapCategoryObject, IsList)

▷ `IsomorphismFromCoequalizerOfIdentityAndAutomorphismsToCoequalizer(A, D)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{CoequalizerOfIdentityAndAutomorphisms}(D), \mathrm{Coequalizer}(D))$.

The arguments are an object A and a list of automorphisms $D = (\alpha_i : A \to A)_{i=1\dots n}$. The output is the isomorphism $\mathrm{CoequalizerOfIdentityAndAutomorphisms}(D) \to \mathrm{Coequalizer}(D)$.

6.11.10 IsomorphismFromCoequalizerToCoequalizerOfIdentityAndAutomorphisms (for IsCapCategoryObject, IsList)

▷ `IsomorphismFromCoequalizerToCoequalizerOfIdentityAndAutomorphisms(A, D)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{Coequalizer}(D), \mathrm{CoequalizerOfIdentityAndAutomorphisms}(D))$.

The arguments are an object A and a list of automorphisms $D = (\alpha_i : A \to A)_{i=1\dots n}$. The output is the isomorphism $\mathrm{Coequalizer}(D) \to \mathrm{CoequalizerOfIdentityAndAutomorphisms}(D)$.

6.12 Fiber Product (= Pullback)

For an integer $n \geq 1$ and a given list of morphisms $D = (\beta_i : P_i \to B)_{i=1\dots n}$, a fiber product of D consists of three parts:

- an object P,
- a list of morphisms $\pi = (\pi_i : P \to P_i)_{i=1\dots n}$ such that $\beta_i \circ \pi_i \sim_{P,B} \beta_j \circ \pi_j$ for all pairs i, j.
- a dependent function u mapping each list of morphisms $\tau = (\tau_i : T \to P_i)$ such that $\beta_i \circ \tau_i \sim_{T,B} \beta_j \circ \tau_j$ for all pairs i, j to a morphism $u(\tau) : T \to P$ such that $\pi_i \circ u(\tau) \sim_{T,P_i} \tau_i$ for all $i = 1,\dots,n$.

The triple (P, π, u) is called a *fiber product* of D if the morphisms $u(\tau)$ are uniquely determined up to congruence of morphisms. We denote the object P of such a triple by $\mathrm{FiberProduct}(D)$. We say that the morphism $u(\tau)$ is induced by the *universal property of the fiber product*.
FiberProduct is a functorial operation. This means: For a second diagram $D' = (\beta_i' : P_i' \to B')_{i=1\dots n}$ and a natural morphism between pullback diagrams (i.e., a collection of morphisms $(\mu_i : P_i \to P_i')_{i=1\dots n}$

and $\beta : B \to B'$ such that $\beta_i' \circ \mu_i \sim_{P_i,B'} \beta \circ \beta_i$ for $i = 1, \dots, n$) we obtain a morphism $\mathrm{FiberProduct}(D) \to \mathrm{FiberProduct}(D')$.

$$\begin{tikzcd}
T \arrow[drr, bend left, "\tau_2"] \arrow[ddr, bend right, "\tau_1"'] \arrow[dr, dashed, "\exists ! u(\tau)"] & & \\
& P \arrow[r, "\pi_2"] \arrow[d, "\pi_1"'] & P_2 \arrow[d, "\beta_2"] \\
& P_1 \arrow[r, "\beta_1"'] & B
\end{tikzcd}$$

6.12.1 IsomorphismFromFiberProductToEqualizerOfDirectProductDiagram (for IsList)

▷ `IsomorphismFromFiberProductToEqualizerOfDirectProductDiagram(D)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{FiberProduct}(D), \Delta)$

The argument is a list of morphisms $D = (\beta_i : P_i \to B)_{i=1\dots n}$. The output is a morphism $\mathrm{FiberProduct}(D) \to \Delta$, where Δ denotes the equalizer of the product diagram of the morphisms β_i.

6.12.2 IsomorphismFromEqualizerOfDirectProductDiagramToFiberProduct (for IsList)

▷ `IsomorphismFromEqualizerOfDirectProductDiagramToFiberProduct(D)` (operation)

Returns: a morphism in $\mathrm{Hom}(\Delta, \mathrm{FiberProduct}(D))$

The argument is a list of morphisms $D = (\beta_i : P_i \to B)_{i=1\dots n}$. The output is a morphism $\Delta \to \mathrm{FiberProduct}(D)$, where Δ denotes the equalizer of the product diagram of the morphisms β_i.

6.12.3 FiberProductEmbeddingInDirectProduct (for IsList)

▷ `FiberProductEmbeddingInDirectProduct(D)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{FiberProduct}(D), \prod_{i=1}^n P_i)$

This is a convenience method. The argument is a list of morphisms $D = (\beta_i : P_i \to B)_{i=1\dots n}$. The output is the natural embedding $\mathrm{FiberProduct}(D) \to \prod_{i=1}^n P_i$.

6.12.4 FiberProductEmbeddingInDirectSum (for IsList)

▷ `FiberProductEmbeddingInDirectSum(D)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{FiberProduct}(D), \bigoplus_{i=1}^n P_i)$

This is a convenience method. The argument is a list of morphisms $D = (\beta_i : P_i \to B)_{i=1\dots n}$. The output is the natural embedding $\mathrm{FiberProduct}(D) \to \bigoplus_{i=1}^n P_i$.

6.12.5 FiberProduct

▷ `FiberProduct(arg)` (function)

Returns: an object

This is a convenience method. There are two different ways to use this method:

- The argument is a list of morphisms $D = (\beta_i : P_i \to B)_{i=1\dots n}$.
- The arguments are morphisms $\beta_1 : P_1 \to B, \dots, \beta_n : P_n \to B$.

The output is the fiber product $\mathrm{FiberProduct}(D)$.

6.12.6 FiberProductOp (for IsList)

▷ `FiberProductOp(D)` (operation)

Returns: an object

The argument is a list of morphisms $D = (\beta_i : P_i \to B)_{i=1\dots n}$. The output is the fiber product $\mathrm{FiberProduct}(D)$.

6.12.7 ProjectionInFactorOfFiberProduct (for IsList, IsInt)

▷ `ProjectionInFactorOfFiberProduct(D, k)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{FiberProduct}(D), P_k)$

The arguments are a list of morphisms $D = (\beta_i : P_i \to B)_{i=1\dots n}$ and an integer k. The output is the k-th projection $\pi_k : \mathrm{FiberProduct}(D) \to P_k$.

6.12.8 ProjectionInFactorOfFiberProductWithGivenFiberProduct (for IsList, IsInt, IsCapCategoryObject)

▷ `ProjectionInFactorOfFiberProductWithGivenFiberProduct(D, k, P)` (operation)

Returns: a morphism in $\mathrm{Hom}(P, P_k)$

The arguments are a list of morphisms $D = (\beta_i : P_i \to B)_{i=1\dots n}$, an integer k, and an object $P = \mathrm{FiberProduct}(D)$. The output is the k-th projection $\pi_k : P \to P_k$.

6.12.9 MorphismFromFiberProductToSink (for IsList)

▷ `MorphismFromFiberProductToSink(D)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{FiberProduct}(D), B)$

The arguments are a list of morphisms $D = (\beta_i : P_i \to B)_{i=1\dots n}$. The output is the composition $\mu : \mathrm{FiberProduct}(D) \to B$ of the 1-st projection $\pi_1 : \mathrm{FiberProduct}(D) \to P_1$ and β_1.

6.12.10 MorphismFromFiberProductToSinkWithGivenFiberProduct (for IsList, IsCapCategoryObject)

▷ `MorphismFromFiberProductToSinkWithGivenFiberProduct(D, P)` (operation)

Returns: a morphism in $\mathrm{Hom}(P, B)$

The arguments are a list of morphisms $D = (\beta_i : P_i \to B)_{i=1\dots n}$ and an object $P = \mathrm{FiberProduct}(D)$. The output is the composition $\mu : P \to B$ of the 1-st projection $\pi_1 : P \to P_1$ and β_1.

6.12.11 UniversalMorphismIntoFiberProduct (for IsList, IsCapCategoryObject, IsList)

▷ `UniversalMorphismIntoFiberProduct(D, T, tau)` (operation)

Returns: a morphism in $\mathrm{Hom}(T, \mathrm{FiberProduct}(D))$

The arguments are a list of morphisms $D = (\beta_i : P_i \to B)_{i=1\dots n}$, a test object T, and a list of morphisms $\tau = (\tau_i : T \to P_i)$ such that $\beta_i \circ \tau_i \sim_{T,B} \beta_j \circ \tau_j$ for all pairs i, j. For convenience, the test object `T` can be omitted and is automatically derived from `tau` in that case. The output is the morphism $u(\tau) : T \to \mathrm{FiberProduct}(D)$ given by the universal property of the fiber product.

6.12.12 UniversalMorphismIntoFiberProductWithGivenFiberProduct (for IsList, IsCapCategoryObject, IsList, IsCapCategoryObject)

▷ `UniversalMorphismIntoFiberProductWithGivenFiberProduct(D, T, tau, P)` (operation)

Returns: a morphism in $\mathrm{Hom}(T, P)$

The arguments are a list of morphisms $D = (\beta_i : P_i \to B)_{i=1\dots n}$, a test object T, a list of morphisms $\tau = (\tau_i : T \to P_i)$ such that $\beta_i \circ \tau_i \sim_{T,B} \beta_j \circ \tau_j$ for all pairs i, j, and an object $P = \mathrm{FiberProduct}(D)$. For convenience, the test object `T` can be omitted and is automatically derived from `tau` in that case. The output is the morphism $u(\tau) : T \to P$ given by the universal property of the fiber product.

6.12.13 FiberProductFunctorial (for IsList, IsList, IsList)

▷ `FiberProductFunctorial(Ls, Lm, Lr)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{FiberProduct}((\beta_i)_{i=1\dots n}), \mathrm{FiberProduct}((\beta_i')_{i=1\dots n}))$

The arguments are three lists of morphisms $L_s = (\beta_i : P_i \to B)_{i=1\dots n}$, $L_m = (\mu_i : P_i \to P_i')_{i=1\dots n}$, $L_r = (\beta_i' : P_i' \to B')_{i=1\dots n}$ having the same length n such that there exists a morphism $\beta : B \to B'$ such that $\beta_i' \circ \mu_i \sim_{P_i,B'} \beta \circ \beta_i$ for $i = 1, \dots, n$. The output is the morphism $\mathrm{FiberProduct}((\beta_i)_{i=1\dots n}) \to \mathrm{FiberProduct}((\beta_i')_{i=1\dots n})$ given by the functoriality of the fiber product.

6.12.14 FiberProductFunctorialWithGivenFiberProducts (for IsCapCategoryObject, IsList, IsList, IsList, IsCapCategoryObject)

▷ `FiberProductFunctorialWithGivenFiberProducts(s, Ls, Lm, Lr, r)` (operation)

Returns: a morphism in $\mathrm{Hom}(s, r)$

The arguments are an object $s = \mathrm{FiberProduct}((\beta_i)_{i=1\dots n})$, three lists of morphisms $L_s = (\beta_i : P_i \to B)_{i=1\dots n}$, $L_m = (\mu_i : P_i \to P_i')_{i=1\dots n}$, $L_r = (\beta_i' : P_i' \to B')_{i=1\dots n}$ having the same length n such that there exists a morphism $\beta : B \to B'$ such that $\beta_i' \circ \mu_i \sim_{P_i,B'} \beta \circ \beta_i$ for $i = 1, \dots, n$, and an object $r = \mathrm{FiberProduct}((\beta_i')_{i=1\dots n})$. The output is the morphism $s \to r$ given by the functoriality of the fiber product.

6.13 Pushout

For an integer $n \geq 1$ and a given list of morphisms $D = (\beta_i : B \to I_i)_{i=1\dots n}$, a pushout of D consists of three parts:

- an object I,
- a list of morphisms $\iota = (\iota_i : I_i \to I)_{i=1\dots n}$ such that $\iota_i \circ \beta_i \sim_{B,I} \iota_j \circ \beta_j$ for all pairs i, j,

- a dependent function u mapping each list of morphisms $\tau = (\tau_i : I_i \to T)_{i=1\dots n}$ such that $\tau_i \circ \beta_i \sim_{B,T} \tau_j \circ \beta_j$ to a morphism $u(\tau) : I \to T$ such that $u(\tau) \circ \iota_i \sim_{I_i,T} \tau_i$ for all $i = 1, \dots, n$.

The triple (I, ι, u) is called a *pushout* of D if the morphisms $u(\tau)$ are uniquely determined up to congruence of morphisms. We denote the object I of such a triple by $\mathrm{Pushout}(D)$. We say that the morphism $u(\tau)$ is induced by the *universal property of the pushout*.
Pushout is a functorial operation. This means: For a second diagram $D' = (\beta_i' : B' \to I_i')_{i=1\dots n}$ and a natural morphism between pushout diagrams (i.e., a collection of morphisms $(\mu_i : I_i \to I_i')_{i=1\dots n}$ and $\beta : B \to B'$ such that $\beta_i' \circ \beta \sim_{B,I_i'} \mu_i \circ \beta_i$ for $i = 1, \dots n$) we obtain a morphism $\mathrm{Pushout}(D) \to \mathrm{Pushout}(D')$.

$$\begin{tikzcd}
 & & T \\
I_2 \arrow[r, "\iota_2"] \arrow[urr, bend left, "\tau_2"] & I \arrow[ur, dashed, "\exists ! u(\tau)"] & \\
B \arrow[u, "\beta_2"] \arrow[r, "\beta_1"'] & I_1 \arrow[u, "\iota_1"'] \arrow[uur, bend right, "\tau_1"'] &
\end{tikzcd}$$

6.13.1 IsomorphismFromPushoutToCoequalizerOfCoproductDiagram (for IsList)

▷ `IsomorphismFromPushoutToCoequalizerOfCoproductDiagram(D)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{Pushout}(D), \Delta)$

The argument is a list of morphisms $D = (\beta_i : B \to I_i)_{i=1\dots n}$. The output is a morphism $\mathrm{Pushout}(D) \to \Delta$, where Δ denotes the coequalizer of the coproduct diagram of the morphisms β_i.

6.13.2 IsomorphismFromCoequalizerOfCoproductDiagramToPushout (for IsList)

▷ `IsomorphismFromCoequalizerOfCoproductDiagramToPushout(D)` (operation)

Returns: a morphism in $\mathrm{Hom}(\Delta, \mathrm{Pushout}(D))$

The argument is a list of morphisms $D = (\beta_i : B \to I_i)_{i=1\dots n}$. The output is a morphism $\Delta \to \mathrm{Pushout}(D)$, where Δ denotes the coequalizer of the coproduct diagram of the morphisms β_i.

6.13.3 PushoutProjectionFromCoproduct (for IsList)

▷ `PushoutProjectionFromCoproduct(D)` (operation)

Returns: a morphism in $\mathrm{Hom}(\bigsqcup i = 1^n I_i, \mathrm{Pushout}(D))$

This is a convenience method. The argument is a list of morphisms $D = (\beta_i : B \to I_i)_{i=1\dots n}$. The output is the natural projection $\bigsqcup_{i=1}^n I_i \to \mathrm{Pushout}(D)$.

6.13.4 PushoutProjectionFromDirectSum (for IsList)

▷ `PushoutProjectionFromDirectSum(D)` (operation)

Returns: a morphism in $\mathrm{Hom}(\bigoplus_{i=1}^n I_i, \mathrm{Pushout}(D))$

This is a convenience method. The argument is a list of morphisms $D = (\beta_i : B \to I_i)_{i=1\dots n}$. The output is the natural projection $\bigoplus_{i=1}^n I_i \to \mathrm{Pushout}(D)$.

6.13.5 Pushout (for IsList)

▷ `Pushout(D)` (operation)

Returns: an object

The argument is a list of morphisms $D = (\beta_i : B \to I_i)_{i=1\dots n}$. The output is the pushout $\mathrm{Pushout}(D)$.

6.13.6 Pushout (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `Pushout(D)` (operation)

Returns: an object

This is a convenience method. The arguments are a morphism α and a morphism β. The output is the pushout $\mathrm{Pushout}(\alpha,\beta)$.

6.13.7 InjectionOfCofactorOfPushout (for IsList, IsInt)

▷ `InjectionOfCofactorOfPushout(D, k)` (operation)

Returns: a morphism in $\mathrm{Hom}(I_k, \mathrm{Pushout}(D))$.

The arguments are a list of morphisms $D = (\beta_i : B \to I_i)_{i=1\dots n}$ and an integer k. The output is the k-th injection $\iota_k : I_k \to \mathrm{Pushout}(D)$.

6.13.8 InjectionOfCofactorOfPushoutWithGivenPushout (for IsList, IsInt, IsCapCategoryObject)

▷ `InjectionOfCofactorOfPushoutWithGivenPushout(D, k, I)` (operation)

Returns: a morphism in $\mathrm{Hom}(I_k, I)$.

The arguments are a list of morphisms $D = (\beta_i : B \to I_i)_{i=1\dots n}$, an integer k, and an object $I = \mathrm{Pushout}(D)$. The output is the k-th injection $\iota_k : I_k \to I$.

6.13.9 MorphismFromSourceToPushout (for IsList)

▷ `MorphismFromSourceToPushout(D)` (operation)

Returns: a morphism in $\mathrm{Hom}(B, \mathrm{Pushout}(D))$.

The arguments are a list of morphisms $D = (\beta_i : B \to I_i)_{i=1\dots n}$. The output is the composition $\mu : B \to \mathrm{Pushout}(D)$ of β_1 and the 1-st injection $\iota_1 : I_1 \to \mathrm{Pushout}(D)$.

6.13.10 MorphismFromSourceToPushoutWithGivenPushout (for IsList, IsCapCategoryObject)

▷ `MorphismFromSourceToPushoutWithGivenPushout(D, I)` (operation)

Returns: a morphism in $\mathrm{Hom}(B, I)$.

The arguments are a list of morphisms $D = (\beta_i : B \to I_i)_{i=1\dots n}$ and an object $I = \mathrm{Pushout}(D)$. The output is the composition $\mu : B \to I$ of β_1 and the 1-st injection $\iota_1 : I_1 \to I$.

6.13.11 UniversalMorphismFromPushout (for IsList, IsCapCategoryObject, IsList)

▷ `UniversalMorphismFromPushout(D, T, tau)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{Pushout}(D), T)$

The arguments are a list of morphisms $D = (\beta_i : B \to I_i)_{i=1\dots n}$, a test object T, and a list of morphisms $\tau = (\tau_i : I_i \to T)_{i=1\dots n}$ such that $\tau_i \circ \beta_i \sim_{B,T} \tau_j \circ \beta_j$. For convenience, the test object `T` can be omitted and is automatically derived from `tau` in that case. The output is the morphism $u(\tau) : \mathrm{Pushout}(D) \to T$ given by the universal property of the pushout.

6.13.12 UniversalMorphismFromPushoutWithGivenPushout (for IsList, IsCapCategoryObject, IsList, IsCapCategoryObject)

▷ `UniversalMorphismFromPushoutWithGivenPushout(D, T, tau, I)` (operation)

Returns: a morphism in $\mathrm{Hom}(I,T)$

The arguments are a list of morphisms $D = (\beta_i : B \to I_i)_{i=1\dots n}$, a test object T, a list of morphisms $\tau = (\tau_i : I_i \to T)_{i=1\dots n}$ such that $\tau_i \circ \beta_i \sim_{B,T} \tau_j \circ \beta_j$, and an object $I = \mathrm{Pushout}(D)$. For convenience, the test object `T` can be omitted and is automatically derived from `tau` in that case. The output is the morphism $u(\tau) : I \to T$ given by the universal property of the pushout.

6.13.13 PushoutFunctorial (for IsList, IsList, IsList)

▷ `PushoutFunctorial(Ls, Lm, Lr)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{Pushout}((\beta_i)_{i=1}^n), \mathrm{Pushout}((\beta_i')_{i=1}^n))$

The arguments are three lists of morphisms $L_s = (\beta_i : B \to I_i)_{i=1\dots n}$, $L_m = (\mu_i : I_i \to I_i')_{i=1\dots n}$, $L_r = (\beta_i' : B' \to I_i')_{i=1\dots n}$ having the same length n such that there exists a morphism $\beta : B \to B'$ such that $\beta_i' \circ \beta \sim_{B,I_i'} \mu_i \circ \beta_i$ for $i = 1,\dots n$. The output is the morphism $\mathrm{Pushout}((\beta_i)_{i=1}^n) \to \mathrm{Pushout}((\beta_i')_{i=1}^n)$ given by the functoriality of the pushout.

6.13.14 PushoutFunctorialWithGivenPushouts (for IsCapCategoryObject, IsList, IsList, IsList, IsCapCategoryObject)

▷ `PushoutFunctorialWithGivenPushouts(s, Ls, Lm, Lr, r)` (operation)

Returns: a morphism in $\mathrm{Hom}(s,r)$

The arguments are an object $s = \mathrm{Pushout}((\beta_i)_{i=1}^n)$, three lists of morphisms $L_s = (\beta_i : B \to I_i)_{i=1\dots n}$, $L_m = (\mu_i : I_i \to I_i')_{i=1\dots n}$, $L_r = (\beta_i' : B' \to I_i')_{i=1\dots n}$ having the same length n such that there exists a morphism $\beta : B \to B'$ such that $\beta_i' \circ \beta \sim_{B,I_i'} \mu_i \circ \beta_i$ for $i = 1,\dots n$, and an object $r = \mathrm{Pushout}((\beta_i')_{i=1}^n)$. The output is the morphism $s \to r$ given by the functoriality of the pushout.

6.14 Image

For a given morphism $\alpha : A \to B$, an image of α consists of four parts:

- an object I,
- a morphism $c : A \to I$,
- a monomorphism $\iota : I \hookrightarrow B$ such that $\iota \circ c \sim_{A,B} \alpha$,
- a dependent function u mapping each pair of morphisms $\tau = (\tau_1 : A \to T, \tau_2 : T \hookrightarrow B)$ where τ_2 is a monomorphism such that $\tau_2 \circ \tau_1 \sim_{A,B} \alpha$ to a morphism $u(\tau) : I \to T$ such that $\tau_2 \circ u(\tau) \sim_{I,B} \iota$ and $u(\tau) \circ c \sim_{A,T} \tau_1$.

The 4-tuple (I, c, ι, u) is called an *image* of α if the morphisms $u(\tau)$ are uniquely determined up to congruence of morphisms. We denote the object I of such a 4-tuple by $\mathrm{im}(\alpha)$. We say that the morphism $u(\tau)$ is induced by the *universal property of the image*.

$$\begin{tikzcd}
A \arrow[rr, "\alpha"] \arrow[dr, "c"] \arrow[ddr, bend right, "\tau_1"'] & & B \\
& I \arrow[ur, hook, "\iota"'] \arrow[d, dashed, "u(\tau)"] & \\
& T \arrow[uur, hook, bend right, "\tau_2"'] &
\end{tikzcd}$$

6.14.1 IsomorphismFromImageObjectToKernelOfCokernel (for IsCapCategoryMorphism)

▷ `IsomorphismFromImageObjectToKernelOfCokernel(alpha)` (attribute)

Returns: a morphism in $\mathrm{Hom}(\mathrm{im}(\alpha), \mathrm{KernelObject}(\mathrm{CokernelProjection}(\alpha)))$

The argument is a morphism α. The output is the canonical morphism $\mathrm{im}(\alpha) \rightarrow \mathrm{KernelObject}(\mathrm{CokernelProjection}(\alpha))$.

6.14.2 IsomorphismFromKernelOfCokernelToImageObject (for IsCapCategoryMorphism)

▷ `IsomorphismFromKernelOfCokernelToImageObject(alpha)` (attribute)

Returns: a morphism in $\mathrm{Hom}(\mathrm{KernelObject}(\mathrm{CokernelProjection}(\alpha)), \mathrm{im}(\alpha))$

The argument is a morphism α. The output is the canonical morphism $\mathrm{KernelObject}(\mathrm{CokernelProjection}(\alpha)) \rightarrow \mathrm{im}(\alpha)$.

6.14.3 ImageObject (for IsCapCategoryMorphism)

▷ `ImageObject(alpha)` (attribute)

Returns: an object

The argument is a morphism α. The output is the image $\mathrm{im}(\alpha)$.

6.14.4 ImageEmbedding (for IsCapCategoryMorphism)

▷ `ImageEmbedding(alpha)` (attribute)

Returns: a morphism in $\mathrm{Hom}(\mathrm{im}(\alpha), B)$

The argument is a morphism $\alpha : A \rightarrow B$. The output is the image embedding $\iota : \mathrm{im}(\alpha) \hookrightarrow B$.

6.14.5 ImageEmbeddingWithGivenImageObject (for IsCapCategoryMorphism, IsCapCategoryObject)

▷ `ImageEmbeddingWithGivenImageObject(alpha, I)` (operation)

Returns: a morphism in $\mathrm{Hom}(I, B)$

The argument is a morphism $\alpha : A \rightarrow B$ and an object $I = \mathrm{im}(\alpha)$. The output is the image embedding $\iota : I \hookrightarrow B$.

6.14.6 CoastrictionToImage (for IsCapCategoryMorphism)

▷ `CoastrictionToImage(alpha)` (attribute)

Returns: a morphism in $\mathrm{Hom}(A, \mathrm{im}(\alpha))$

The argument is a morphism $\alpha : A \rightarrow B$. The output is the coastriction to image $c : A \rightarrow \mathrm{im}(\alpha)$.

6.14.7 CoastrictionToImageWithGivenImageObject (for IsCapCategoryMorphism, IsCapCategoryObject)

▷ `CoastrictionToImageWithGivenImageObject(alpha, I)` (operation)

Returns: a morphism in $\mathrm{Hom}(A, I)$

The argument is a morphism $\alpha : A \rightarrow B$ and an object $I = \mathrm{im}(\alpha)$. The output is the coastriction to image $c : A \rightarrow I$.

6.14.8 UniversalMorphismFromImage (for IsCapCategoryMorphism, IsList)

▷ `UniversalMorphismFromImage(alpha, tau)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{im}(\alpha), T)$

The arguments are a morphism $\alpha : A \rightarrow B$ and a pair of morphisms $\tau = (\tau_1 : A \rightarrow T, \tau_2 : T \hookrightarrow B)$ where τ_2 is a monomorphism such that $\tau_2 \circ \tau_1 \sim_{A,B} \alpha$. The output is the morphism $u(\tau) : \mathrm{im}(\alpha) \rightarrow T$ given by the universal property of the image.

6.14.9 UniversalMorphismFromImageWithGivenImageObject (for IsCapCategoryMorphism, IsList, IsCapCategoryObject)

▷ `UniversalMorphismFromImageWithGivenImageObject(alpha, tau, I)` (operation)

Returns: a morphism in $\mathrm{Hom}(I, T)$

The arguments are a morphism $\alpha : A \rightarrow B$, a pair of morphisms $\tau = (\tau_1 : A \rightarrow T, \tau_2 : T \hookrightarrow B)$ where τ_2 is a monomorphism such that $\tau_2 \circ \tau_1 \sim_{A,B} \alpha$, and an object $I = \mathrm{im}(\alpha)$. The output is the morphism $u(\tau) : \mathrm{im}(\alpha) \rightarrow T$ given by the universal property of the image.

6.14.10 ImageObjectFunctorial (for IsCapCategoryMorphism, IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `ImageObjectFunctorial(alpha, nu, alpha_prime)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{ImageObject}(\alpha), \mathrm{ImageObject}(\alpha'))$

The arguments are three morphisms $\alpha : A \rightarrow B$, $\nu : B \rightarrow B'$, $\alpha' : A' \rightarrow B'$. The output is the morphism $\mathrm{ImageObject}(\alpha) \rightarrow \mathrm{ImageObject}(\alpha')$ given by the functoriality of the image.

6.14.11 ImageObjectFunctorialWithGivenImageObjects (for IsCapCategoryObject, IsCapCategoryMorphism, IsCapCategoryMorphism, IsCapCategoryMorphism, IsCapCategoryObject)

▷ `ImageObjectFunctorialWithGivenImageObjects(s, alpha, nu, alpha_prime, r)` (operation)

Returns: a morphism in $\mathrm{Hom}(s,r)$

The arguments are an object $s = \mathrm{ImageObject}(\alpha)$, three morphisms $\alpha : A \to B$, $\nu : B \to B'$, $\alpha' : A' \to B'$, and an object $r = \mathrm{ImageObject}(\alpha')$. The output is the morphism $\mathrm{ImageObject}(\alpha) \to \mathrm{ImageObject}(\alpha')$ given by the functoriality of the image.

6.15 Coimage

For a given morphism $\alpha : A \to B$, a coimage of α consists of four parts:

- an object C,
- an epimorphism $\pi : A \twoheadrightarrow C$,
- a morphism $a : C \to B$ such that $a \circ \pi \sim_{A,B} \alpha$,
- a dependent function u mapping each pair of morphisms $\tau = (\tau_1 : A \twoheadrightarrow T, \tau_2 : T \to B)$ where τ_1 is an epimorphism such that $\tau_2 \circ \tau_1 \sim_{A,B} \alpha$ to a morphism $u(\tau) : T \to C$ such that $u(\tau) \circ \tau_1 \sim_{A,C} \pi$ and $a \circ u(\tau) \sim_{T,B} \tau_2$.

The 4-tuple (C, π, a, u) is called a *coimage* of α if the morphisms $u(\tau)$ are uniquely determined up to congruence of morphisms. We denote the object C of such a 4-tuple by $\mathrm{coim}(\alpha)$. We say that the morphism $u(\tau)$ is induced by the *universal property of the coimage*.

$$
\begin{tikzcd}
A \arrow[rr, "\alpha"] \arrow[dr, "\pi"] \arrow[ddr, bend right, two heads, "\tau_1"'] & & B \\
& C \arrow[ur, "a"'] & \\
& T \arrow[u, dashed, "u(\tau)"'] \arrow[uur, bend right, "\tau_2"'] &
\end{tikzcd}
$$

6.15.1 IsomorphismFromCoimageToCokernelOfKernel (for IsCapCategoryMorphism)

▷ `IsomorphismFromCoimageToCokernelOfKernel(alpha)` (attribute)

Returns: a morphism in $\mathrm{Hom}(\mathrm{coim}(\alpha), \mathrm{CokernelObject}(\mathrm{KernelEmbedding}(\alpha)))$.

The argument is a morphism $\alpha : A \to B$. The output is the canonical morphism $\mathrm{coim}(\alpha) \to \mathrm{CokernelObject}(\mathrm{KernelEmbedding}(\alpha))$.

6.15.2 IsomorphismFromCokernelOfKernelToCoimage (for IsCapCategoryMorphism)

▷ `IsomorphismFromCokernelOfKernelToCoimage(alpha)` (attribute)

Returns: a morphism in $\mathrm{Hom}(\mathrm{CokernelObject}(\mathrm{KernelEmbedding}(\alpha)), \mathrm{coim}(\alpha))$.

The argument is a morphism $\alpha : A \to B$. The output is the canonical morphism $\mathrm{CokernelObject}(\mathrm{KernelEmbedding}(\alpha)) \to \mathrm{coim}(\alpha)$.

6.15.3 CoimageObject (for IsCapCategoryMorphism)

▷ `CoimageObject(alpha)` (attribute)

Returns: an object

The argument is a morphism α. The output is the coimage $\mathrm{coim}(\alpha)$.

6.15.4 CoimageProjection (for IsCapCategoryMorphism)

▷ `CoimageProjection(alpha)` (attribute)

Returns: a morphism in $\mathrm{Hom}(A, \mathrm{coim}(\alpha))$

The argument is a morphism $\alpha : A \rightarrow B$. The output is the coimage projection $\pi : A \twoheadrightarrow \mathrm{coim}(\alpha)$.

6.15.5 CoimageProjectionWithGivenCoimageObject (for IsCapCategoryMorphism, IsCapCategoryObject)

▷ `CoimageProjectionWithGivenCoimageObject(alpha, C)` (operation)

Returns: a morphism in $\mathrm{Hom}(A, C)$

The arguments are a morphism $\alpha : A \rightarrow B$ and an object $C = \mathrm{coim}(\alpha)$. The output is the coimage projection $\pi : A \twoheadrightarrow C$.

6.15.6 AstrictionToCoimage (for IsCapCategoryMorphism)

▷ `AstrictionToCoimage(alpha)` (attribute)

Returns: a morphism in $\mathrm{Hom}(\mathrm{coim}(\alpha), B)$

The argument is a morphism $\alpha : A \rightarrow B$. The output is the astriction to coimage $a : \mathrm{coim}(\alpha) \rightarrow B$.

6.15.7 AstrictionToCoimageWithGivenCoimageObject (for IsCapCategoryMorphism, IsCapCategoryObject)

▷ `AstrictionToCoimageWithGivenCoimageObject(alpha, C)` (operation)

Returns: a morphism in $\mathrm{Hom}(C, B)$

The argument are a morphism $\alpha : A \rightarrow B$ and an object $C = \mathrm{coim}(\alpha)$. The output is the astriction to coimage $a : C \rightarrow B$.

6.15.8 UniversalMorphismIntoCoimage (for IsCapCategoryMorphism, IsList)

▷ `UniversalMorphismIntoCoimage(alpha, tau)` (operation)

Returns: a morphism in $\mathrm{Hom}(T, \mathrm{coim}(\alpha))$

The arguments are a morphism $\alpha : A \rightarrow B$ and a pair of morphisms $\tau = (\tau_1 : A \twoheadrightarrow T, \tau_2 : T \rightarrow B)$ where τ_1 is an epimorphism such that $\tau_2 \circ \tau_1 \sim_{A,B} \alpha$. The output is the morphism $u(\tau) : T \rightarrow \mathrm{coim}(\alpha)$ given by the universal property of the coimage.

6.15.9 UniversalMorphismIntoCoimageWithGivenCoimageObject (for IsCapCategoryMorphism, IsList, IsCapCategoryObject)

▷ `UniversalMorphismIntoCoimageWithGivenCoimageObject(alpha, tau, C)` (operation)

Returns: a morphism in $\mathrm{Hom}(T, C)$

The arguments are a morphism $\alpha : A \rightarrow B$, a pair of morphisms $\tau = (\tau_1 : A \twoheadrightarrow T, \tau_2 : T \rightarrow B)$ where τ_1 is an epimorphism such that $\tau_2 \circ \tau_1 \sim_{A,B} \alpha$, and an object $C = \mathrm{coim}(\alpha)$. The output is the morphism $u(\tau) : T \rightarrow C$ given by the universal property of the coimage.

Whenever the `CoastrictionToImage` is an epi, or the `AstrictionToCoimage` is a mono, there is a canonical morphism from the image to the coimage. If this canonical morphism is an isomorphism, we call it the *canonical identification* (between image and coimage).

6.15.10 CoimageObjectFunctorial (for IsCapCategoryMorphism, IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `CoimageObjectFunctorial(alpha, mu, alpha_prime)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{CoimageObject}(\alpha), \mathrm{CoimageObject}(\alpha'))$

The arguments are three morphisms $\alpha : A \rightarrow B, \mu : A \rightarrow A', \alpha' : A' \rightarrow B'$. The output is the morphism $\mathrm{CoimageObject}(\alpha) \rightarrow \mathrm{CoimageObject}(\alpha')$ given by the functoriality of the coimage.

6.15.11 CoimageObjectFunctorialWithGivenCoimageObjects (for IsCapCategoryObject, IsCapCategoryMorphism, IsCapCategoryMorphism, IsCapCategoryMorphism, IsCapCategoryObject)

▷ `CoimageObjectFunctorialWithGivenCoimageObjects(s, alpha, mu, alpha_prime, r)` (operation)

Returns: a morphism in $\mathrm{Hom}(s, r)$

The arguments are an object $s = \mathrm{CoimageObject}(\alpha)$, three morphisms $\alpha : A \rightarrow B, \mu : A \rightarrow A', \alpha' : A' \rightarrow B'$, and an object $r = \mathrm{CoimageObject}(\alpha')$. The output is the morphism $\mathrm{CoimageObject}(\alpha) \rightarrow \mathrm{CoimageObject}(\alpha')$ given by the functoriality of the coimage.

6.16 Morphism between Coimage and Image

6.16.1 MorphismFromCoimageToImage (for IsCapCategoryMorphism)

▷ `MorphismFromCoimageToImage(alpha)` (attribute)

Returns: a morphism in $\mathrm{Hom}(\mathrm{coim}(\alpha), \mathrm{im}(\alpha))$

The argument is a morphism $\alpha : A \rightarrow B$. The output is the canonical morphism (in a preabelian category) $\mathrm{coim}(\alpha) \rightarrow \mathrm{im}(\alpha)$.

6.16.2 MorphismFromCoimageToImageWithGivenObjects (for IsCapCategoryObject, IsCapCategoryMorphism, IsCapCategoryObject)

▷ `MorphismFromCoimageToImageWithGivenObjects(C, alpha, I)` (operation)

Returns: a morphism in $\mathrm{Hom}(C, I)$

The argument is an object $C = \mathrm{coim}(\alpha)$, a morphism $\alpha : A \rightarrow B$, and an object $I = \mathrm{im}(\alpha)$. The output is the canonical morphism (in a preabelian category) $C \rightarrow I$.

6.16.3 InverseOfMorphismFromCoimageToImage (for IsCapCategoryMorphism)

▷ `InverseOfMorphismFromCoimageToImage(alpha)` (attribute)

Returns: a morphism in $\mathrm{Hom}(\mathrm{im}(\alpha), \mathrm{coim}(\alpha))$

The argument is a morphism $\alpha : A \rightarrow B$. The output is the inverse of the canonical morphism (in an abelian category) $\mathrm{im}(\alpha) \rightarrow \mathrm{coim}(\alpha)$.

6.16.4 InverseOfMorphismFromCoimageToImageWithGivenObjects (for IsCapCategoryObject, IsCapCategoryMorphism, IsCapCategoryObject)

▷ `InverseOfMorphismFromCoimageToImageWithGivenObjects(I, alpha, C)` (operation)

Returns: a morphism in $\mathrm{Hom}(I,C)$

The argument is an object $C = \mathrm{coim}(\alpha)$, a morphism $\alpha : A \rightarrow B$, and an object $I = \mathrm{im}(\alpha)$. The output is the inverse of the canonical morphism (in an abelian category) $I \rightarrow C$.

6.17 Homology objects

In an abelian category, we can define the operation that takes as an input a pair of morphisms $\alpha : A \rightarrow B$, $\beta : B \rightarrow C$ and outputs the subquotient of B given by

- $H := \mathrm{KernelObject}(\beta)/(\mathrm{KernelObject}(\beta) \cap \mathrm{ImageObject}(\alpha))$.

This object is called a *homology object* of the pair α, β. Note that we do not need the precomposition of α and β to be zero in order to make sense of this notion. Moreover, given a second pair $\gamma : D \rightarrow E$, $\delta : E \rightarrow F$ of morphisms, and a morphism $\varepsilon : B \rightarrow E$ such that there exists $\omega_1 : A \rightarrow D$, $\omega_2 : C \rightarrow F$ with $\varepsilon \circ \alpha \sim_{A,E} \gamma \circ \omega_1$ and $\omega_2 \circ \beta \sim_{B,F} \delta \circ \varepsilon$ there is a functorial way to obtain from these data a morphism between the two corresponding homology objects.

6.17.1 HomologyObject (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `HomologyObject(alpha, beta)` (operation)

Returns: an object

The arguments are two morphisms $\alpha : A \rightarrow B, \beta : B \rightarrow C$. The output is the homology object H of this pair.

6.17.2 HomologyObjectFunctorial (for IsCapCategoryMorphism, IsCapCategoryMorphism, IsCapCategoryMorphism, IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `HomologyObjectFunctorial(alpha, beta, epsilon, gamma, delta)` (operation)

Returns: a morphism in $\mathrm{Hom}(H_1, H_2)$

The argument are five morphisms $\alpha : A \rightarrow B, \beta : B \rightarrow C, \varepsilon : B \rightarrow E, \gamma : D \rightarrow E, \delta : E \rightarrow F$ such that there exists $\omega_1 : A \rightarrow D$, $\omega_2 : C \rightarrow F$ with $\varepsilon \circ \alpha \sim_{A,E} \gamma \circ \omega_1$ and $\omega_2 \circ \beta \sim_{B,F} \delta \circ \varepsilon$. The output is the functorial morphism induced by ε between the corresponding homology objects H_1 and H_2, where H_1 denotes the homology object of the pair α, β, and H_2 denotes the homology object of the pair γ, δ.

6.17.3 HomologyObjectFunctorialWithGivenHomologyObjects (for IsCapCategoryObject, IsList, IsCapCategoryObject)

▷ `HomologyObjectFunctorialWithGivenHomologyObjects(H_1, L, H_2)` (operation)

Returns: a morphism in $\mathrm{Hom}(H_1, H_2)$

The arguments are an object H_1, a list L consisting of five morphisms $\alpha : A \rightarrow B$, $\beta : B \rightarrow C$, $\varepsilon : B \rightarrow E$, $\gamma : D \rightarrow E, \delta : E \rightarrow F$, and an object H_2, such that $H_1 = \mathrm{HomologyObject}(\alpha, \beta)$ and $H_2 = \mathrm{HomologyObject}(\gamma, \delta)$, and such that there exists $\omega_1 : A \rightarrow D$, $\omega_2 : C \rightarrow F$ with $\varepsilon \circ \alpha \sim_{A,E} \gamma \circ \omega_1$ and $\omega_2 \circ \beta \sim_{B,F} \delta \circ \varepsilon$. The output is the functorial morphism induced by ε between the corresponding homology objects H_1 and H_2, where H_1 denotes the homology object of the pair α, β, and H_2 denotes the homology object of the pair γ, δ.

6.17.4 IsomorphismFromHomologyObjectToItsConstructionAsAnImageObject (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `IsomorphismFromHomologyObjectToItsConstructionAsAnImageObject(alpha, beta)` (operation)

Returns: a morphism in $\mathrm{Hom}(\mathrm{HomologyObject}(\alpha, \beta), I)$

The arguments are two morphisms $\alpha : A \rightarrow B, \beta : B \rightarrow C$. The output is the natural isomorphism from the homology object H of α and β to the construction of the homology object as $\mathrm{ImageObject}(\mathrm{PreCompose}(\mathrm{KernelEmbedding}(\beta), \mathrm{CokernelProjection}(\alpha)))$, denoted by I.

6.17.5 IsomorphismFromItsConstructionAsAnImageObjectToHomologyObject (for IsCapCategoryMorphism, IsCapCategoryMorphism)

▷ `IsomorphismFromItsConstructionAsAnImageObjectToHomologyObject(alpha, beta)` (operation)

Returns: a morphism in $\mathrm{Hom}(I, \mathrm{HomologyObject}(\alpha, \beta))$

The arguments are two morphisms $\alpha : A \rightarrow B, \beta : B \rightarrow C$. The output is the natural isomorphism from the construction of the homology object as $\mathrm{ImageObject}(\mathrm{PreCompose}(\mathrm{KernelEmbedding}(\beta), \mathrm{CokernelProjection}(\alpha)))$, denoted by I, to the homology object H of α and β.

6.18 Projective covers and injective envelopes

6.18.1 ProjectiveCoverObject (for IsCapCategoryObject)

▷ `ProjectiveCoverObject(A)` (attribute)

Returns: an object

The argument is an object A. The output is a projective cover of A.

6.18.2 EpimorphismFromProjectiveCoverObject (for IsCapCategoryObject)

▷ `EpimorphismFromProjectiveCoverObject(A)` (attribute)

Returns: an epimorphism

The argument is an object A. The output is an epimorphism from a projective cover of A.

6.18.3 EpimorphismFromProjectiveCoverObjectWithGivenProjectiveCoverObject (for IsCapCategoryObject, IsCapCategoryObject)

▷ `EpimorphismFromProjectiveCoverObjectWithGivenProjectiveCoverObject(A, P)` (operation)

Returns: an epimorphism

The argument is an object A. The output is the epimorphism from the projective cover P of A.

6.18.4 InjectiveEnvelopeObject (for IsCapCategoryObject)

▷ `InjectiveEnvelopeObject(A)` (attribute)

Returns: an object

The argument is an object A. The output is an injective envelope of A.

6.18.5 MonomorphismIntoInjectiveEnvelopeObject (for IsCapCategoryObject)

▷ `MonomorphismIntoInjectiveEnvelopeObject(A)` (attribute)

Returns: a monomorphism

The argument is an object A. The output is a monomorphism into an injective envelope of A.

6.18.6 MonomorphismIntoInjectiveEnvelopeObjectWithGivenInjectiveEnvelopeObject (for IsCapCategoryObject, IsCapCategoryObject)

▷ `MonomorphismIntoInjectiveEnvelopeObjectWithGivenInjectiveEnvelopeObject(A, I)` (operation)

Returns: a monomorphism

The argument is an object A. The output is a monomorphism into an injective envelope I of A.

Chapter 7

Add Functions

This section describes the overall structure of Add-functions and the functions installed by them.

7.1 Functions Installed by Add

Add functions have the following syntax:

Code
```
DeclareOperation( "AddSomeFunc",
                  [ IsCapCategory, IsList, IsInt ] );
```

The first argument is the category to which some function (e.g. `KernelObject`) is added, the second is a list containing pairs of functions and additional filters for the arguments, (e.g. if one argument is a morphism, an additional filter could be `IsMomomorphism`). The third is an optional weight which will then be the weight for `SomeFunc` (default value: 100). This is described later. If only one function is to be installed, the list can be replaced by the function. CAP installs the given function(s) as methods for `SomeFunc` (resp. `SomeFuncOp` if `SomeFunc` is not an operation).

All installed methods follow the following steps, described below:

- Redirect function
- Prefunction
- Function
- Logic
- Postfunction
- Addfunction

Every other part, except from function, does only depend on the name `SomeFunc`. We now explain the steps in detail.

- Redirect function: The redirect is used to redirect the computation from the given functions to some other symbol. If there is for example a with given method for some universal property, and the universal object is already computed, the redirect function might detect such a thing, calls the with given operation with the universal object as additional argument and then returns the value. In general, the redirect can be an arbitrary function. It is called with the same arguments as the operation `SomeFunc` itself and can return an array containing `[ true, something ]`, which will cause the installed method to simply return the object `something`, or `[ false ]`. If the output is `false`, the computation will continue with the step Prefunction.

- Prefunction: The prefunction should be used for error handling and plausibility checks of the input to `SomeFunc` (e.g. for `KernelLift` it should check wether range and source of the morphims coincide). Generally, the prefunction is defined in the method record and only depends on the name `SomeFunc`. It is called with the same input as the function itself, and should return either `[ true ]`, which continues the computation, or `[ false, "message" ]`, which will cause an error with message `"message"` and some additional information.

- Full prefunction: The full prefuction has the same semantics as the prefunction, but can perform additional, very costly checks. They are disabled by default.

- Function: This will launch the function(s) given as arguments. The result should be as specified in the type of `SomeFunc`. The resulting object is now named the result.

- Logic: For every function, some logical todos can be implemented in a logic texfile for the category. If there is some logic written down in a file belonging to the category, or belonging to some type of category. Please see the description of logic for more details. If there is some logic and some predicate relations for the function `SomeFunc`, it is installed in this step for the result.

- Postfunction: The postfunction called with the arguments of the function and the result. It can be an arbitrary function doing some cosmetics. If for example `SomeFunc` is `KernelEmbedding`, it will set the `KernelObject` of the input morphism to result. The postfunction is also taken from the method record and does only depend on the name `SomeFunc`.

- Addfunction: If the result is a category cell, it is added to the category for which the function was installed. This is disabled by default and can be enabled via `EnableAddForCategoricalOperations` (1.12.1).

7.2 Add Method

Except from installing a new method for the name `SomeFunc`, an Add method does slightly more. Every Add method has the same structure. The steps in the Add method are as follows:

- Default weight: If the weight parameter is `-1`, the default weight is assumed, which is 100.

- Weight check: If the current weight of the operation is lower than the given weight of the new functions, then the add function returns and installs nothing.

- Installation: Next, the method to install the functions is created. It creates the correct filter list, by merging the standard filters for the operation with the particular filters for the given functions, then installs the method as described above.

After calling an add method, the corresponding operation is available in the category. Also, some derivations, which are triggered by the setting of the primitive value, might be available.

7.3 Method name record entries

The entries of method name records can have the following components:

```
[
    "filter_list",
    "input_arguments_names",
    "return_type",
    "output_source_getter_string",
    "output_source_getter_preconditions",
    "output_range_getter_string",
    "output_range_getter_preconditions",
    "with_given_object_position",
    "dual_operation",
    "dual_arguments_reversed",
    "dual_with_given_objects_reversed",
    "dual_preprocessor_func",
    "dual_preprocessor_func_string",
    "dual_postprocessor_func",
    "dual_postprocessor_func_string",
    "functorial",
    "compatible_with_congruence_of_morphisms",
    "redirect_function",
    "pre_function",
    "pre_function_full",
    "post_function",
]
```

- pre_function (optional): A function which is used as the prefunction of the installed methods, as described above. Can also be the name of another operation. In this case the pre function of the referenced operation is used.

- pre_function_full (optional): A function which is used as the full prefunction of the installed methods, as described above. Can also be the name of another operation. In this case the full pre function of the referenced operation is used.

- redirect_function (optional): A function which is used as the redirect function of the installed methods, as described above. Can also be the name of another operation. In this case the redirect function of the referenced operation is used.

- post_function (optional): A function which is used as the postfunction of the installed methods, as described above.

- filter_list: A list containing the basic filters for the methods installed by the add methods. Possible entries are filters, or the strings listed below, which will be replaced by appropriate filters at the time the add method is called. The first entry of `filter_list` must be the string `category`. If the category can be inferred from the remaining arguments, a convenience method without the category as the first argument is installed automatically.

  - `category`,

- `object`,
- `morphism`,
- `twocell`,
- `object_in_range_category_of_homomorphism_structure`,
- `morphism_in_range_category_of_homomorphism_structure`,
- `list_of_objects`,
- `list_of_morphisms`,
- `list_of_twocells`.

- return_type: The return type can either be a filter or one of the strings in the list below. For objects, morphisms and 2-cells the correct `Add` function (see above) is used for the result of the computation. Otherwise, no `Add` function is used after all.

Code
```
[
    "object",
    "morphism",
    "twocell",
    "object_in_range_category_of_homomorphism_structure",
    "morphism_in_range_category_of_homomorphism_structure",
    "bool",
    "list_of_objects",
    "list_of_morphisms",
    "list_of_lists_of_morphisms",
    "object_datum",
    "morphism_datum",
    "nonneg_integer_or_infinity",
    "list_of_elements_of_commutative_semiring_of_linear_structure",
]
```

- functorial (optional): If an object has a corresponding functorial function, e.g., `KernelObject` and `KernelObjectFunctorial`, the name of the functorial is stored as a string.

- dual_operation (optional): Name of the dual operation.

- dual_arguments_reversed (optional): Boolean, marks whether for the call of the dual operation all arguments have to be given in reversed order.

- dual_with_given_objects_reversed (optional): Boolean, marks whether for the call of the dual operation the source and range of a with given operation have to be given in reversed order.

- dual_preprocessor_func[_string] (optional): let f be an operation with dual operation g. For the automatic installation of g from f, the arguments given to g are preprocessed by this given function. The function can also be given as a string.

- dual_postprocessor_func[_string] (optional): let f be an operation with dual operation g. For the automatic installation of g from f, the computed value of f is postprocessed by the given function. The function can also be given as a string.

- input_arguments_names (optional): A duplicate free list (of the same length as `filter_list`) of strings. For example, these strings will be used as the names of the arguments when automatically generating functions for this operation, e.g. in the opposite category.

- output_source_getter_string (optional): Only valid if the operation returns a morphism: a piece of GAP code which computes the source of the returned morphism. The input arguments are available via the names given in `input_arguments_names`.

- output_source_getter_preconditions (optional): Only valid if output_source_getter_string is set: The preconditions of output_source_getter_string in the same form as preconditions of derivations but with the CAP operations given as strings.

- output_range_getter_string (optional): Only valid if the operation returns a morphism: a piece of GAP code which computes the range of the returned morphism. The input arguments are available via the names given in `input_arguments_names`.

- output_range_getter_preconditions (optional): Only valid if output_range_getter_string is set: The preconditions of output_range_getter_string in the same form as preconditions of derivations but with the CAP operations given as strings.

- with_given_object_position (optional): One of the following strings: `"Source"`, `"Range"`, or `"both"`. Set for the without given operation in a with given pair. Describes whether the source resp. range are given (as the last argument of the with given operation) or both (as the second and the last argument of the with given operation).

- compatible_with_congruence_of_morphisms (optional): Indicates if the operation is compatible with the congruence of morphisms, that is, if the output does not change with regard to `IsEqualForObjects` and `IsCongruentForMorphisms` if the input changes with regard to `IsEqualForObjects` and `IsCongruentForMorphisms`.

7.4 Enhancing the method name record

The function CAP_INTERNAL_ENHANCE_NAME_RECORD can be applied to a method name record to make the following enhancements:

- Function name: Set the component function_name to the entry name.

- WithGiven special case: If the current entry belongs to a WithGiven operation or its without given pair, the with_given_without_given_name_pair is set. Additionally, the with given flag of the WithGiven operation is set to true.

- Redirect and post functions are created for all operations belonging to universal constructions (e.g. `KernelLift`) which are not a WithGiven operation.

7.5 Prepare functions

7.5.1 CAPOperationPrepareFunction

▷ `CAPOperationPrepareFunction(`*`prepare_function, category, func`*`)` (function)

Returns: a function

Given a non-CAP-conform function for any of the categorical operations, i.e., a function that computes the direct sum of two objects instead of a list of objects, this function wraps the function with a wrapper function to fit in the CAP context. For the mentioned binary direct sum one can call this function with `"BinaryDirectSumToDirectSum"` as *`prepare_function`*, the category, and the binary direct sum function. The function then returns a function that can be used for the direct sum categorical operation.

Note that *`func`* is not handled by the CAP caching mechanism and that the use of prepare functions is incompatible with `WithGiven` operations. Thus, one has to ensure manually that the equality and typing specifications are fulfilled.

7.5.2 CAPAddPrepareFunction

▷ `CAPAddPrepareFunction(`*`prepare_function, name, doc_string[, precondition_list]`*`)` (function)

Adds a prepare function to the list of CAP's prepare functions. The first argument is the prepare function itself. It should always be a function that takes a category and a function and returns a function. The argument *`name`* is the name of the prepare function, which is used in `CAPOperationPrepareFunction`. The argument *`doc_string`* should be a short string describing the functions. The optional argument *`precondition_list`* can describe preconditions for the prepare function to work, i.e., if the category does need to have PreCompose computable. This information is also recovered automatically from the prepare function itself, so the *`precondition_list`* is only necessary if the function needed is not explicitly used in the prepare function, e.g., if you use `+` instead of `AdditionForMorphisms`.

7.5.3 ListCAPPrepareFunctions

▷ `ListCAPPrepareFunctions(`*`arg`*`)` (function)

Lists all prepare functions.

7.6 Available Add functions

7.6.1 AddAdditionForMorphisms (for IsCapCategory, IsFunction)

▷ `AddAdditionForMorphisms(`*`C, F`*`)` (operation)

▷ `AddAdditionForMorphisms(`*`C, F, weight`*`)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `AdditionForMorphisms`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, beta) \mapsto$ `AdditionForMorphisms`$(alpha, beta)$.

7.6.2 AddAdditiveGenerators (for IsCapCategory, IsFunction)

▷ `AddAdditiveGenerators(C, F)` (operation)
▷ `AddAdditiveGenerators(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `AdditiveGenerators`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : () \mapsto$ `AdditiveGenerators`$()$.

7.6.3 AddAdditiveInverseForMorphisms (for IsCapCategory, IsFunction)

▷ `AddAdditiveInverseForMorphisms(C, F)` (operation)
▷ `AddAdditiveInverseForMorphisms(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `AdditiveInverseForMorphisms`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ `AdditiveInverseForMorphisms`$(alpha)$.

7.6.4 AddAstrictionToCoimage (for IsCapCategory, IsFunction)

▷ `AddAstrictionToCoimage(C, F)` (operation)
▷ `AddAstrictionToCoimage(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `AstrictionToCoimage`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ `AstrictionToCoimage`$(alpha)$.

7.6.5 AddAstrictionToCoimageWithGivenCoimageObject (for IsCapCategory, IsFunction)

▷ `AddAstrictionToCoimageWithGivenCoimageObject(C, F)` (operation)
▷ `AddAstrictionToCoimageWithGivenCoimageObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `AstrictionToCoimageWithGivenCoimageObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, C) \mapsto$ `AstrictionToCoimageWithGivenCoimageObject`$(alpha, C)$.

7.6.6 AddBasisOfExternalHom (for IsCapCategory, IsFunction)

▷ `AddBasisOfExternalHom(C, F)` (operation)
▷ `AddBasisOfExternalHom(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `BasisOfExternalHom`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `BasisOfExternalHom`$(arg2, arg3)$.

7.6.7 AddBasisOfSolutionsOfHomogeneousDoubleLinearSystemInLinearCategory (for IsCapCategory, IsFunction)

▷ `AddBasisOfSolutionsOfHomogeneousDoubleLinearSystemInLinearCategory(C, F)` (operation)
▷ `AddBasisOfSolutionsOfHomogeneousDoubleLinearSystemInLinearCategory(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `BasisOfSolutionsOfHomogeneousDoubleLinearSystemInLinearCategory`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3, arg4, arg5) \mapsto$ `BasisOfSolutionsOfHomogeneousDoubleLinearSystemInLinearCategory`$(arg2, arg3, arg4, arg5)$.

7.6.8 AddBasisOfSolutionsOfHomogeneousLinearSystemInLinearCategory (for IsCapCategory, IsFunction)

▷ `AddBasisOfSolutionsOfHomogeneousLinearSystemInLinearCategory(C, F)` (operation)
▷ `AddBasisOfSolutionsOfHomogeneousLinearSystemInLinearCategory(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `BasisOfSolutionsOfHomogeneousLinearSystemInLinearCategory`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `BasisOfSolutionsOfHomogeneousLinearSystemInLinearCategory`$(arg2, arg3)$.

7.6.9 AddCoastrictionToImage (for IsCapCategory, IsFunction)

▷ `AddCoastrictionToImage(C, F)` (operation)
▷ `AddCoastrictionToImage(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `CoastrictionToImage`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ `CoastrictionToImage`$(alpha)$.

7.6.10 AddCoastrictionToImageWithGivenImageObject (for IsCapCategory, IsFunction)

▷ `AddCoastrictionToImageWithGivenImageObject(C, F)` (operation)
▷ `AddCoastrictionToImageWithGivenImageObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `CoastrictionToImageWithGivenImageObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, I) \mapsto$ `CoastrictionToImageWithGivenImageObject`$(alpha, I)$.

7.6.11 AddCoefficientsOfMorphism (for IsCapCategory, IsFunction)

▷ `AddCoefficientsOfMorphism(C, F)` (operation)
▷ `AddCoefficientsOfMorphism(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `CoefficientsOfMorphism`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `CoefficientsOfMorphism`$(arg2)$.

7.6.12 AddCoequalizer (for IsCapCategory, IsFunction)

▷ `AddCoequalizer(C, F)` (operation)
▷ `AddCoequalizer(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `Coequalizer`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (Y, morphisms) \mapsto$ `Coequalizer`$(Y, morphisms)$.

7.6.13 AddCoequalizerFunctorial (for IsCapCategory, IsFunction)

▷ `AddCoequalizerFunctorial(C, F)` (operation)
▷ `AddCoequalizerFunctorial(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `CoequalizerFunctorial`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (morphisms, mu, morphismsp) \mapsto$ `CoequalizerFunctorial`$(morphisms, mu, morphismsp)$.

7.6.14 AddCoequalizerFunctorialWithGivenCoequalizers (for IsCapCategory, IsFunction)

▷ `AddCoequalizerFunctorialWithGivenCoequalizers(C, F)` (operation)
▷ `AddCoequalizerFunctorialWithGivenCoequalizers(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `CoequalizerFunctorialWithGivenCoequalizers`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (P, morphisms, mu, morphismsp, Pp) \mapsto$ `CoequalizerFunctorialWithGivenCoequalizers`$(P, morphisms, mu, morphismsp, Pp)$.

7.6.15 AddCoequalizerOfIdentityAndAutomorphisms (for IsCapCategory, IsFunction)

▷ `AddCoequalizerOfIdentityAndAutomorphisms(C, F)` (operation)

▷ `AddCoequalizerOfIdentityAndAutomorphisms(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `CoequalizerOfIdentityAndAutomorphisms`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (Y, automorphisms) \mapsto$ `CoequalizerOfIdentityAndAutomorphisms`$(Y, automorphisms)$.

7.6.16 AddCoequalizerOfIdentityAndAutomorphismsFunctorial (for IsCapCategory, IsFunction)

▷ `AddCoequalizerOfIdentityAndAutomorphismsFunctorial(C, F)` (operation)

▷ `AddCoequalizerOfIdentityAndAutomorphismsFunctorial(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `CoequalizerOfIdentityAndAutomorphismsFunctorial`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (automorphisms, mu, automorphismsp) \mapsto$ `CoequalizerOfIdentityAndAutomorphismsFunctorial`$(automorphisms, mu, automorphismsp)$.

7.6.17 AddCoequalizerOfIdentityAndAutomorphismsFunctorialWithGivenCoequalizers (for IsCapCategory, IsFunction)

▷ `AddCoequalizerOfIdentityAndAutomorphismsFunctorialWithGivenCoequalizers(C, F)` (operation)

▷ `AddCoequalizerOfIdentityAndAutomorphismsFunctorialWithGivenCoequalizers(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `CoequalizerOfIdentityAndAutomorphismsFunctorialWithGivenCoequalizers`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (P, automorphisms, mu, automorphismsp, Pp) \mapsto$ `CoequalizerOfIdentityAndAutomorphismsFunctorialWithGivenCoequalizers`$(P, automorphisms, mu, autom$

7.6.18 AddCoimageObject (for IsCapCategory, IsFunction)

▷ `AddCoimageObject(C, F)` (operation)

▷ `AddCoimageObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `CoimageObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `CoimageObject`$(arg2)$.

7.6.19 AddCoimageObjectFunctorial (for IsCapCategory, IsFunction)

▷ `AddCoimageObjectFunctorial(C, F)` (operation)

▷ `AddCoimageObjectFunctorial(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `CoimageObjectFunctorial`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, mu, alphap) \mapsto$ `CoimageObjectFunctorial`$(alpha, mu, alphap)$.

7.6.20 AddCoimageObjectFunctorialWithGivenCoimageObjects (for IsCapCategory, IsFunction)

▷ `AddCoimageObjectFunctorialWithGivenCoimageObjects(C, F)` (operation)

▷ `AddCoimageObjectFunctorialWithGivenCoimageObjects(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `CoimageObjectFunctorialWithGivenCoimageObjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (C, alpha, mu, alphap, Cp) \mapsto$ `CoimageObjectFunctorialWithGivenCoimageObjects`$(C, alpha, mu, alphap, Cp)$.

7.6.21 AddCoimageProjection (for IsCapCategory, IsFunction)

▷ `AddCoimageProjection(C, F)` (operation)

▷ `AddCoimageProjection(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `CoimageProjection`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ `CoimageProjection`$(alpha)$.

7.6.22 AddCoimageProjectionWithGivenCoimageObject (for IsCapCategory, IsFunction)

▷ `AddCoimageProjectionWithGivenCoimageObject(C, F)` (operation)
▷ `AddCoimageProjectionWithGivenCoimageObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `CoimageProjectionWithGivenCoimageObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, C) \mapsto$ `CoimageProjectionWithGivenCoimageObject`$(alpha, C)$.

7.6.23 AddCokernelColift (for IsCapCategory, IsFunction)

▷ `AddCokernelColift(C, F)` (operation)
▷ `AddCokernelColift(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `CokernelColift`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, T, tau) \mapsto$ `CokernelColift`$(alpha, T, tau)$.

7.6.24 AddCokernelColiftWithGivenCokernelObject (for IsCapCategory, IsFunction)

▷ `AddCokernelColiftWithGivenCokernelObject(C, F)` (operation)
▷ `AddCokernelColiftWithGivenCokernelObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `CokernelColiftWithGivenCokernelObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, T, tau, P) \mapsto$ `CokernelColiftWithGivenCokernelObject`$(alpha, T, tau, P)$.

7.6.25 AddCokernelObject (for IsCapCategory, IsFunction)

▷ `AddCokernelObject(C, F)` (operation)
▷ `AddCokernelObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `CokernelObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ `CokernelObject`$(alpha)$.

7.6.26 AddCokernelObjectFunctorial (for IsCapCategory, IsFunction)

▷ `AddCokernelObjectFunctorial(C, F)` (operation)
▷ `AddCokernelObjectFunctorial(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `CokernelObjectFunctorial`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, mu, alphap) \mapsto$ `CokernelObjectFunctorial`$(alpha, mu, alphap)$.

7.6.27 AddCokernelObjectFunctorialWithGivenCokernelObjects (for IsCapCategory, IsFunction)

▷ `AddCokernelObjectFunctorialWithGivenCokernelObjects(C, F)` (operation)

▷ `AddCokernelObjectFunctorialWithGivenCokernelObjects(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `CokernelObjectFunctorialWithGivenCokernelObjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (P, alpha, mu, alphap, Pp) \mapsto$ `CokernelObjectFunctorialWithGivenCokernelObjects`$(P, alpha, mu, alphap, Pp)$.

7.6.28 AddCokernelProjection (for IsCapCategory, IsFunction)

▷ `AddCokernelProjection(C, F)` (operation)

▷ `AddCokernelProjection(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `CokernelProjection`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ `CokernelProjection`$(alpha)$.

7.6.29 AddCokernelProjectionWithGivenCokernelObject (for IsCapCategory, IsFunction)

▷ `AddCokernelProjectionWithGivenCokernelObject(C, F)` (operation)

▷ `AddCokernelProjectionWithGivenCokernelObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `CokernelProjectionWithGivenCokernelObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, P) \mapsto$ `CokernelProjectionWithGivenCokernelObject`$(alpha, P)$.

7.6.30 AddColift (for IsCapCategory, IsFunction)

▷ `AddColift(C, F)` (operation)

▷ `AddColift(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `Colift`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, beta) \mapsto$ `Colift`$(alpha, beta)$.

7.6.31 AddColiftAlongEpimorphism (for IsCapCategory, IsFunction)

▷ `AddColiftAlongEpimorphism(C, F)` (operation)
▷ `AddColiftAlongEpimorphism(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ColiftAlongEpimorphism`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (epsilon, tau) \mapsto$ `ColiftAlongEpimorphism`$(epsilon, tau)$.

7.6.32 AddComponentOfMorphismFromCoproduct (for IsCapCategory, IsFunction)

▷ `AddComponentOfMorphismFromCoproduct(C, F)` (operation)
▷ `AddComponentOfMorphismFromCoproduct(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ComponentOfMorphismFromCoproduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, I, i) \mapsto$ `ComponentOfMorphismFromCoproduct`$(alpha, I, i)$.

7.6.33 AddComponentOfMorphismFromDirectSum (for IsCapCategory, IsFunction)

▷ `AddComponentOfMorphismFromDirectSum(C, F)` (operation)
▷ `AddComponentOfMorphismFromDirectSum(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ComponentOfMorphismFromDirectSum`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, S, i) \mapsto$ `ComponentOfMorphismFromDirectSum`$(alpha, S, i)$.

7.6.34 AddComponentOfMorphismIntoDirectProduct (for IsCapCategory, IsFunction)

▷ `AddComponentOfMorphismIntoDirectProduct(C, F)` (operation)
▷ `AddComponentOfMorphismIntoDirectProduct(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ComponentOfMorphismIntoDirectProduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, P, i) \mapsto$ `ComponentOfMorphismIntoDirectProduct`$(alpha, P, i)$.

7.6.35 AddComponentOfMorphismIntoDirectSum (for IsCapCategory, IsFunction)

▷ `AddComponentOfMorphismIntoDirectSum(C, F)` (operation)
▷ `AddComponentOfMorphismIntoDirectSum(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ComponentOfMorphismIntoDirectSum`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, S, i) \mapsto$ `ComponentOfMorphismIntoDirectSum`$(alpha, S, i)$.

7.6.36 AddCoproduct (for IsCapCategory, IsFunction)

▷ `AddCoproduct(C, F)` (operation)
▷ `AddCoproduct(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `Coproduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (objects) \mapsto$ `Coproduct`$(objects)$.

7.6.37 AddCoproductFunctorial (for IsCapCategory, IsFunction)

▷ `AddCoproductFunctorial(C, F)` (operation)
▷ `AddCoproductFunctorial(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `CoproductFunctorial`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (objects, L, objectsp) \mapsto$ `CoproductFunctorial`$(objects, L, objectsp)$.

7.6.38 AddCoproductFunctorialWithGivenCoproducts (for IsCapCategory, IsFunction)

▷ `AddCoproductFunctorialWithGivenCoproducts(C, F)` (operation)
▷ `AddCoproductFunctorialWithGivenCoproducts(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `CoproductFunctorialWithGivenCoproducts`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (P, objects, L, objectsp, Pp) \mapsto$ `CoproductFunctorialWithGivenCoproducts`$(P, objects, L, objectsp, Pp)$.

7.6.39 AddDirectProduct (for IsCapCategory, IsFunction)

▷ `AddDirectProduct(C, F)` (operation)
▷ `AddDirectProduct(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `DirectProduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (objects) \mapsto \texttt{DirectProduct}(objects)$.

7.6.40 AddDirectProductFunctorial (for IsCapCategory, IsFunction)

▷ `AddDirectProductFunctorial(C, F)` (operation)
▷ `AddDirectProductFunctorial(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `DirectProductFunctorial`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (objects, L, objectsp) \mapsto \texttt{DirectProductFunctorial}(objects, L, objectsp)$.

7.6.41 AddDirectProductFunctorialWithGivenDirectProducts (for IsCapCategory, IsFunction)

▷ `AddDirectProductFunctorialWithGivenDirectProducts(C, F)` (operation)
▷ `AddDirectProductFunctorialWithGivenDirectProducts(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `DirectProductFunctorialWithGivenDirectProducts`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (P, objects, L, objectsp, Pp) \mapsto \texttt{DirectProductFunctorialWithGivenDirectProducts}(P, objects, L, objectsp, Pp)$.

7.6.42 AddDirectSum (for IsCapCategory, IsFunction)

▷ `AddDirectSum(C, F)` (operation)
▷ `AddDirectSum(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `DirectSum`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (objects) \mapsto \texttt{DirectSum}(objects)$.

7.6.43 AddDirectSumFunctorial (for IsCapCategory, IsFunction)

▷ `AddDirectSumFunctorial(C, F)` (operation)
▷ `AddDirectSumFunctorial(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `DirectSumFunctorial`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (objects, L, objectsp) \mapsto$ `DirectSumFunctorial`$(objects, L, objectsp)$.

7.6.44 AddDirectSumFunctorialWithGivenDirectSums (for IsCapCategory, IsFunction)

▷ `AddDirectSumFunctorialWithGivenDirectSums(C, F)` (operation)

▷ `AddDirectSumFunctorialWithGivenDirectSums(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `DirectSumFunctorialWithGivenDirectSums`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (P, objects, L, objectsp, Pp) \mapsto$ `DirectSumFunctorialWithGivenDirectSums`$(P, objects, L, objectsp, Pp)$.

7.6.45 AddDistinguishedObjectOfHomomorphismStructure (for IsCapCategory, IsFunction)

▷ `AddDistinguishedObjectOfHomomorphismStructure(C, F)` (operation)

▷ `AddDistinguishedObjectOfHomomorphismStructure(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `DistinguishedObjectOfHomomorphismStructure`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : () \mapsto$ `DistinguishedObjectOfHomomorphismStructure`$()$.

7.6.46 AddEmbeddingOfEqualizer (for IsCapCategory, IsFunction)

▷ `AddEmbeddingOfEqualizer(C, F)` (operation)

▷ `AddEmbeddingOfEqualizer(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `EmbeddingOfEqualizer`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (Y, morphisms) \mapsto$ `EmbeddingOfEqualizer`$(Y, morphisms)$.

7.6.47 AddEmbeddingOfEqualizerWithGivenEqualizer (for IsCapCategory, IsFunction)

▷ `AddEmbeddingOfEqualizerWithGivenEqualizer(C, F)` (operation)

▷ `AddEmbeddingOfEqualizerWithGivenEqualizer(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `EmbeddingOfEqualizerWithGivenEqualizer`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (Y, morphisms, P) \mapsto$ `EmbeddingOfEqualizerWithGivenEqualizer`$(Y, morphisms, P)$.

7.6.48 AddEpimorphismFromProjectiveCoverObject (for IsCapCategory, IsFunction)

▷ `AddEpimorphismFromProjectiveCoverObject(C, F)` (operation)
▷ `AddEpimorphismFromProjectiveCoverObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `EpimorphismFromProjectiveCoverObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (A) \mapsto$ `EpimorphismFromProjectiveCoverObject`(A).

7.6.49 AddEpimorphismFromProjectiveCoverObjectWithGivenProjectiveCoverObject (for IsCapCategory, IsFunction)

▷ `AddEpimorphismFromProjectiveCoverObjectWithGivenProjectiveCoverObject(C, F)` (operation)
▷ `AddEpimorphismFromProjectiveCoverObjectWithGivenProjectiveCoverObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `EpimorphismFromProjectiveCoverObjectWithGivenProjectiveCoverObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (A, P) \mapsto$ `EpimorphismFromProjectiveCoverObjectWithGivenProjectiveCoverObject`(A, P).

7.6.50 AddEpimorphismFromSomeProjectiveObject (for IsCapCategory, IsFunction)

▷ `AddEpimorphismFromSomeProjectiveObject(C, F)` (operation)
▷ `AddEpimorphismFromSomeProjectiveObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `EpimorphismFromSomeProjectiveObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (A) \mapsto$ `EpimorphismFromSomeProjectiveObject`(A).

7.6.51 AddEpimorphismFromSomeProjectiveObjectWithGivenSomeProjectiveObject (for IsCapCategory, IsFunction)

▷ `AddEpimorphismFromSomeProjectiveObjectWithGivenSomeProjectiveObject(C, F)` (operation)

▷ `AddEpimorphismFromSomeProjectiveObjectWithGivenSomeProjectiveObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `EpimorphismFromSomeProjectiveObjectWithGivenSomeProjectiveObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (A,P) \mapsto$ `EpimorphismFromSomeProjectiveObjectWithGivenSomeProjectiveObject`(A,P).

7.6.52 AddEqualizer (for IsCapCategory, IsFunction)

▷ `AddEqualizer(C, F)` (operation)

▷ `AddEqualizer(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `Equalizer`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (Y, morphisms) \mapsto$ `Equalizer`$(Y, morphisms)$.

7.6.53 AddEqualizerFunctorial (for IsCapCategory, IsFunction)

▷ `AddEqualizerFunctorial(C, F)` (operation)

▷ `AddEqualizerFunctorial(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `EqualizerFunctorial`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (morphisms, mu, morphismsp) \mapsto$ `EqualizerFunctorial`$(morphisms, mu, morphismsp)$.

7.6.54 AddEqualizerFunctorialWithGivenEqualizers (for IsCapCategory, IsFunction)

▷ `AddEqualizerFunctorialWithGivenEqualizers(C, F)` (operation)

▷ `AddEqualizerFunctorialWithGivenEqualizers(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `EqualizerFunctorialWithGivenEqualizers`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (P, morphisms, mu, morphismsp, Pp) \mapsto$ `EqualizerFunctorialWithGivenEqualizers`$(P, morphisms, mu, morphismsp, Pp)$.

7.6.55 AddFiberProduct (for IsCapCategory, IsFunction)

▷ `AddFiberProduct(C, F)` (operation)
▷ `AddFiberProduct(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `FiberProduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (morphisms) \mapsto$ `FiberProduct`$(morphisms)$.

7.6.56 AddFiberProductFunctorial (for IsCapCategory, IsFunction)

▷ `AddFiberProductFunctorial(C, F)` (operation)
▷ `AddFiberProductFunctorial(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `FiberProductFunctorial`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (morphisms, L, morphismsp) \mapsto$ `FiberProductFunctorial`$(morphisms, L, morphismsp)$.

7.6.57 AddFiberProductFunctorialWithGivenFiberProducts (for IsCapCategory, IsFunction)

▷ `AddFiberProductFunctorialWithGivenFiberProducts(C, F)` (operation)
▷ `AddFiberProductFunctorialWithGivenFiberProducts(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `FiberProductFunctorialWithGivenFiberProducts`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (P, morphisms, L, morphismsp, Pp) \mapsto$ `FiberProductFunctorialWithGivenFiberProducts`$(P, morphisms, L, morphismsp, Pp)$.

7.6.58 AddHomologyObject (for IsCapCategory, IsFunction)

▷ `AddHomologyObject(C, F)` (operation)
▷ `AddHomologyObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `HomologyObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, beta) \mapsto$ `HomologyObject`$(alpha, beta)$.

7.6.59 AddHomologyObjectFunctorialWithGivenHomologyObjects (for IsCapCategory, IsFunction)

▷ `AddHomologyObjectFunctorialWithGivenHomologyObjects(C, F)` (operation)
▷ `AddHomologyObjectFunctorialWithGivenHomologyObjects(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `HomologyObjectFunctorialWithGivenHomologyObjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (H_1, L, H_2) \mapsto$ `HomologyObjectFunctorialWithGivenHomologyObjects`(H_1, L, H_2).

7.6.60 AddHomomorphismStructureOnMorphisms (for IsCapCategory, IsFunction)

▷ `AddHomomorphismStructureOnMorphisms(C, F)` (operation)
▷ `AddHomomorphismStructureOnMorphisms(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `HomomorphismStructureOnMorphisms`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, beta) \mapsto$ `HomomorphismStructureOnMorphisms`$(alpha, beta)$.

7.6.61 AddHomomorphismStructureOnMorphismsWithGivenObjects (for IsCapCategory, IsFunction)

▷ `AddHomomorphismStructureOnMorphismsWithGivenObjects(C, F)` (operation)
▷ `AddHomomorphismStructureOnMorphismsWithGivenObjects(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `HomomorphismStructureOnMorphismsWithGivenObjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (source, alpha, beta, range) \mapsto$ `HomomorphismStructureOnMorphismsWithGivenObjects`$(source, alpha, beta, range)$.

7.6.62 AddHomomorphismStructureOnObjects (for IsCapCategory, IsFunction)

▷ `AddHomomorphismStructureOnObjects(C, F)` (operation)
▷ `AddHomomorphismStructureOnObjects(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `HomomorphismStructureOnObjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `HomomorphismStructureOnObjects`$(arg2, arg3)$.

7.6.63 AddHorizontalPostCompose (for IsCapCategory, IsFunction)

▷ `AddHorizontalPostCompose(C, F)` (operation)
▷ `AddHorizontalPostCompose(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `HorizontalPostCompose`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `HorizontalPostCompose`$(arg2, arg3)$.

7.6.64 AddHorizontalPreCompose (for IsCapCategory, IsFunction)

▷ `AddHorizontalPreCompose(C, F)` (operation)
▷ `AddHorizontalPreCompose(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `HorizontalPreCompose`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `HorizontalPreCompose`$(arg2, arg3)$.

7.6.65 AddIdentityMorphism (for IsCapCategory, IsFunction)

▷ `AddIdentityMorphism(C, F)` (operation)
▷ `AddIdentityMorphism(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IdentityMorphism`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (a) \mapsto$ `IdentityMorphism`(a).

7.6.66 AddIdentityTwoCell (for IsCapCategory, IsFunction)

▷ `AddIdentityTwoCell(C, F)` (operation)
▷ `AddIdentityTwoCell(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IdentityTwoCell`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `IdentityTwoCell`$(arg2)$.

7.6.67 AddImageEmbedding (for IsCapCategory, IsFunction)

▷ `AddImageEmbedding(C, F)` (operation)
▷ `AddImageEmbedding(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ImageEmbedding`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ `ImageEmbedding`$(alpha)$.

7.6.68 AddImageEmbeddingWithGivenImageObject (for IsCapCategory, IsFunction)

▷ `AddImageEmbeddingWithGivenImageObject(C, F)` (operation)
▷ `AddImageEmbeddingWithGivenImageObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ImageEmbeddingWithGivenImageObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, I) \mapsto$ `ImageEmbeddingWithGivenImageObject`$(alpha, I)$.

7.6.69 AddImageObject (for IsCapCategory, IsFunction)

▷ `AddImageObject(C, F)` (operation)
▷ `AddImageObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ImageObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `ImageObject`$(arg2)$.

7.6.70 AddImageObjectFunctorial (for IsCapCategory, IsFunction)

▷ `AddImageObjectFunctorial(C, F)` (operation)
▷ `AddImageObjectFunctorial(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ImageObjectFunctorial`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, nu, alphap) \mapsto$ `ImageObjectFunctorial`$(alpha, nu, alphap)$.

7.6.71 AddImageObjectFunctorialWithGivenImageObjects (for IsCapCategory, IsFunction)

▷ `AddImageObjectFunctorialWithGivenImageObjects(C, F)` (operation)
▷ `AddImageObjectFunctorialWithGivenImageObjects(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ImageObjectFunctorialWithGivenImageObjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (I, alpha, nu, alphap, Ip) \mapsto$ `ImageObjectFunctorialWithGivenImageObjects`$(I, alpha, nu, alphap, Ip)$.

7.6.72 AddIndecomposableInjectiveObjects (for IsCapCategory, IsFunction)

▷ `AddIndecomposableInjectiveObjects(C, F)` (operation)
▷ `AddIndecomposableInjectiveObjects(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IndecomposableInjectiveObjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : () \mapsto$ `IndecomposableInjectiveObjects`().

7.6.73 AddIndecomposableProjectiveObjects (for IsCapCategory, IsFunction)

▷ `AddIndecomposableProjectiveObjects(C, F)` (operation)
▷ `AddIndecomposableProjectiveObjects(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IndecomposableProjectiveObjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : () \mapsto$ `IndecomposableProjectiveObjects`().

7.6.74 AddInitialObject (for IsCapCategory, IsFunction)

▷ `AddInitialObject(C, F)` (operation)
▷ `AddInitialObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `InitialObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : () \mapsto$ `InitialObject`().

7.6.75 AddInitialObjectFunctorial (for IsCapCategory, IsFunction)

▷ `AddInitialObjectFunctorial(C, F)` (operation)
▷ `AddInitialObjectFunctorial(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `InitialObjectFunctorial`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : () \mapsto$ `InitialObjectFunctorial`().

7.6.76 AddInitialObjectFunctorialWithGivenInitialObjects (for IsCapCategory, IsFunction)

▷ `AddInitialObjectFunctorialWithGivenInitialObjects(C, F)` (operation)
▷ `AddInitialObjectFunctorialWithGivenInitialObjects(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `InitialObjectFunctorialWithGivenInitialObjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (P, Pp) \mapsto$ `InitialObjectFunctorialWithGivenInitialObjects`(P, Pp).

7.6.77 AddInjectionOfCofactorOfCoproduct (for IsCapCategory, IsFunction)

▷ `AddInjectionOfCofactorOfCoproduct(C, F)` (operation)
▷ `AddInjectionOfCofactorOfCoproduct(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `InjectionOfCofactorOfCoproduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (objects, k) \mapsto$ `InjectionOfCofactorOfCoproduct`$(objects, k)$.

7.6.78 AddInjectionOfCofactorOfCoproductWithGivenCoproduct (for IsCapCategory, IsFunction)

▷ `AddInjectionOfCofactorOfCoproductWithGivenCoproduct(C, F)` (operation)
▷ `AddInjectionOfCofactorOfCoproductWithGivenCoproduct(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `InjectionOfCofactorOfCoproductWithGivenCoproduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (objects, k, P) \mapsto$ `InjectionOfCofactorOfCoproductWithGivenCoproduct`$(objects, k, P)$.

7.6.79 AddInjectionOfCofactorOfDirectSum (for IsCapCategory, IsFunction)

▷ `AddInjectionOfCofactorOfDirectSum(C, F)` (operation)
▷ `AddInjectionOfCofactorOfDirectSum(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `InjectionOfCofactorOfDirectSum`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (objects, k) \mapsto$ `InjectionOfCofactorOfDirectSum`$(objects, k)$.

7.6.80 AddInjectionOfCofactorOfDirectSumWithGivenDirectSum (for IsCapCategory, IsFunction)

▷ `AddInjectionOfCofactorOfDirectSumWithGivenDirectSum(C, F)` (operation)
▷ `AddInjectionOfCofactorOfDirectSumWithGivenDirectSum(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `InjectionOfCofactorOfDirectSumWithGivenDirectSum`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (objects, k, P) \mapsto$ `InjectionOfCofactorOfDirectSumWithGivenDirectSum`$(objects, k, P)$.

7.6.81 AddInjectionOfCofactorOfPushout (for IsCapCategory, IsFunction)

▷ `AddInjectionOfCofactorOfPushout(C, F)` (operation)
▷ `AddInjectionOfCofactorOfPushout(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `InjectionOfCofactorOfPushout`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (morphisms, k) \mapsto$ `InjectionOfCofactorOfPushout`$(morphisms, k)$.

7.6.82 AddInjectionOfCofactorOfPushoutWithGivenPushout (for IsCapCategory, IsFunction)

▷ `AddInjectionOfCofactorOfPushoutWithGivenPushout(C, F)` (operation)
▷ `AddInjectionOfCofactorOfPushoutWithGivenPushout(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `InjectionOfCofactorOfPushoutWithGivenPushout`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (morphisms, k, P) \mapsto$ `InjectionOfCofactorOfPushoutWithGivenPushout`$(morphisms, k, P)$.

7.6.83 AddInjectiveColift (for IsCapCategory, IsFunction)

▷ `AddInjectiveColift(C, F)` (operation)
▷ `AddInjectiveColift(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `InjectiveColift`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, beta) \mapsto$ `InjectiveColift`$(alpha, beta)$.

7.6.84 AddInjectiveDimension (for IsCapCategory, IsFunction)

▷ `AddInjectiveDimension(C, F)` (operation)
▷ `AddInjectiveDimension(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `InjectiveDimension`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `InjectiveDimension`$(arg2)$.

7.6.85 AddInjectiveEnvelopeObject (for IsCapCategory, IsFunction)

▷ `AddInjectiveEnvelopeObject(C, F)` (operation)
▷ `AddInjectiveEnvelopeObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `InjectiveEnvelopeObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `InjectiveEnvelopeObject`$(arg2)$.

7.6.86 AddInterpretMorphismAsMorphismFromDistinguishedObjectToHomomorphismStructure (for IsCapCategory, IsFunction)

▷ `AddInterpretMorphismAsMorphismFromDistinguishedObjectToHomomorphismStructure(C, F)` (operation)
▷ `AddInterpretMorphismAsMorphismFromDistinguishedObjectToHomomorphismStructure(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `InterpretMorphismAsMorphismFromDistinguishedObjectToHomomorphismStructure`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ `InterpretMorphismAsMorphismFromDistinguishedObjectToHomomorphismStructure`$(alpha)$.

7.6.87 AddInterpretMorphismAsMorphismFromDistinguishedObjectToHomomorphismStructureW (for IsCapCategory, IsFunction)

▷ `AddInterpretMorphismAsMorphismFromDistinguishedObjectToHomomorphismStructureWithGivenObjec F)` (operation)
▷ `AddInterpretMorphismAsMorphismFromDistinguishedObjectToHomomorphismStructureWithGivenObjec F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `InterpretMorphismAsMorphismFromDistinguishedObjectToHomomorphismStructureWithGivenObjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (source, alpha, range) \mapsto$ `InterpretMorphismAsMorphismFromDistinguishedObjectToHomomorphismStructureWithGivenObjects`$(s$

7.6.88 AddInterpretMorphismFromDistinguishedObjectToHomomorphismStructureAsMorphism (for IsCapCategory, IsFunction)

▷ `AddInterpretMorphismFromDistinguishedObjectToHomomorphismStructureAsMorphism(C, F)` (operation)
▷ `AddInterpretMorphismFromDistinguishedObjectToHomomorphismStructureAsMorphism(C,`

`F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `InterpretMorphismFromDistinguishedObjectToHomomorphismStructureAsMorphism`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (source, range, alpha) \mapsto$ `InterpretMorphismFromDistinguishedObjectToHomomorphismStructureAsMorphism`$(source, range, alpha)$.

7.6.89 AddInverseForMorphisms (for IsCapCategory, IsFunction)

▷ `AddInverseForMorphisms(C, F)` (operation)

▷ `AddInverseForMorphisms(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `InverseForMorphisms`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ `InverseForMorphisms`$(alpha)$.

7.6.90 AddInverseOfMorphismFromCoimageToImage (for IsCapCategory, IsFunction)

▷ `AddInverseOfMorphismFromCoimageToImage(C, F)` (operation)

▷ `AddInverseOfMorphismFromCoimageToImage(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `InverseOfMorphismFromCoimageToImage`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ `InverseOfMorphismFromCoimageToImage`$(alpha)$.

7.6.91 AddInverseOfMorphismFromCoimageToImageWithGivenObjects (for IsCapCategory, IsFunction)

▷ `AddInverseOfMorphismFromCoimageToImageWithGivenObjects(C, F)` (operation)

▷ `AddInverseOfMorphismFromCoimageToImageWithGivenObjects(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `InverseOfMorphismFromCoimageToImageWithGivenObjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (I, alpha, C) \mapsto$ `InverseOfMorphismFromCoimageToImageWithGivenObjects`$(I, alpha, C)$.

7.6.92 AddIsAutomorphism (for IsCapCategory, IsFunction)

▷ `AddIsAutomorphism(C, F)` (operation)

▷ `AddIsAutomorphism(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsAutomorphism`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `IsAutomorphism`$(arg2)$.

7.6.93 AddIsBijectiveObject (for IsCapCategory, IsFunction)

▷ `AddIsBijectiveObject(C, F)` (operation)

▷ `AddIsBijectiveObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsBijectiveObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `IsBijectiveObject`$(arg2)$.

7.6.94 AddIsCodominating (for IsCapCategory, IsFunction)

▷ `AddIsCodominating(C, F)` (operation)

▷ `AddIsCodominating(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsCodominating`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `IsCodominating`$(arg2, arg3)$.

7.6.95 AddIsColiftable (for IsCapCategory, IsFunction)

▷ `AddIsColiftable(C, F)` (operation)

▷ `AddIsColiftable(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsColiftable`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `IsColiftable`$(arg2, arg3)$.

7.6.96 AddIsColiftableAlongEpimorphism (for IsCapCategory, IsFunction)

▷ `AddIsColiftableAlongEpimorphism(C, F)` (operation)

▷ `AddIsColiftableAlongEpimorphism(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsColiftableAlongEpimorphism`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `IsColiftableAlongEpimorphism`$(arg2, arg3)$.

7.6.97 AddIsCongruentForMorphisms (for IsCapCategory, IsFunction)

▷ `AddIsCongruentForMorphisms(C, F)` (operation)
▷ `AddIsCongruentForMorphisms(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsCongruentForMorphisms`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `IsCongruentForMorphisms`$(arg2, arg3)$.

7.6.98 AddIsDominating (for IsCapCategory, IsFunction)

▷ `AddIsDominating(C, F)` (operation)
▷ `AddIsDominating(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsDominating`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `IsDominating`$(arg2, arg3)$.

7.6.99 AddIsEndomorphism (for IsCapCategory, IsFunction)

▷ `AddIsEndomorphism(C, F)` (operation)
▷ `AddIsEndomorphism(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsEndomorphism`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `IsEndomorphism`$(arg2)$.

7.6.100 AddIsEpimorphism (for IsCapCategory, IsFunction)

▷ `AddIsEpimorphism(C, F)` (operation)
▷ `AddIsEpimorphism(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsEpimorphism`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `IsEpimorphism`$(arg2)$.

7.6.101 AddIsEqualAsFactorobjects (for IsCapCategory, IsFunction)

▷ `AddIsEqualAsFactorobjects(C, F)` (operation)
▷ `AddIsEqualAsFactorobjects(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsEqualAsFactorobjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `IsEqualAsFactorobjects`$(arg2, arg3)$.

7.6.102 AddIsEqualAsSubobjects (for IsCapCategory, IsFunction)

▷ `AddIsEqualAsSubobjects(C, F)` (operation)
▷ `AddIsEqualAsSubobjects(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsEqualAsSubobjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `IsEqualAsSubobjects`$(arg2, arg3)$.

7.6.103 AddIsEqualForCacheForMorphisms (for IsCapCategory, IsFunction)

▷ `AddIsEqualForCacheForMorphisms(C, F)` (operation)
▷ `AddIsEqualForCacheForMorphisms(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsEqualForCacheForMorphisms`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `IsEqualForCacheForMorphisms`$(arg2, arg3)$.

7.6.104 AddIsEqualForCacheForObjects (for IsCapCategory, IsFunction)

▷ `AddIsEqualForCacheForObjects(C, F)` (operation)
▷ `AddIsEqualForCacheForObjects(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsEqualForCacheForObjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `IsEqualForCacheForObjects`$(arg2, arg3)$.

7.6.105 AddIsEqualForMorphisms (for IsCapCategory, IsFunction)

▷ `AddIsEqualForMorphisms(C, F)` (operation)
▷ `AddIsEqualForMorphisms(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsEqualForMorphisms`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `IsEqualForMorphisms`$(arg2, arg3)$.

7.6.106 AddIsEqualForMorphismsOnMor (for IsCapCategory, IsFunction)

▷ `AddIsEqualForMorphismsOnMor(C, F)` (operation)

▷ `AddIsEqualForMorphismsOnMor(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsEqualForMorphismsOnMor`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `IsEqualForMorphismsOnMor`$(arg2, arg3)$.

7.6.107 AddIsEqualForObjects (for IsCapCategory, IsFunction)

▷ `AddIsEqualForObjects(C, F)` (operation)

▷ `AddIsEqualForObjects(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsEqualForObjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `IsEqualForObjects`$(arg2, arg3)$.

7.6.108 AddIsEqualToIdentityMorphism (for IsCapCategory, IsFunction)

▷ `AddIsEqualToIdentityMorphism(C, F)` (operation)

▷ `AddIsEqualToIdentityMorphism(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsEqualToIdentityMorphism`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `IsEqualToIdentityMorphism`$(arg2)$.

7.6.109 AddIsEqualToZeroMorphism (for IsCapCategory, IsFunction)

▷ `AddIsEqualToZeroMorphism(C, F)` (operation)

▷ `AddIsEqualToZeroMorphism(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsEqualToZeroMorphism`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `IsEqualToZeroMorphism`$(arg2)$.

7.6.110 AddIsHomSetInhabited (for IsCapCategory, IsFunction)

▷ `AddIsHomSetInhabited(C, F)` (operation)

▷ `AddIsHomSetInhabited(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsHomSetInhabited`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `IsHomSetInhabited`$(arg2, arg3)$.

7.6.111 AddIsIdempotent (for IsCapCategory, IsFunction)

▷ `AddIsIdempotent(C, F)` (operation)
▷ `AddIsIdempotent(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsIdempotent`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `IsIdempotent`$(arg2)$.

7.6.112 AddIsInitial (for IsCapCategory, IsFunction)

▷ `AddIsInitial(C, F)` (operation)
▷ `AddIsInitial(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsInitial`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `IsInitial`$(arg2)$.

7.6.113 AddIsInjective (for IsCapCategory, IsFunction)

▷ `AddIsInjective(C, F)` (operation)
▷ `AddIsInjective(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsInjective`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `IsInjective`$(arg2)$.

7.6.114 AddIsIsomorphicForObjects (for IsCapCategory, IsFunction)

▷ `AddIsIsomorphicForObjects(C, F)` (operation)
▷ `AddIsIsomorphicForObjects(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsIsomorphicForObjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (object_1, object_2) \mapsto$ `IsIsomorphicForObjects`$(object_1, object_2)$.

7.6.115 AddIsIsomorphism (for IsCapCategory, IsFunction)

▷ `AddIsIsomorphism(C, F)` (operation)
▷ `AddIsIsomorphism(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsIsomorphism`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `IsIsomorphism`$(arg2)$.

7.6.116 AddIsLiftable (for IsCapCategory, IsFunction)

▷ `AddIsLiftable(C, F)` (operation)
▷ `AddIsLiftable(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsLiftable`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `IsLiftable`$(arg2, arg3)$.

7.6.117 AddIsLiftableAlongMonomorphism (for IsCapCategory, IsFunction)

▷ `AddIsLiftableAlongMonomorphism(C, F)` (operation)
▷ `AddIsLiftableAlongMonomorphism(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsLiftableAlongMonomorphism`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `IsLiftableAlongMonomorphism`$(arg2, arg3)$.

7.6.118 AddIsMonomorphism (for IsCapCategory, IsFunction)

▷ `AddIsMonomorphism(C, F)` (operation)
▷ `AddIsMonomorphism(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsMonomorphism`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `IsMonomorphism`$(arg2)$.

7.6.119 AddIsOne (for IsCapCategory, IsFunction)

▷ `AddIsOne(C, F)` (operation)
▷ `AddIsOne(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsOne`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `IsOne`$(arg2)$.

7.6.120 AddIsProjective (for IsCapCategory, IsFunction)

▷ `AddIsProjective(C, F)` (operation)
▷ `AddIsProjective(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsProjective`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `IsProjective`$(arg2)$.

7.6.121 AddIsSplitEpimorphism (for IsCapCategory, IsFunction)

▷ `AddIsSplitEpimorphism(C, F)` (operation)
▷ `AddIsSplitEpimorphism(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsSplitEpimorphism`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `IsSplitEpimorphism`$(arg2)$.

7.6.122 AddIsSplitMonomorphism (for IsCapCategory, IsFunction)

▷ `AddIsSplitMonomorphism(C, F)` (operation)
▷ `AddIsSplitMonomorphism(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsSplitMonomorphism`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `IsSplitMonomorphism`$(arg2)$.

7.6.123 AddIsTerminal (for IsCapCategory, IsFunction)

▷ `AddIsTerminal(C, F)` (operation)
▷ `AddIsTerminal(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsTerminal`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `IsTerminal`$(arg2)$.

7.6.124 AddIsWellDefinedForMorphisms (for IsCapCategory, IsFunction)

▷ `AddIsWellDefinedForMorphisms(C, F)` (operation)
▷ `AddIsWellDefinedForMorphisms(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsWellDefinedForMorphisms`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ `IsWellDefinedForMorphisms`$(alpha)$.

7.6.125 AddIsWellDefinedForMorphismsWithGivenSourceAndRange (for IsCapCategory, IsFunction)

▷ `AddIsWellDefinedForMorphismsWithGivenSourceAndRange(C, F)` (operation)
▷ `AddIsWellDefinedForMorphismsWithGivenSourceAndRange(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsWellDefinedForMorphismsWithGivenSourceAndRange`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (source, alpha, range) \mapsto$ `IsWellDefinedForMorphismsWithGivenSourceAndRange`$(source, alpha, range)$.

7.6.126 AddIsWellDefinedForObjects (for IsCapCategory, IsFunction)

▷ `AddIsWellDefinedForObjects(C, F)` (operation)
▷ `AddIsWellDefinedForObjects(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsWellDefinedForObjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `IsWellDefinedForObjects`$(arg2)$.

7.6.127 AddIsWellDefinedForTwoCells (for IsCapCategory, IsFunction)

▷ `AddIsWellDefinedForTwoCells(C, F)` (operation)
▷ `AddIsWellDefinedForTwoCells(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsWellDefinedForTwoCells`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `IsWellDefinedForTwoCells`$(arg2)$.

7.6.128 AddIsZeroForMorphisms (for IsCapCategory, IsFunction)

▷ `AddIsZeroForMorphisms(C, F)` (operation)
▷ `AddIsZeroForMorphisms(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsZeroForMorphisms`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `IsZeroForMorphisms`$(arg2)$.

7.6.129 AddIsZeroForObjects (for IsCapCategory, IsFunction)

▷ `AddIsZeroForObjects(C, F)` (operation)
▷ `AddIsZeroForObjects(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsZeroForObjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `IsZeroForObjects`$(arg2)$.

7.6.130 AddIsomorphismFromCoequalizerOfCoproductDiagramToPushout (for IsCapCategory, IsFunction)

▷ `AddIsomorphismFromCoequalizerOfCoproductDiagramToPushout(C, F)` (operation)
▷ `AddIsomorphismFromCoequalizerOfCoproductDiagramToPushout(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsomorphismFromCoequalizerOfCoproductDiagramToPushout`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (D) \mapsto$ `IsomorphismFromCoequalizerOfCoproductDiagramToPushout`(D).

7.6.131 AddIsomorphismFromCoequalizerOfIdentityAndAutomorphismsToCoequalizer (for IsCapCategory, IsFunction)

▷ `AddIsomorphismFromCoequalizerOfIdentityAndAutomorphismsToCoequalizer(C, F)` (operation)
▷ `AddIsomorphismFromCoequalizerOfIdentityAndAutomorphismsToCoequalizer(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsomorphismFromCoequalizerOfIdentityAndAutomorphismsToCoequalizer`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (Y, automorphisms) \mapsto$ `IsomorphismFromCoequalizerOfIdentityAndAutomorphismsToCoequalizer`$(Y, automorphisms)$.

7.6.132 AddIsomorphismFromCoequalizerToCoequalizerOfIdentityAndAutomorphisms (for IsCapCategory, IsFunction)

▷ `AddIsomorphismFromCoequalizerToCoequalizerOfIdentityAndAutomorphisms(C, F)` (operation)
▷ `AddIsomorphismFromCoequalizerToCoequalizerOfIdentityAndAutomorphisms(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation

IsomorphismFromCoequalizerToCoequalizerOfIdentityAndAutomorphisms. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (Y, automorphisms) \mapsto$ IsomorphismFromCoequalizerToCoequalizerOfIdentityAndAutomorphisms$(Y, automorphisms)$.

7.6.133 AddIsomorphismFromCoequalizerToCokernelOfJointPairwiseDifferencesOfMorphismsFro (for IsCapCategory, IsFunction)

▷ AddIsomorphismFromCoequalizerToCokernelOfJointPairwiseDifferencesOfMorphismsFromCoproduct(*F*) (operation)

▷ AddIsomorphismFromCoequalizerToCokernelOfJointPairwiseDifferencesOfMorphismsFromCoproduct(*F*, *weight*) (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation IsomorphismFromCoequalizerToCokernelOfJointPairwiseDifferencesOfMorphismsFromCoproduct. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (A, D) \mapsto$ IsomorphismFromCoequalizerToCokernelOfJointPairwiseDifferencesOfMorphismsFromCoproduct(A, D)

7.6.134 AddIsomorphismFromCoimageToCokernelOfKernel (for IsCapCategory, IsFunction)

▷ AddIsomorphismFromCoimageToCokernelOfKernel(*C*, *F*) (operation)

▷ AddIsomorphismFromCoimageToCokernelOfKernel(*C*, *F*, *weight*) (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation IsomorphismFromCoimageToCokernelOfKernel. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ IsomorphismFromCoimageToCokernelOfKernel$(alpha)$.

7.6.135 AddIsomorphismFromCokernelOfJointPairwiseDifferencesOfMorphismsFromCoproductTo (for IsCapCategory, IsFunction)

▷ AddIsomorphismFromCokernelOfJointPairwiseDifferencesOfMorphismsFromCoproductToCoequalizer(*F*) (operation)

▷ AddIsomorphismFromCokernelOfJointPairwiseDifferencesOfMorphismsFromCoproductToCoequalizer(*F*, *weight*) (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation IsomorphismFromCokernelOfJointPairwiseDifferencesOfMorphismsFromCoproductToCoequalizer. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (A, D) \mapsto$ IsomorphismFromCokernelOfJointPairwiseDifferencesOfMorphismsFromCoproductToCoequalizer(A, D)

7.6.136 AddIsomorphismFromCokernelOfKernelToCoimage (for IsCapCategory, IsFunction)

▷ `AddIsomorphismFromCokernelOfKernelToCoimage(C, F)` (operation)
▷ `AddIsomorphismFromCokernelOfKernelToCoimage(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsomorphismFromCokernelOfKernelToCoimage`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ `IsomorphismFromCokernelOfKernelToCoimage`$(alpha)$.

7.6.137 AddIsomorphismFromCoproductToDirectSum (for IsCapCategory, IsFunction)

▷ `AddIsomorphismFromCoproductToDirectSum(C, F)` (operation)
▷ `AddIsomorphismFromCoproductToDirectSum(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsomorphismFromCoproductToDirectSum`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (D) \mapsto$ `IsomorphismFromCoproductToDirectSum`(D).

7.6.138 AddIsomorphismFromDirectProductToDirectSum (for IsCapCategory, IsFunction)

▷ `AddIsomorphismFromDirectProductToDirectSum(C, F)` (operation)
▷ `AddIsomorphismFromDirectProductToDirectSum(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsomorphismFromDirectProductToDirectSum`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (D) \mapsto$ `IsomorphismFromDirectProductToDirectSum`(D).

7.6.139 AddIsomorphismFromDirectSumToCoproduct (for IsCapCategory, IsFunction)

▷ `AddIsomorphismFromDirectSumToCoproduct(C, F)` (operation)
▷ `AddIsomorphismFromDirectSumToCoproduct(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsomorphismFromDirectSumToCoproduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (D) \mapsto$ `IsomorphismFromDirectSumToCoproduct`(D).

7.6.140 AddIsomorphismFromDirectSumToDirectProduct (for IsCapCategory, IsFunction)

▷ `AddIsomorphismFromDirectSumToDirectProduct(C, F)` (operation)
▷ `AddIsomorphismFromDirectSumToDirectProduct(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsomorphismFromDirectSumToDirectProduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (D) \mapsto$ `IsomorphismFromDirectSumToDirectProduct`(D).

7.6.141 AddIsomorphismFromEqualizerOfDirectProductDiagramToFiberProduct (for IsCapCategory, IsFunction)

▷ `AddIsomorphismFromEqualizerOfDirectProductDiagramToFiberProduct(C, F)` (operation)
▷ `AddIsomorphismFromEqualizerOfDirectProductDiagramToFiberProduct(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsomorphismFromEqualizerOfDirectProductDiagramToFiberProduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (D) \mapsto$ `IsomorphismFromEqualizerOfDirectProductDiagramToFiberProduct`(D).

7.6.142 AddIsomorphismFromEqualizerToKernelOfJointPairwiseDifferencesOfMorphismsIntoDire (for IsCapCategory, IsFunction)

▷ `AddIsomorphismFromEqualizerToKernelOfJointPairwiseDifferencesOfMorphismsIntoDirectProduct( F)` (operation)
▷ `AddIsomorphismFromEqualizerToKernelOfJointPairwiseDifferencesOfMorphismsIntoDirectProduct( F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsomorphismFromEqualizerToKernelOfJointPairwiseDifferencesOfMorphismsIntoDirectProduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (A, D) \mapsto$ `IsomorphismFromEqualizerToKernelOfJointPairwiseDifferencesOfMorphismsIntoDirectProduct`(A, D)

7.6.143 AddIsomorphismFromFiberProductToEqualizerOfDirectProductDiagram (for IsCapCategory, IsFunction)

▷ `AddIsomorphismFromFiberProductToEqualizerOfDirectProductDiagram(C, F)` (operation)
▷ `AddIsomorphismFromFiberProductToEqualizerOfDirectProductDiagram(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation IsomorphismFromFiberProductToEqualizerOfDirectProductDiagram. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (D) \mapsto$ IsomorphismFromFiberProductToEqualizerOfDirectProductDiagram(D).

7.6.144 AddIsomorphismFromHomologyObjectToItsConstructionAsAnImageObject (for IsCapCategory, IsFunction)

▷ AddIsomorphismFromHomologyObjectToItsConstructionAsAnImageObject(*C*, *F*) (operation)

▷ AddIsomorphismFromHomologyObjectToItsConstructionAsAnImageObject(*C*, *F*, *weight*) (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation IsomorphismFromHomologyObjectToItsConstructionAsAnImageObject. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, beta) \mapsto$ IsomorphismFromHomologyObjectToItsConstructionAsAnImageObject$(alpha, beta)$.

7.6.145 AddIsomorphismFromImageObjectToKernelOfCokernel (for IsCapCategory, IsFunction)

▷ AddIsomorphismFromImageObjectToKernelOfCokernel(*C*, *F*) (operation)

▷ AddIsomorphismFromImageObjectToKernelOfCokernel(*C*, *F*, *weight*) (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation IsomorphismFromImageObjectToKernelOfCokernel. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ IsomorphismFromImageObjectToKernelOfCokernel$(alpha)$.

7.6.146 AddIsomorphismFromInitialObjectToZeroObject (for IsCapCategory, IsFunction)

▷ AddIsomorphismFromInitialObjectToZeroObject(*C*, *F*) (operation)

▷ AddIsomorphismFromInitialObjectToZeroObject(*C*, *F*, *weight*) (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation IsomorphismFromInitialObjectToZeroObject. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : () \mapsto$ IsomorphismFromInitialObjectToZeroObject$()$.

7.6.147 AddIsomorphismFromItsConstructionAsAnImageObjectToHomologyObject (for IsCapCategory, IsFunction)

▷ `AddIsomorphismFromItsConstructionAsAnImageObjectToHomologyObject(C, F)` (operation)
▷ `AddIsomorphismFromItsConstructionAsAnImageObjectToHomologyObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsomorphismFromItsConstructionAsAnImageObjectToHomologyObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, beta) \mapsto$ `IsomorphismFromItsConstructionAsAnImageObjectToHomologyObject`$(alpha, beta)$.

7.6.148 AddIsomorphismFromKernelOfCokernelToImageObject (for IsCapCategory, IsFunction)

▷ `AddIsomorphismFromKernelOfCokernelToImageObject(C, F)` (operation)
▷ `AddIsomorphismFromKernelOfCokernelToImageObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsomorphismFromKernelOfCokernelToImageObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ `IsomorphismFromKernelOfCokernelToImageObject`$(alpha)$.

7.6.149 AddIsomorphismFromKernelOfJointPairwiseDifferencesOfMorphismsIntoDirectProductTo (for IsCapCategory, IsFunction)

▷ `AddIsomorphismFromKernelOfJointPairwiseDifferencesOfMorphismsIntoDirectProductToEqualizer( F)` (operation)
▷ `AddIsomorphismFromKernelOfJointPairwiseDifferencesOfMorphismsIntoDirectProductToEqualizer( F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `IsomorphismFromKernelOfJointPairwiseDifferencesOfMorphismsIntoDirectProductToEqualizer`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (A, D) \mapsto$ `IsomorphismFromKernelOfJointPairwiseDifferencesOfMorphismsIntoDirectProductToEqualizer`(A, D)

7.6.150 AddIsomorphismFromPushoutToCoequalizerOfCoproductDiagram (for IsCapCategory, IsFunction)

▷ `AddIsomorphismFromPushoutToCoequalizerOfCoproductDiagram(C, F)` (operation)
▷ `AddIsomorphismFromPushoutToCoequalizerOfCoproductDiagram(C, F, weight)` (opera-

tion)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation IsomorphismFromPushoutToCoequalizerOfCoproductDiagram. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (D) \mapsto$ IsomorphismFromPushoutToCoequalizerOfCoproductDiagram(D).

7.6.151 AddIsomorphismFromTerminalObjectToZeroObject (for IsCapCategory, IsFunction)

▷ AddIsomorphismFromTerminalObjectToZeroObject(*C*, *F*) (operation)

▷ AddIsomorphismFromTerminalObjectToZeroObject(*C*, *F*, *weight*) (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation IsomorphismFromTerminalObjectToZeroObject. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : () \mapsto$ IsomorphismFromTerminalObjectToZeroObject().

7.6.152 AddIsomorphismFromZeroObjectToInitialObject (for IsCapCategory, IsFunction)

▷ AddIsomorphismFromZeroObjectToInitialObject(*C*, *F*) (operation)

▷ AddIsomorphismFromZeroObjectToInitialObject(*C*, *F*, *weight*) (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation IsomorphismFromZeroObjectToInitialObject. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : () \mapsto$ IsomorphismFromZeroObjectToInitialObject().

7.6.153 AddIsomorphismFromZeroObjectToTerminalObject (for IsCapCategory, IsFunction)

▷ AddIsomorphismFromZeroObjectToTerminalObject(*C*, *F*) (operation)

▷ AddIsomorphismFromZeroObjectToTerminalObject(*C*, *F*, *weight*) (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation IsomorphismFromZeroObjectToTerminalObject. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : () \mapsto$ IsomorphismFromZeroObjectToTerminalObject().

7.6.154 AddJointPairwiseDifferencesOfMorphismsFromCoproduct (for IsCapCategory, IsFunction)

▷ `AddJointPairwiseDifferencesOfMorphismsFromCoproduct(C, F)` (operation)
▷ `AddJointPairwiseDifferencesOfMorphismsFromCoproduct(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `JointPairwiseDifferencesOfMorphismsFromCoproduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (A,D) \mapsto$ `JointPairwiseDifferencesOfMorphismsFromCoproduct`(A,D).

7.6.155 AddJointPairwiseDifferencesOfMorphismsIntoDirectProduct (for IsCapCategory, IsFunction)

▷ `AddJointPairwiseDifferencesOfMorphismsIntoDirectProduct(C, F)` (operation)
▷ `AddJointPairwiseDifferencesOfMorphismsIntoDirectProduct(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `JointPairwiseDifferencesOfMorphismsIntoDirectProduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (A,D) \mapsto$ `JointPairwiseDifferencesOfMorphismsIntoDirectProduct`(A,D).

7.6.156 AddKernelEmbedding (for IsCapCategory, IsFunction)

▷ `AddKernelEmbedding(C, F)` (operation)
▷ `AddKernelEmbedding(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `KernelEmbedding`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ `KernelEmbedding`$(alpha)$.

7.6.157 AddKernelEmbeddingWithGivenKernelObject (for IsCapCategory, IsFunction)

▷ `AddKernelEmbeddingWithGivenKernelObject(C, F)` (operation)
▷ `AddKernelEmbeddingWithGivenKernelObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `KernelEmbeddingWithGivenKernelObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha,P) \mapsto$ `KernelEmbeddingWithGivenKernelObject`$(alpha,P)$.

7.6.158 AddKernelLift (for IsCapCategory, IsFunction)

▷ `AddKernelLift(C, F)` (operation)
▷ `AddKernelLift(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `KernelLift`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, T, tau) \mapsto$ `KernelLift`$(alpha, T, tau)$.

7.6.159 AddKernelLiftWithGivenKernelObject (for IsCapCategory, IsFunction)

▷ `AddKernelLiftWithGivenKernelObject(C, F)` (operation)
▷ `AddKernelLiftWithGivenKernelObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `KernelLiftWithGivenKernelObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, T, tau, P) \mapsto$ `KernelLiftWithGivenKernelObject`$(alpha, T, tau, P)$.

7.6.160 AddKernelObject (for IsCapCategory, IsFunction)

▷ `AddKernelObject(C, F)` (operation)
▷ `AddKernelObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `KernelObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ `KernelObject`$(alpha)$.

7.6.161 AddKernelObjectFunctorial (for IsCapCategory, IsFunction)

▷ `AddKernelObjectFunctorial(C, F)` (operation)
▷ `AddKernelObjectFunctorial(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `KernelObjectFunctorial`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, mu, alphap) \mapsto$ `KernelObjectFunctorial`$(alpha, mu, alphap)$.

7.6.162 AddKernelObjectFunctorialWithGivenKernelObjects (for IsCapCategory, IsFunction)

▷ `AddKernelObjectFunctorialWithGivenKernelObjects(C, F)` (operation)
▷ `AddKernelObjectFunctorialWithGivenKernelObjects(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `KernelObjectFunctorialWithGivenKernelObjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (P, alpha, mu, alphap, Pp) \mapsto$ `KernelObjectFunctorialWithGivenKernelObjects`$(P, alpha, mu, alphap, Pp)$.

7.6.163 AddLift (for IsCapCategory, IsFunction)

▷ `AddLift(C, F)` (operation)
▷ `AddLift(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `Lift`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, beta) \mapsto$ `Lift`$(alpha, beta)$.

7.6.164 AddLiftAlongMonomorphism (for IsCapCategory, IsFunction)

▷ `AddLiftAlongMonomorphism(C, F)` (operation)
▷ `AddLiftAlongMonomorphism(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `LiftAlongMonomorphism`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (iota, tau) \mapsto$ `LiftAlongMonomorphism`$(iota, tau)$.

7.6.165 AddLinearCombinationOfMorphisms (for IsCapCategory, IsFunction)

▷ `AddLinearCombinationOfMorphisms(C, F)` (operation)
▷ `AddLinearCombinationOfMorphisms(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `LinearCombinationOfMorphisms`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (source, list_of_ring_elements, list_of_morphisms, range) \mapsto$ `LinearCombinationOfMorphisms`$(source, list_of_ring_elements, list_of_morphisms, range)$.

7.6.166 AddMereExistenceOfSolutionOfLinearSystemInAbCategory (for IsCapCategory, IsFunction)

▷ `AddMereExistenceOfSolutionOfLinearSystemInAbCategory(C, F)` (operation)
▷ `AddMereExistenceOfSolutionOfLinearSystemInAbCategory(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation

MereExistenceOfSolutionOfLinearSystemInAbCategory. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3, arg4) \mapsto$ MereExistenceOfSolutionOfLinearSystemInAbCategory$(arg2, arg3, arg4)$.

7.6.167 AddMereExistenceOfUniqueSolutionOfHomogeneousLinearSystemInAbCategory (for IsCapCategory, IsFunction)

▷ AddMereExistenceOfUniqueSolutionOfHomogeneousLinearSystemInAbCategory(*C*, *F*) (operation)

▷ AddMereExistenceOfUniqueSolutionOfHomogeneousLinearSystemInAbCategory(*C*, *F*, *weight*) (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation MereExistenceOfUniqueSolutionOfHomogeneousLinearSystemInAbCategory. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ MereExistenceOfUniqueSolutionOfHomogeneousLinearSystemInAbCategory$(arg2, arg3)$.

7.6.168 AddMereExistenceOfUniqueSolutionOfLinearSystemInAbCategory (for IsCapCategory, IsFunction)

▷ AddMereExistenceOfUniqueSolutionOfLinearSystemInAbCategory(*C*, *F*) (operation)

▷ AddMereExistenceOfUniqueSolutionOfLinearSystemInAbCategory(*C*, *F*, *weight*) (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation MereExistenceOfUniqueSolutionOfLinearSystemInAbCategory. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3, arg4) \mapsto$ MereExistenceOfUniqueSolutionOfLinearSystemInAbCategory$(arg2, arg3, arg4)$.

7.6.169 AddMonomorphismIntoInjectiveEnvelopeObject (for IsCapCategory, IsFunction)

▷ AddMonomorphismIntoInjectiveEnvelopeObject(*C*, *F*) (operation)

▷ AddMonomorphismIntoInjectiveEnvelopeObject(*C*, *F*, *weight*) (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation MonomorphismIntoInjectiveEnvelopeObject. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (A) \mapsto$ MonomorphismIntoInjectiveEnvelopeObject(A).

7.6.170 AddMonomorphismIntoInjectiveEnvelopeObjectWithGivenInjectiveEnvelopeObject (for IsCapCategory, IsFunction)

▷ `AddMonomorphismIntoInjectiveEnvelopeObjectWithGivenInjectiveEnvelopeObject(C, F)` (operation)
▷ `AddMonomorphismIntoInjectiveEnvelopeObjectWithGivenInjectiveEnvelopeObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MonomorphismIntoInjectiveEnvelopeObjectWithGivenInjectiveEnvelopeObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (A,I) \mapsto$ `MonomorphismIntoInjectiveEnvelopeObjectWithGivenInjectiveEnvelopeObject`(A,I).

7.6.171 AddMonomorphismIntoSomeInjectiveObject (for IsCapCategory, IsFunction)

▷ `AddMonomorphismIntoSomeInjectiveObject(C, F)` (operation)
▷ `AddMonomorphismIntoSomeInjectiveObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MonomorphismIntoSomeInjectiveObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (A) \mapsto$ `MonomorphismIntoSomeInjectiveObject`(A).

7.6.172 AddMonomorphismIntoSomeInjectiveObjectWithGivenSomeInjectiveObject (for IsCapCategory, IsFunction)

▷ `AddMonomorphismIntoSomeInjectiveObjectWithGivenSomeInjectiveObject(C, F)` (operation)
▷ `AddMonomorphismIntoSomeInjectiveObjectWithGivenSomeInjectiveObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MonomorphismIntoSomeInjectiveObjectWithGivenSomeInjectiveObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (A,I) \mapsto$ `MonomorphismIntoSomeInjectiveObjectWithGivenSomeInjectiveObject`(A,I).

7.6.173 AddMorphismBetweenDirectSums (for IsCapCategory, IsFunction)

▷ `AddMorphismBetweenDirectSums(C, F)` (operation)
▷ `AddMorphismBetweenDirectSums(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MorphismBetweenDirectSums`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (source_diagram, mat, range_diagram) \mapsto$ `MorphismBetweenDirectSums`$(source_diagram, mat, range_diagram)$.

7.6.174 AddMorphismBetweenDirectSumsWithGivenDirectSums (for IsCapCategory, IsFunction)

▷ `AddMorphismBetweenDirectSumsWithGivenDirectSums(C, F)` (operation)
▷ `AddMorphismBetweenDirectSumsWithGivenDirectSums(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MorphismBetweenDirectSumsWithGivenDirectSums`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (S, source_diagram, mat, range_diagram, T) \mapsto$ `MorphismBetweenDirectSumsWithGivenDirectSums`$(S, source_diagram, mat, range_diagram, T)$.

7.6.175 AddMorphismConstructor (for IsCapCategory, IsFunction)

▷ `AddMorphismConstructor(C, F)` (operation)
▷ `AddMorphismConstructor(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MorphismConstructor`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3, arg4) \mapsto$ `MorphismConstructor`$(arg2, arg3, arg4)$.

7.6.176 AddMorphismDatum (for IsCapCategory, IsFunction)

▷ `AddMorphismDatum(C, F)` (operation)
▷ `AddMorphismDatum(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MorphismDatum`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `MorphismDatum`$(arg2)$.

7.6.177 AddMorphismFromCoimageToImage (for IsCapCategory, IsFunction)

▷ `AddMorphismFromCoimageToImage(C, F)` (operation)
▷ `AddMorphismFromCoimageToImage(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MorphismFromCoimageToImage`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ `MorphismFromCoimageToImage`$(alpha)$.

7.6.178 AddMorphismFromCoimageToImageWithGivenObjects (for IsCapCategory, IsFunction)

▷ `AddMorphismFromCoimageToImageWithGivenObjects(C, F)` (operation)
▷ `AddMorphismFromCoimageToImageWithGivenObjects(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MorphismFromCoimageToImageWithGivenObjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (C, alpha, I) \mapsto$ `MorphismFromCoimageToImageWithGivenObjects`$(C, alpha, I)$.

7.6.179 AddMorphismFromEqualizerToSink (for IsCapCategory, IsFunction)

▷ `AddMorphismFromEqualizerToSink(C, F)` (operation)
▷ `AddMorphismFromEqualizerToSink(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MorphismFromEqualizerToSink`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (Y, morphisms) \mapsto$ `MorphismFromEqualizerToSink`$(Y, morphisms)$.

7.6.180 AddMorphismFromEqualizerToSinkWithGivenEqualizer (for IsCapCategory, IsFunction)

▷ `AddMorphismFromEqualizerToSinkWithGivenEqualizer(C, F)` (operation)
▷ `AddMorphismFromEqualizerToSinkWithGivenEqualizer(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MorphismFromEqualizerToSinkWithGivenEqualizer`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (Y, morphisms, P) \mapsto$ `MorphismFromEqualizerToSinkWithGivenEqualizer`$(Y, morphisms, P)$.

7.6.181 AddMorphismFromFiberProductToSink (for IsCapCategory, IsFunction)

▷ `AddMorphismFromFiberProductToSink(C, F)` (operation)
▷ `AddMorphismFromFiberProductToSink(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MorphismFromFiberProductToSink`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (morphisms) \mapsto$ `MorphismFromFiberProductToSink`$(morphisms)$.

7.6.182 AddMorphismFromFiberProductToSinkWithGivenFiberProduct (for IsCapCategory, IsFunction)

▷ `AddMorphismFromFiberProductToSinkWithGivenFiberProduct(C, F)` (operation)
▷ `AddMorphismFromFiberProductToSinkWithGivenFiberProduct(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MorphismFromFiberProductToSinkWithGivenFiberProduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (morphisms, P) \mapsto$ `MorphismFromFiberProductToSinkWithGivenFiberProduct`$(morphisms, P)$.

7.6.183 AddMorphismFromKernelObjectToSink (for IsCapCategory, IsFunction)

▷ `AddMorphismFromKernelObjectToSink(C, F)` (operation)
▷ `AddMorphismFromKernelObjectToSink(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MorphismFromKernelObjectToSink`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ `MorphismFromKernelObjectToSink`$(alpha)$.

7.6.184 AddMorphismFromKernelObjectToSinkWithGivenKernelObject (for IsCapCategory, IsFunction)

▷ `AddMorphismFromKernelObjectToSinkWithGivenKernelObject(C, F)` (operation)
▷ `AddMorphismFromKernelObjectToSinkWithGivenKernelObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MorphismFromKernelObjectToSinkWithGivenKernelObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, P) \mapsto$ `MorphismFromKernelObjectToSinkWithGivenKernelObject`$(alpha, P)$.

7.6.185 AddMorphismFromSourceToCoequalizer (for IsCapCategory, IsFunction)

▷ `AddMorphismFromSourceToCoequalizer(C, F)` (operation)
▷ `AddMorphismFromSourceToCoequalizer(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MorphismFromSourceToCoequalizer`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (Y, morphisms) \mapsto$ `MorphismFromSourceToCoequalizer`$(Y, morphisms)$.

7.6.186 AddMorphismFromSourceToCoequalizerWithGivenCoequalizer (for IsCapCategory, IsFunction)

▷ `AddMorphismFromSourceToCoequalizerWithGivenCoequalizer(C, F)` (operation)
▷ `AddMorphismFromSourceToCoequalizerWithGivenCoequalizer(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MorphismFromSourceToCoequalizerWithGivenCoequalizer`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (Y, morphisms, P) \mapsto$ `MorphismFromSourceToCoequalizerWithGivenCoequalizer`$(Y, morphisms, P)$.

7.6.187 AddMorphismFromSourceToCokernelObject (for IsCapCategory, IsFunction)

▷ `AddMorphismFromSourceToCokernelObject(C, F)` (operation)
▷ `AddMorphismFromSourceToCokernelObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MorphismFromSourceToCokernelObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ `MorphismFromSourceToCokernelObject`$(alpha)$.

7.6.188 AddMorphismFromSourceToCokernelObjectWithGivenCokernelObject (for IsCapCategory, IsFunction)

▷ `AddMorphismFromSourceToCokernelObjectWithGivenCokernelObject(C, F)` (operation)
▷ `AddMorphismFromSourceToCokernelObjectWithGivenCokernelObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MorphismFromSourceToCokernelObjectWithGivenCokernelObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, P) \mapsto$ `MorphismFromSourceToCokernelObjectWithGivenCokernelObject`$(alpha, P)$.

7.6.189 AddMorphismFromSourceToPushout (for IsCapCategory, IsFunction)

▷ `AddMorphismFromSourceToPushout(C, F)` (operation)
▷ `AddMorphismFromSourceToPushout(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MorphismFromSourceToPushout`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (morphisms) \mapsto$ `MorphismFromSourceToPushout`$(morphisms)$.

7.6.190 AddMorphismFromSourceToPushoutWithGivenPushout (for IsCapCategory, IsFunction)

▷ `AddMorphismFromSourceToPushoutWithGivenPushout(C, F)` (operation)
▷ `AddMorphismFromSourceToPushoutWithGivenPushout(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MorphismFromSourceToPushoutWithGivenPushout`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (morphisms, P) \mapsto$ `MorphismFromSourceToPushoutWithGivenPushout`$(morphisms, P)$.

7.6.191 AddMorphismsOfExternalHom (for IsCapCategory, IsFunction)

▷ `AddMorphismsOfExternalHom(C, F)` (operation)
▷ `AddMorphismsOfExternalHom(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MorphismsOfExternalHom`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `MorphismsOfExternalHom`$(arg2, arg3)$.

7.6.192 AddMultiplyWithElementOfCommutativeSemiringForMorphisms (for IsCapCategory, IsFunction)

▷ `AddMultiplyWithElementOfCommutativeSemiringForMorphisms(C, F)` (operation)
▷ `AddMultiplyWithElementOfCommutativeSemiringForMorphisms(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `MultiplyWithElementOfCommutativeSemiringForMorphisms`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (r, alpha) \mapsto$ `MultiplyWithElementOfCommutativeSemiringForMorphisms`$(r, alpha)$.

7.6.193 AddObjectConstructor (for IsCapCategory, IsFunction)

▷ `AddObjectConstructor(C, F)` (operation)
▷ `AddObjectConstructor(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ObjectConstructor`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto \texttt{ObjectConstructor}(arg2)$.

7.6.194 AddObjectDatum (for IsCapCategory, IsFunction)

▷ `AddObjectDatum(C, F)` (operation)
▷ `AddObjectDatum(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ObjectDatum`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto \texttt{ObjectDatum}(arg2)$.

7.6.195 AddPostCompose (for IsCapCategory, IsFunction)

▷ `AddPostCompose(C, F)` (operation)
▷ `AddPostCompose(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `PostCompose`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (beta, alpha) \mapsto \texttt{PostCompose}(beta, alpha)$.

7.6.196 AddPostComposeList (for IsCapCategory, IsFunction)

▷ `AddPostComposeList(C, F)` (operation)
▷ `AddPostComposeList(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `PostComposeList`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (source, list_o f_m orphisms, range) \mapsto \texttt{PostComposeList}(source, list_o f_m orphisms, range)$.

7.6.197 AddPostInverseForMorphisms (for IsCapCategory, IsFunction)

▷ `AddPostInverseForMorphisms(C, F)` (operation)
▷ `AddPostInverseForMorphisms(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `PostInverseForMorphisms`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto \texttt{PostInverseForMorphisms}(alpha)$.

7.6.198 AddPreCompose (for IsCapCategory, IsFunction)

▷ `AddPreCompose(C, F)` (operation)
▷ `AddPreCompose(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `PreCompose`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, beta) \mapsto$ `PreCompose`$(alpha, beta)$.

7.6.199 AddPreComposeList (for IsCapCategory, IsFunction)

▷ `AddPreComposeList(C, F)` (operation)
▷ `AddPreComposeList(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `PreComposeList`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (source, list_of_morphisms, range) \mapsto$ `PreComposeList`$(source, list_of_morphisms, range)$.

7.6.200 AddPreInverseForMorphisms (for IsCapCategory, IsFunction)

▷ `AddPreInverseForMorphisms(C, F)` (operation)
▷ `AddPreInverseForMorphisms(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `PreInverseForMorphisms`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ `PreInverseForMorphisms`$(alpha)$.

7.6.201 AddProjectionInFactorOfDirectProduct (for IsCapCategory, IsFunction)

▷ `AddProjectionInFactorOfDirectProduct(C, F)` (operation)
▷ `AddProjectionInFactorOfDirectProduct(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ProjectionInFactorOfDirectProduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (objects, k) \mapsto$ `ProjectionInFactorOfDirectProduct`$(objects, k)$.

7.6.202 AddProjectionInFactorOfDirectProductWithGivenDirectProduct (for IsCapCategory, IsFunction)

▷ `AddProjectionInFactorOfDirectProductWithGivenDirectProduct(C, F)` (operation)
▷ `AddProjectionInFactorOfDirectProductWithGivenDirectProduct(C, F, weight)` (oper-

ation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ProjectionInFactorOfDirectProductWithGivenDirectProduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (objects, k, P) \mapsto$ `ProjectionInFactorOfDirectProductWithGivenDirectProduct`$(objects, k, P)$.

7.6.203 AddProjectionInFactorOfDirectSum (for IsCapCategory, IsFunction)

▷ `AddProjectionInFactorOfDirectSum(C, F)` (operation)
▷ `AddProjectionInFactorOfDirectSum(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ProjectionInFactorOfDirectSum`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (objects, k) \mapsto$ `ProjectionInFactorOfDirectSum`$(objects, k)$.

7.6.204 AddProjectionInFactorOfDirectSumWithGivenDirectSum (for IsCapCategory, IsFunction)

▷ `AddProjectionInFactorOfDirectSumWithGivenDirectSum(C, F)` (operation)
▷ `AddProjectionInFactorOfDirectSumWithGivenDirectSum(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ProjectionInFactorOfDirectSumWithGivenDirectSum`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (objects, k, P) \mapsto$ `ProjectionInFactorOfDirectSumWithGivenDirectSum`$(objects, k, P)$.

7.6.205 AddProjectionInFactorOfFiberProduct (for IsCapCategory, IsFunction)

▷ `AddProjectionInFactorOfFiberProduct(C, F)` (operation)
▷ `AddProjectionInFactorOfFiberProduct(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ProjectionInFactorOfFiberProduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (morphisms, k) \mapsto$ `ProjectionInFactorOfFiberProduct`$(morphisms, k)$.

7.6.206 AddProjectionInFactorOfFiberProductWithGivenFiberProduct (for IsCapCategory, IsFunction)

▷ `AddProjectionInFactorOfFiberProductWithGivenFiberProduct(C, F)` (operation)
▷ `AddProjectionInFactorOfFiberProductWithGivenFiberProduct(C, F, weight)` (opera-

tion)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ProjectionInFactorOfFiberProductWithGivenFiberProduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (morphisms, k, P) \mapsto$ `ProjectionInFactorOfFiberProductWithGivenFiberProduct`$(morphisms, k, P)$.

7.6.207 AddProjectionOntoCoequalizer (for IsCapCategory, IsFunction)

▷ `AddProjectionOntoCoequalizer(C, F)` (operation)

▷ `AddProjectionOntoCoequalizer(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ProjectionOntoCoequalizer`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (Y, morphisms) \mapsto$ `ProjectionOntoCoequalizer`$(Y, morphisms)$.

7.6.208 AddProjectionOntoCoequalizerOfIdentityAndAutomorphisms (for IsCapCategory, IsFunction)

▷ `AddProjectionOntoCoequalizerOfIdentityAndAutomorphisms(C, F)` (operation)

▷ `AddProjectionOntoCoequalizerOfIdentityAndAutomorphisms(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ProjectionOntoCoequalizerOfIdentityAndAutomorphisms`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (Y, automorphisms) \mapsto$ `ProjectionOntoCoequalizerOfIdentityAndAutomorphisms`$(Y, automorphisms)$.

7.6.209 AddProjectionOntoCoequalizerOfIdentityAndAutomorphismsWithGivenCoequalizer (for IsCapCategory, IsFunction)

▷ `AddProjectionOntoCoequalizerOfIdentityAndAutomorphismsWithGivenCoequalizer(C, F)` (operation)

▷ `AddProjectionOntoCoequalizerOfIdentityAndAutomorphismsWithGivenCoequalizer(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ProjectionOntoCoequalizerOfIdentityAndAutomorphismsWithGivenCoequalizer`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (Y, automorphisms, P) \mapsto$ `ProjectionOntoCoequalizerOfIdentityAndAutomorphismsWithGivenCoequalizer`$(Y, automorphisms, P)$.

7.6.210 AddProjectionOntoCoequalizerWithGivenCoequalizer (for IsCapCategory, IsFunction)

▷ `AddProjectionOntoCoequalizerWithGivenCoequalizer(C, F)` (operation)
▷ `AddProjectionOntoCoequalizerWithGivenCoequalizer(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ProjectionOntoCoequalizerWithGivenCoequalizer`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (Y, morphisms, P) \mapsto$ `ProjectionOntoCoequalizerWithGivenCoequalizer`$(Y, morphisms, P)$.

7.6.211 AddProjectiveCoverObject (for IsCapCategory, IsFunction)

▷ `AddProjectiveCoverObject(C, F)` (operation)
▷ `AddProjectiveCoverObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ProjectiveCoverObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `ProjectiveCoverObject`$(arg2)$.

7.6.212 AddProjectiveDimension (for IsCapCategory, IsFunction)

▷ `AddProjectiveDimension(C, F)` (operation)
▷ `AddProjectiveDimension(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ProjectiveDimension`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `ProjectiveDimension`$(arg2)$.

7.6.213 AddProjectiveLift (for IsCapCategory, IsFunction)

▷ `AddProjectiveLift(C, F)` (operation)
▷ `AddProjectiveLift(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ProjectiveLift`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, beta) \mapsto$ `ProjectiveLift`$(alpha, beta)$.

7.6.214 AddPushout (for IsCapCategory, IsFunction)

▷ `AddPushout(C, F)` (operation)
▷ `AddPushout(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `Pushout`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (morphisms) \mapsto \texttt{Pushout}(morphisms)$.

7.6.215 AddPushoutFunctorial (for IsCapCategory, IsFunction)

▷ `AddPushoutFunctorial(C, F)` (operation)
▷ `AddPushoutFunctorial(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `PushoutFunctorial`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (morphisms, L, morphismsp) \mapsto \texttt{PushoutFunctorial}(morphisms, L, morphismsp)$.

7.6.216 AddPushoutFunctorialWithGivenPushouts (for IsCapCategory, IsFunction)

▷ `AddPushoutFunctorialWithGivenPushouts(C, F)` (operation)
▷ `AddPushoutFunctorialWithGivenPushouts(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `PushoutFunctorialWithGivenPushouts`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (P, morphisms, L, morphismsp, Pp) \mapsto \texttt{PushoutFunctorialWithGivenPushouts}(P, morphisms, L, morphismsp, Pp)$.

7.6.217 AddRandomMorphismByInteger (for IsCapCategory, IsFunction)

▷ `AddRandomMorphismByInteger(C, F)` (operation)
▷ `AddRandomMorphismByInteger(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `RandomMorphismByInteger`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (n) \mapsto \texttt{RandomMorphismByInteger}(n)$.

7.6.218 AddRandomMorphismByList (for IsCapCategory, IsFunction)

▷ `AddRandomMorphismByList(C, F)` (operation)
▷ `AddRandomMorphismByList(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `RandomMorphismByList`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (L) \mapsto$ `RandomMorphismByList`(L).

7.6.219 AddRandomMorphismWithFixedRangeByInteger (for IsCapCategory, IsFunction)

▷ `AddRandomMorphismWithFixedRangeByInteger(C, F)` (operation)
▷ `AddRandomMorphismWithFixedRangeByInteger(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `RandomMorphismWithFixedRangeByInteger`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (B,n) \mapsto$ `RandomMorphismWithFixedRangeByInteger`(B,n).

7.6.220 AddRandomMorphismWithFixedRangeByList (for IsCapCategory, IsFunction)

▷ `AddRandomMorphismWithFixedRangeByList(C, F)` (operation)
▷ `AddRandomMorphismWithFixedRangeByList(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `RandomMorphismWithFixedRangeByList`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (B,L) \mapsto$ `RandomMorphismWithFixedRangeByList`(B,L).

7.6.221 AddRandomMorphismWithFixedSourceAndRangeByInteger (for IsCapCategory, IsFunction)

▷ `AddRandomMorphismWithFixedSourceAndRangeByInteger(C, F)` (operation)
▷ `AddRandomMorphismWithFixedSourceAndRangeByInteger(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `RandomMorphismWithFixedSourceAndRangeByInteger`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (A,B,n) \mapsto$ `RandomMorphismWithFixedSourceAndRangeByInteger`(A,B,n).

7.6.222 AddRandomMorphismWithFixedSourceAndRangeByList (for IsCapCategory, IsFunction)

▷ `AddRandomMorphismWithFixedSourceAndRangeByList(C, F)` (operation)
▷ `AddRandomMorphismWithFixedSourceAndRangeByList(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `RandomMorphismWithFixedSourceAndRangeByList`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (A,B,L) \mapsto$ `RandomMorphismWithFixedSourceAndRangeByList`(A,B,L).

7.6.223 AddRandomMorphismWithFixedSourceByInteger (for IsCapCategory, IsFunction)

▷ `AddRandomMorphismWithFixedSourceByInteger(C, F)` (operation)
▷ `AddRandomMorphismWithFixedSourceByInteger(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `RandomMorphismWithFixedSourceByInteger`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (A,n) \mapsto$ `RandomMorphismWithFixedSourceByInteger`(A,n).

7.6.224 AddRandomMorphismWithFixedSourceByList (for IsCapCategory, IsFunction)

▷ `AddRandomMorphismWithFixedSourceByList(C, F)` (operation)
▷ `AddRandomMorphismWithFixedSourceByList(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `RandomMorphismWithFixedSourceByList`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (A,L) \mapsto$ `RandomMorphismWithFixedSourceByList`(A,L).

7.6.225 AddRandomObjectByInteger (for IsCapCategory, IsFunction)

▷ `AddRandomObjectByInteger(C, F)` (operation)
▷ `AddRandomObjectByInteger(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `RandomObjectByInteger`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (n) \mapsto$ `RandomObjectByInteger`(n).

7.6.226 AddRandomObjectByList (for IsCapCategory, IsFunction)

▷ `AddRandomObjectByList(C, F)` (operation)
▷ `AddRandomObjectByList(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `RandomObjectByList`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (L) \mapsto$ `RandomObjectByList`(L).

be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (L) \mapsto$ `RandomObjectByList`(L).

7.6.227 AddSetOfGeneratingMorphismsOfCategory (for IsCapCategory, IsFunction)

▷ `AddSetOfGeneratingMorphismsOfCategory(C, F)` (operation)
▷ `AddSetOfGeneratingMorphismsOfCategory(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SetOfGeneratingMorphismsOfCategory`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : () \mapsto$ `SetOfGeneratingMorphismsOfCategory`$()$.

7.6.228 AddSetOfMorphismsOfFiniteCategory (for IsCapCategory, IsFunction)

▷ `AddSetOfMorphismsOfFiniteCategory(C, F)` (operation)
▷ `AddSetOfMorphismsOfFiniteCategory(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SetOfMorphismsOfFiniteCategory`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : () \mapsto$ `SetOfMorphismsOfFiniteCategory`$()$.

7.6.229 AddSetOfObjectsOfCategory (for IsCapCategory, IsFunction)

▷ `AddSetOfObjectsOfCategory(C, F)` (operation)
▷ `AddSetOfObjectsOfCategory(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SetOfObjectsOfCategory`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : () \mapsto$ `SetOfObjectsOfCategory`$()$.

7.6.230 AddSimplifyEndo (for IsCapCategory, IsFunction)

▷ `AddSimplifyEndo(C, F)` (operation)
▷ `AddSimplifyEndo(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SimplifyEndo`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (mor, n) \mapsto$ `SimplifyEndo`(mor, n).

7.6.231 AddSimplifyEndo_IsoFromInputObject (for IsCapCategory, IsFunction)

▷ `AddSimplifyEndo_IsoFromInputObject(C, F)` (operation)
▷ `AddSimplifyEndo_IsoFromInputObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SimplifyEndo_IsoFromInputObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (mor, n) \mapsto$ $\mathtt{SimplifyEndo_IsoFromInputObject}(mor, n)$.

7.6.232 AddSimplifyEndo_IsoToInputObject (for IsCapCategory, IsFunction)

▷ `AddSimplifyEndo_IsoToInputObject(C, F)` (operation)
▷ `AddSimplifyEndo_IsoToInputObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SimplifyEndo_IsoToInputObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (mor, n) \mapsto$ $\mathtt{SimplifyEndo_IsoToInputObject}(mor, n)$.

7.6.233 AddSimplifyMorphism (for IsCapCategory, IsFunction)

▷ `AddSimplifyMorphism(C, F)` (operation)
▷ `AddSimplifyMorphism(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SimplifyMorphism`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (mor, n) \mapsto \mathtt{SimplifyMorphism}(mor, n)$.

7.6.234 AddSimplifyObject (for IsCapCategory, IsFunction)

▷ `AddSimplifyObject(C, F)` (operation)
▷ `AddSimplifyObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SimplifyObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (A, n) \mapsto \mathtt{SimplifyObject}(A, n)$.

7.6.235 AddSimplifyObject_IsoFromInputObject (for IsCapCategory, IsFunction)

▷ `AddSimplifyObject_IsoFromInputObject(C, F)` (operation)
▷ `AddSimplifyObject_IsoFromInputObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SimplifyObject_IsoFromInputObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (A,n) \mapsto$ `SimplifyObject`$_{\mathtt{I}}$`soFromInputObject`(A,n).

7.6.236 AddSimplifyObject_IsoToInputObject (for IsCapCategory, IsFunction)

▷ `AddSimplifyObject_IsoToInputObject(C, F)` (operation)
▷ `AddSimplifyObject_IsoToInputObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SimplifyObject_IsoToInputObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (A,n) \mapsto$ `SimplifyObject`$_{\mathtt{I}}$`soToInputObject`(A,n).

7.6.237 AddSimplifyRange (for IsCapCategory, IsFunction)

▷ `AddSimplifyRange(C, F)` (operation)
▷ `AddSimplifyRange(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SimplifyRange`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (mor,n) \mapsto$ `SimplifyRange`(mor,n).

7.6.238 AddSimplifyRange_IsoFromInputObject (for IsCapCategory, IsFunction)

▷ `AddSimplifyRange_IsoFromInputObject(C, F)` (operation)
▷ `AddSimplifyRange_IsoFromInputObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SimplifyRange_IsoFromInputObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (mor,n) \mapsto$ `SimplifyRange`$_{\mathtt{I}}$`soFromInputObject`(mor,n).

7.6.239 AddSimplifyRange_IsoToInputObject (for IsCapCategory, IsFunction)

▷ `AddSimplifyRange_IsoToInputObject(C, F)` (operation)
▷ `AddSimplifyRange_IsoToInputObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SimplifyRange_IsoToInputObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (mor,n) \mapsto$ `SimplifyRange`$_{\mathtt{I}}$`soToInputObject`(mor,n).

7.6.240 AddSimplifySource (for IsCapCategory, IsFunction)

▷ `AddSimplifySource(C, F)` (operation)
▷ `AddSimplifySource(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SimplifySource`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (mor, n) \mapsto$ `SimplifySource`(mor, n).

7.6.241 AddSimplifySourceAndRange (for IsCapCategory, IsFunction)

▷ `AddSimplifySourceAndRange(C, F)` (operation)
▷ `AddSimplifySourceAndRange(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SimplifySourceAndRange`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (mor, n) \mapsto$ `SimplifySourceAndRange`(mor, n).

7.6.242 AddSimplifySourceAndRange_IsoFromInputRange (for IsCapCategory, IsFunction)

▷ `AddSimplifySourceAndRange_IsoFromInputRange(C, F)` (operation)
▷ `AddSimplifySourceAndRange_IsoFromInputRange(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SimplifySourceAndRange_IsoFromInputRange`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (mor, n) \mapsto$ `SimplifySourceAndRange`$_{\mathtt{I}}$`soFromInputRange`(mor, n).

7.6.243 AddSimplifySourceAndRange_IsoFromInputSource (for IsCapCategory, IsFunction)

▷ `AddSimplifySourceAndRange_IsoFromInputSource(C, F)` (operation)
▷ `AddSimplifySourceAndRange_IsoFromInputSource(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SimplifySourceAndRange_IsoFromInputSource`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (mor, n) \mapsto$ `SimplifySourceAndRange`$_{\mathtt{I}}$`soFromInputSource`(mor, n).

7.6.244 AddSimplifySourceAndRange_IsoToInputRange (for IsCapCategory, IsFunction)

▷ `AddSimplifySourceAndRange_IsoToInputRange(C, F)` (operation)

▷ `AddSimplifySourceAndRange_IsoToInputRange(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SimplifySourceAndRange_IsoToInputRange`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (mor,n) \mapsto$ `SimplifySourceAndRange`$_\mathtt{I}$`soToInputRange`(mor,n).

7.6.245 AddSimplifySourceAndRange_IsoToInputSource (for IsCapCategory, IsFunction)

▷ `AddSimplifySourceAndRange_IsoToInputSource(C, F)` (operation)

▷ `AddSimplifySourceAndRange_IsoToInputSource(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SimplifySourceAndRange_IsoToInputSource`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (mor,n) \mapsto$ `SimplifySourceAndRange`$_\mathtt{I}$`soToInputSource`(mor,n).

7.6.246 AddSimplifySource_IsoFromInputObject (for IsCapCategory, IsFunction)

▷ `AddSimplifySource_IsoFromInputObject(C, F)` (operation)

▷ `AddSimplifySource_IsoFromInputObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SimplifySource_IsoFromInputObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (mor,n) \mapsto$ `SimplifySource`$_\mathtt{I}$`soFromInputObject`(mor,n).

7.6.247 AddSimplifySource_IsoToInputObject (for IsCapCategory, IsFunction)

▷ `AddSimplifySource_IsoToInputObject(C, F)` (operation)

▷ `AddSimplifySource_IsoToInputObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SimplifySource_IsoToInputObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (mor,n) \mapsto$ `SimplifySource`$_\mathtt{I}$`soToInputObject`(mor,n).

7.6.248 AddSolveLinearSystemInAbCategory (for IsCapCategory, IsFunction)

▷ `AddSolveLinearSystemInAbCategory(C, F)` (operation)
▷ `AddSolveLinearSystemInAbCategory(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SolveLinearSystemInAbCategory`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3, arg4) \mapsto$ `SolveLinearSystemInAbCategory`$(arg2, arg3, arg4)$.

7.6.249 AddSomeInjectiveObject (for IsCapCategory, IsFunction)

▷ `AddSomeInjectiveObject(C, F)` (operation)
▷ `AddSomeInjectiveObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SomeInjectiveObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `SomeInjectiveObject`$(arg2)$.

7.6.250 AddSomeIsomorphismBetweenObjects (for IsCapCategory, IsFunction)

▷ `AddSomeIsomorphismBetweenObjects(C, F)` (operation)
▷ `AddSomeIsomorphismBetweenObjects(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SomeIsomorphismBetweenObjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (object_1, object_2) \mapsto$ `SomeIsomorphismBetweenObjects`$(object_1, object_2)$.

7.6.251 AddSomeProjectiveObject (for IsCapCategory, IsFunction)

▷ `AddSomeProjectiveObject(C, F)` (operation)
▷ `AddSomeProjectiveObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SomeProjectiveObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2) \mapsto$ `SomeProjectiveObject`$(arg2)$.

7.6.252 AddSomeReductionBySplitEpiSummand (for IsCapCategory, IsFunction)

▷ `AddSomeReductionBySplitEpiSummand(C, F)` (operation)
▷ `AddSomeReductionBySplitEpiSummand(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SomeReductionBySplitEpiSummand`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ `SomeReductionBySplitEpiSummand`$(alpha)$.

7.6.253 AddSomeReductionBySplitEpiSummand_MorphismFromInputRange (for IsCapCategory, IsFunction)

▷ `AddSomeReductionBySplitEpiSummand_MorphismFromInputRange(C, F)` (operation)
▷ `AddSomeReductionBySplitEpiSummand_MorphismFromInputRange(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SomeReductionBySplitEpiSummand_MorphismFromInputRange`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ `SomeReductionBySplitEpiSummand`$_{\mathtt{M}}$`orphismFromInputRange`$(alpha)$.

7.6.254 AddSomeReductionBySplitEpiSummand_MorphismToInputRange (for IsCapCategory, IsFunction)

▷ `AddSomeReductionBySplitEpiSummand_MorphismToInputRange(C, F)` (operation)
▷ `AddSomeReductionBySplitEpiSummand_MorphismToInputRange(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SomeReductionBySplitEpiSummand_MorphismToInputRange`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha) \mapsto$ `SomeReductionBySplitEpiSummand`$_{\mathtt{M}}$`orphismToInputRange`$(alpha)$.

7.6.255 AddSubtractionForMorphisms (for IsCapCategory, IsFunction)

▷ `AddSubtractionForMorphisms(C, F)` (operation)
▷ `AddSubtractionForMorphisms(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SubtractionForMorphisms`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, beta) \mapsto$ `SubtractionForMorphisms`$(alpha, beta)$.

7.6.256 AddSumOfMorphisms (for IsCapCategory, IsFunction)

▷ `AddSumOfMorphisms(C, F)` (operation)
▷ `AddSumOfMorphisms(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `SumOfMorphisms`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (source, list_of_morphisms, range) \mapsto$ `SumOfMorphisms`$(source, list_of_morphisms, range)$.

7.6.257 AddTerminalObject (for IsCapCategory, IsFunction)

▷ `AddTerminalObject(C, F)` (operation)
▷ `AddTerminalObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `TerminalObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : () \mapsto$ `TerminalObject`$()$.

7.6.258 AddTerminalObjectFunctorial (for IsCapCategory, IsFunction)

▷ `AddTerminalObjectFunctorial(C, F)` (operation)
▷ `AddTerminalObjectFunctorial(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `TerminalObjectFunctorial`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : () \mapsto$ `TerminalObjectFunctorial`$()$.

7.6.259 AddTerminalObjectFunctorialWithGivenTerminalObjects (for IsCapCategory, IsFunction)

▷ `AddTerminalObjectFunctorialWithGivenTerminalObjects(C, F)` (operation)
▷ `AddTerminalObjectFunctorialWithGivenTerminalObjects(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `TerminalObjectFunctorialWithGivenTerminalObjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (P, Pp) \mapsto$ `TerminalObjectFunctorialWithGivenTerminalObjects`(P, Pp).

7.6.260 AddUniversalMorphismFromCoequalizer (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismFromCoequalizer(C, F)` (operation)
▷ `AddUniversalMorphismFromCoequalizer(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismFromCoequalizer`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (Y, morphisms, T, tau) \mapsto$ `UniversalMorphismFromCoequalizer`$(Y, morphisms, T, tau)$.

7.6.261 AddUniversalMorphismFromCoequalizerOfIdentityAndAutomorphisms (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismFromCoequalizerOfIdentityAndAutomorphisms(C, F)` (operation)
▷ `AddUniversalMorphismFromCoequalizerOfIdentityAndAutomorphisms(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismFromCoequalizerOfIdentityAndAutomorphisms`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (Y, automorphisms, T, tau) \mapsto$ `UniversalMorphismFromCoequalizerOfIdentityAndAutomorphisms`$(Y, automorphisms, T, tau)$.

7.6.262 AddUniversalMorphismFromCoequalizerOfIdentityAndAutomorphismsWithGivenCoequal (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismFromCoequalizerOfIdentityAndAutomorphismsWithGivenCoequalizer(C, F)` (operation)
▷ `AddUniversalMorphismFromCoequalizerOfIdentityAndAutomorphismsWithGivenCoequalizer(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismFromCoequalizerOfIdentityAndAutomorphismsWithGivenCoequalizer`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (Y, automorphisms, T, tau, P) \mapsto$ `UniversalMorphismFromCoequalizerOfIdentityAndAutomorphismsWithGivenCoequalizer`$(Y, automorphis$

7.6.263 AddUniversalMorphismFromCoequalizerWithGivenCoequalizer (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismFromCoequalizerWithGivenCoequalizer(C, F)` (operation)
▷ `AddUniversalMorphismFromCoequalizerWithGivenCoequalizer(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismFromCoequalizerWithGivenCoequalizer`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (Y, morphisms, T, tau, P) \mapsto$ `UniversalMorphismFromCoequalizerWithGivenCoequalizer`$(Y, morphisms, T, tau, P)$.

7.6.264 AddUniversalMorphismFromCoproduct (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismFromCoproduct(C, F)` (operation)
▷ `AddUniversalMorphismFromCoproduct(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismFromCoproduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (objects, T, tau) \mapsto$ `UniversalMorphismFromCoproduct`$(objects, T, tau)$.

7.6.265 AddUniversalMorphismFromCoproductWithGivenCoproduct (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismFromCoproductWithGivenCoproduct(C, F)` (operation)
▷ `AddUniversalMorphismFromCoproductWithGivenCoproduct(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismFromCoproductWithGivenCoproduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (objects, T, tau, P) \mapsto$ `UniversalMorphismFromCoproductWithGivenCoproduct`$(objects, T, tau, P)$.

7.6.266 AddUniversalMorphismFromDirectSum (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismFromDirectSum(C, F)` (operation)
▷ `AddUniversalMorphismFromDirectSum(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismFromDirectSum`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (objects, T, tau) \mapsto$ `UniversalMorphismFromDirectSum`$(objects, T, tau)$.

7.6.267 AddUniversalMorphismFromDirectSumWithGivenDirectSum (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismFromDirectSumWithGivenDirectSum(C, F)` (operation)
▷ `AddUniversalMorphismFromDirectSumWithGivenDirectSum(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismFromDirectSumWithGivenDirectSum`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (objects, T, tau, P) \mapsto$ `UniversalMorphismFromDirectSumWithGivenDirectSum`$(objects, T, tau, P)$.

7.6.268 AddUniversalMorphismFromImage (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismFromImage(C, F)` (operation)
▷ `AddUniversalMorphismFromImage(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismFromImage`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, tau) \mapsto$ `UniversalMorphismFromImage`$(alpha, tau)$.

7.6.269 AddUniversalMorphismFromImageWithGivenImageObject (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismFromImageWithGivenImageObject(C, F)` (operation)
▷ `AddUniversalMorphismFromImageWithGivenImageObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismFromImageWithGivenImageObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha, tau, I) \mapsto$ `UniversalMorphismFromImageWithGivenImageObject`$(alpha, tau, I)$.

7.6.270 AddUniversalMorphismFromInitialObject (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismFromInitialObject(C, F)` (operation)
▷ `AddUniversalMorphismFromInitialObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismFromInitialObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (T) \mapsto$ `UniversalMorphismFromInitialObject`(T).

7.6.271 AddUniversalMorphismFromInitialObjectWithGivenInitialObject (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismFromInitialObjectWithGivenInitialObject(C, F)` (operation)
▷ `AddUniversalMorphismFromInitialObjectWithGivenInitialObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismFromInitialObjectWithGivenInitialObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (T, P) \mapsto$ `UniversalMorphismFromInitialObjectWithGivenInitialObject`(T, P).

7.6.272 AddUniversalMorphismFromPushout (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismFromPushout(C, F)` (operation)
▷ `AddUniversalMorphismFromPushout(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismFromPushout`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (morphisms, T, tau) \mapsto$ `UniversalMorphismFromPushout`$(morphisms, T, tau)$.

7.6.273 AddUniversalMorphismFromPushoutWithGivenPushout (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismFromPushoutWithGivenPushout(C, F)` (operation)
▷ `AddUniversalMorphismFromPushoutWithGivenPushout(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismFromPushoutWithGivenPushout`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (morphisms, T, tau, P) \mapsto$ `UniversalMorphismFromPushoutWithGivenPushout`$(morphisms, T, tau, P)$.

7.6.274 AddUniversalMorphismFromZeroObject (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismFromZeroObject(C, F)` (operation)
▷ `AddUniversalMorphismFromZeroObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismFromZeroObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (T) \mapsto$ `UniversalMorphismFromZeroObject`(T).

7.6.275 AddUniversalMorphismFromZeroObjectWithGivenZeroObject (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismFromZeroObjectWithGivenZeroObject(C, F)` (operation)
▷ `AddUniversalMorphismFromZeroObjectWithGivenZeroObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismFromZeroObjectWithGivenZeroObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (T, P) \mapsto$ `UniversalMorphismFromZeroObjectWithGivenZeroObject`(T, P).

7.6.276 AddUniversalMorphismIntoCoimage (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismIntoCoimage(C, F)` (operation)
▷ `AddUniversalMorphismIntoCoimage(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismIntoCoimage`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha,tau) \mapsto$ `UniversalMorphismIntoCoimage`$(alpha,tau)$.

7.6.277 AddUniversalMorphismIntoCoimageWithGivenCoimageObject (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismIntoCoimageWithGivenCoimageObject(C, F)` (operation)
▷ `AddUniversalMorphismIntoCoimageWithGivenCoimageObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismIntoCoimageWithGivenCoimageObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (alpha,tau,C) \mapsto$ `UniversalMorphismIntoCoimageWithGivenCoimageObject`$(alpha,tau,C)$.

7.6.278 AddUniversalMorphismIntoDirectProduct (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismIntoDirectProduct(C, F)` (operation)
▷ `AddUniversalMorphismIntoDirectProduct(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismIntoDirectProduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (objects,T,tau) \mapsto$ `UniversalMorphismIntoDirectProduct`$(objects,T,tau)$.

7.6.279 AddUniversalMorphismIntoDirectProductWithGivenDirectProduct (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismIntoDirectProductWithGivenDirectProduct(C, F)` (operation)
▷ `AddUniversalMorphismIntoDirectProductWithGivenDirectProduct(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismIntoDirectProductWithGivenDirectProduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (objects,T,tau,P) \mapsto$ `UniversalMorphismIntoDirectProductWithGivenDirectProduct`$(objects,T,tau,P)$.

7.6.280 AddUniversalMorphismIntoDirectSum (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismIntoDirectSum(C, F)` (operation)
▷ `AddUniversalMorphismIntoDirectSum(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismIntoDirectSum`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (objects, T, tau) \mapsto$ `UniversalMorphismIntoDirectSum`$(objects, T, tau)$.

7.6.281 AddUniversalMorphismIntoDirectSumWithGivenDirectSum (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismIntoDirectSumWithGivenDirectSum(C, F)` (operation)
▷ `AddUniversalMorphismIntoDirectSumWithGivenDirectSum(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismIntoDirectSumWithGivenDirectSum`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (objects, T, tau, P) \mapsto$ `UniversalMorphismIntoDirectSumWithGivenDirectSum`$(objects, T, tau, P)$.

7.6.282 AddUniversalMorphismIntoEqualizer (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismIntoEqualizer(C, F)` (operation)
▷ `AddUniversalMorphismIntoEqualizer(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismIntoEqualizer`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (Y, morphisms, T, tau) \mapsto$ `UniversalMorphismIntoEqualizer`$(Y, morphisms, T, tau)$.

7.6.283 AddUniversalMorphismIntoEqualizerWithGivenEqualizer (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismIntoEqualizerWithGivenEqualizer(C, F)` (operation)
▷ `AddUniversalMorphismIntoEqualizerWithGivenEqualizer(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismIntoEqualizerWithGivenEqualizer`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (Y, morphisms, T, tau, P) \mapsto$ `UniversalMorphismIntoEqualizerWithGivenEqualizer`$(Y, morphisms, T, tau, P)$.

7.6.284 AddUniversalMorphismIntoFiberProduct (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismIntoFiberProduct(C, F)` (operation)
▷ `AddUniversalMorphismIntoFiberProduct(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismIntoFiberProduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (morphisms, T, tau) \mapsto$ `UniversalMorphismIntoFiberProduct`$(morphisms, T, tau)$.

7.6.285 AddUniversalMorphismIntoFiberProductWithGivenFiberProduct (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismIntoFiberProductWithGivenFiberProduct(C, F)` (operation)
▷ `AddUniversalMorphismIntoFiberProductWithGivenFiberProduct(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismIntoFiberProductWithGivenFiberProduct`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (morphisms, T, tau, P) \mapsto$ `UniversalMorphismIntoFiberProductWithGivenFiberProduct`$(morphisms, T, tau, P)$.

7.6.286 AddUniversalMorphismIntoTerminalObject (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismIntoTerminalObject(C, F)` (operation)
▷ `AddUniversalMorphismIntoTerminalObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismIntoTerminalObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (T) \mapsto$ `UniversalMorphismIntoTerminalObject`(T).

7.6.287 AddUniversalMorphismIntoTerminalObjectWithGivenTerminalObject (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismIntoTerminalObjectWithGivenTerminalObject(C, F)` (operation)
▷ `AddUniversalMorphismIntoTerminalObjectWithGivenTerminalObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismIntoTerminalObjectWithGivenTerminalObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (T, P) \mapsto$ `UniversalMorphismIntoTerminalObjectWithGivenTerminalObject`(T, P).

7.6.288 AddUniversalMorphismIntoZeroObject (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismIntoZeroObject(C, F)` (operation)
▷ `AddUniversalMorphismIntoZeroObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismIntoZeroObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (T) \mapsto$ `UniversalMorphismIntoZeroObject`(T).

7.6.289 AddUniversalMorphismIntoZeroObjectWithGivenZeroObject (for IsCapCategory, IsFunction)

▷ `AddUniversalMorphismIntoZeroObjectWithGivenZeroObject(C, F)` (operation)
▷ `AddUniversalMorphismIntoZeroObjectWithGivenZeroObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `UniversalMorphismIntoZeroObjectWithGivenZeroObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (T,P) \mapsto$ `UniversalMorphismIntoZeroObjectWithGivenZeroObject`(T,P).

7.6.290 AddVerticalPostCompose (for IsCapCategory, IsFunction)

▷ `AddVerticalPostCompose(C, F)` (operation)
▷ `AddVerticalPostCompose(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `VerticalPostCompose`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `VerticalPostCompose`$(arg2, arg3)$.

7.6.291 AddVerticalPreCompose (for IsCapCategory, IsFunction)

▷ `AddVerticalPreCompose(C, F)` (operation)
▷ `AddVerticalPreCompose(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `VerticalPreCompose`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (arg2, arg3) \mapsto$ `VerticalPreCompose`$(arg2, arg3)$.

7.6.292 AddZeroMorphism (for IsCapCategory, IsFunction)

▷ `AddZeroMorphism(C, F)` (operation)
▷ `AddZeroMorphism(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ZeroMorphism`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (a,b) \mapsto$ `ZeroMorphism`(a,b).

7.6.293 AddZeroObject (for IsCapCategory, IsFunction)

▷ `AddZeroObject(C, F)` (operation)
▷ `AddZeroObject(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ZeroObject`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : () \mapsto$ `ZeroObject`$()$.

7.6.294 AddZeroObjectFunctorial (for IsCapCategory, IsFunction)

▷ `AddZeroObjectFunctorial(C, F)` (operation)
▷ `AddZeroObjectFunctorial(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ZeroObjectFunctorial`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : () \mapsto$ `ZeroObjectFunctorial`$()$.

7.6.295 AddZeroObjectFunctorialWithGivenZeroObjects (for IsCapCategory, IsFunction)

▷ `AddZeroObjectFunctorialWithGivenZeroObjects(C, F)` (operation)
▷ `AddZeroObjectFunctorialWithGivenZeroObjects(C, F, weight)` (operation)

Returns: nothing

The arguments are a category C and a function F. This operation adds the given function F to the category for the basic operation `ZeroObjectFunctorialWithGivenZeroObjects`. Optionally, a weight (default: 100) can be specified which should roughly correspond to the computational complexity of the function (lower weight = less complex = faster execution). $F : (P, Pp) \mapsto$ `ZeroObjectFunctorialWithGivenZeroObjects`(P, Pp).

Chapter 8

Managing Derived Methods

8.1 Info Class

8.1.1 DerivationInfo

▷ `DerivationInfo` (info class)

Info class for derivations.

8.1.2 ActivateDerivationInfo

▷ `ActivateDerivationInfo(`*`arg`*`)` (function)

8.1.3 DeactivateDerivationInfo

▷ `DeactivateDerivationInfo(`*`arg`*`)` (function)

8.2 Derivation Objects

8.2.1 IsDerivedMethod (for IsAttributeStoringRep)

▷ `IsDerivedMethod(`*`arg`*`)` (Category)

Returns: `true` or `false`

A derivation object describes a derived method. It contains information about which operation the derived method implements, and which other operations it relies on.

8.2.2 CreateDerivation

▷ `CreateDerivation(`*`target_op_name, description, used_ops_with_multiples, func, weight, category_filter, loop_multiplier, category_getters`*`)` (function)

Creates a new derivation object. The argument *`target_op_name`* is the name of the operation which the derived method implements. The argument *`description`* should describe the derivation. The argument *`used_ops_with_multiples`* contains

- the name of each operation used by the derived method,
- together with a positive integer specifying how many times that operation is used and
- either a category getter or `fail`.

This is given as a list of lists, where each sublist has as first entry the name of an operation, as second entry an integer and as third entry either a function or `fail`. This function should accept the category for which this derivation will be installed, and return a category for which the operation in the first entry must be installed for the derivation to be considered applicable. The argument *func* contains the actual implementation of the derived method. The argument *weight* is an additional number to add when calculating the resulting weight of the target operation using this derivation. Unless there is any particular reason to regard the derivation as exceedingly expensive, this number should be `1`. The argument *category_filter* is a filter (or function) describing which categories the derivation is valid for. If it is valid for all categories, then this argument should have the value `IsCapCategory`. The output of *category_filter* must not change during the installation of operations. In particular, it must not rely on `CanCompute` to check conditions.

8.2.3 Description (for IsDerivedMethod)

▷ `Description(d)` (attribute)

A description of the derivation.

8.2.4 AdditionalWeight (for IsDerivedMethod)

▷ `AdditionalWeight(d)` (attribute)

Additional weight for the derivation.

8.2.5 DerivationFunction (for IsDerivedMethod)

▷ `DerivationFunction(d)` (attribute)

The implementation of the derivation.

8.2.6 CategoryFilter (for IsDerivedMethod)

▷ `CategoryFilter(d)` (attribute)

Filter describing which categories the derivation is valid for.

8.2.7 IsApplicableToCategory (for IsDerivedMethod, IsCapCategory)

▷ `IsApplicableToCategory(d, C)` (operation)

Returns: `true` if the category *C* is known to satisfy the category filter of the derivation *d*.

Checks if the derivation is known to be valid for a given category.

8.2.8 TargetOperation (for IsDerivedMethod)

▷ `TargetOperation(d)` (attribute)

Returns: The name (as a string) of the operation implemented by the derivation *d*

8.2.9 UsedOperationsWithMultiplesAndCategoryGetters (for IsDerivedMethod)

▷ `UsedOperationsWithMultiplesAndCategoryGetters(d)` (attribute)

Returns: The names of the operations used by the derivation *d*, together with their multiplicities and category getters. The result is a list consisting of lists of the form `[op_name, mult, getter]`, where `op_name` is a string, `mult` a positive integer and `getter` is a function or `fail`.

8.2.10 InstallDerivationForCategory (for IsDerivedMethod, IsPosInt, IsCapCategory)

▷ `InstallDerivationForCategory(d, weight, C)` (operation)

Install the derived method *d* for the category *C*. The integer *weight* is the computed weight of the operation implemented by this derivation.

8.2.11 FunctionCalledBeforeInstallation (for IsDerivedMethod)

▷ `FunctionCalledBeforeInstallation(d)` (attribute)

Input is a derived method. Output is a unary function that takes as an input a category and does not output anything. This function is always called before the installation of the derived method for a concrete instance of a category.

8.3 Derivation Graphs

8.3.1 IsDerivedMethodGraph (for IsAttributeStoringRep)

▷ `IsDerivedMethodGraph(arg)` (Category)

Returns: `true` or `false`

A derivation graph consists of a set of operations and a set of derivations specifying how some operations can be implemented in terms of other operations.

8.3.2 MakeDerivationGraph (for IsDenseList)

▷ `MakeDerivationGraph(operations)` (operation)

Make a derivation graph containing the given set of operations and no derivations. The argument *operations* should be a list of strings, the names of the operations. The set of operations is fixed once the graph is created. Derivations can be added to the graph by calling `AddDerivation`.

8.3.3 AddOperationsToDerivationGraph (for IsDerivedMethodGraph, IsDenseList)

▷ `AddOperationsToDerivationGraph(graph, operations)` (operation)

Adds a list of operation names `operations` to a given derivation graph `graph`. This is used in extensions of CAP which want to have their own basic operations, but do not want to pollute the CAP kernel any more. Please use it with caution. If a weight list/category was created before it will not be aware of the operations.

8.3.4 AddDerivation

▷ AddDerivation(*graph, target_op, description, used_ops_with_multiples_and_category_getters, func, weight, category_filter, loop_multiplier, category_getters, function_called_before_installation*) (function)

Add a derivation to a derivation graph.

8.3.5 AddDerivationToCAP

▷ AddDerivationToCAP(*arg*) (function)

8.3.6 Operations (for IsDerivedMethodGraph)

▷ Operations(*G*) (attribute)

Gives the operations in the graph *G*, as a list of strings.

8.3.7 DerivationsUsingOperation (for IsDerivedMethodGraph, IsString)

▷ DerivationsUsingOperation(*G, op_name*) (operation)

Finds all the derivations in the graph *G* that use the operation named *op_name*, and returns them as a list.

8.3.8 DerivationsOfOperation (for IsDerivedMethodGraph, IsString)

▷ DerivationsOfOperation(*G, op_name*) (operation)

Finds all the derivations in the graph *G* targeting the operation named *op_name* (that is, the derivations that provide implementations of this operation), and returns them as a list.

8.4 Managing Derivations in a Category

8.4.1 IsOperationWeightList (for IsAttributeStoringRep)

▷ IsOperationWeightList(*arg*) (Category)

Returns: `true` or `false`

An operation weight list manages the use of derivations in a single category C. For every operation, it keeps a weight value which indicates how costly it is to perform that operation in the category C. Whenever a new operation is implemented in C, the operation weight list should be notified about this and given a weight to assign to this operation. It will then automatically install all possible derived methods for C in such a way that every operation has the smallest possible weight (the weight of a derived method is computed by using the weights of the operations it uses).

8.4.2 MakeOperationWeightList (for IsCapCategory, IsDerivedMethodGraph)

▷ `MakeOperationWeightList(C, G)` (operation)

Create the operation weight list for a category. This should only be done once for every category, and the category should afterwards remember the returned object. The argument `C` is the CAP category this operation weight list is associated to, and the argument `G` is a derivation graph containing operation names and derivations.

8.4.3 DerivationGraph (for IsOperationWeightList)

▷ `DerivationGraph(owl)` (attribute)

Returns the derivation graph used by the operation weight list `owl`.

8.4.4 CategoryOfOperationWeightList (for IsOperationWeightList)

▷ `CategoryOfOperationWeightList(owl)` (attribute)

Returns the CAP category associated to the operation weight list `owl`.

8.4.5 CurrentOperationWeight (for IsOperationWeightList, IsString)

▷ `CurrentOperationWeight(owl, op_name)` (operation)

Returns the current weight of the operation named `op_name`.

8.4.6 OperationWeightUsingDerivation (for IsOperationWeightList, IsDerivedMethod)

▷ `OperationWeightUsingDerivation(owl, d)` (operation)

Finds out what the weight of the operation implemented by the derivation `d` would be if we had used that derivation.

8.4.7 DerivationOfOperation (for IsOperationWeightList, IsString)

▷ `DerivationOfOperation(owl, op_name)` (operation)

Returns the derivation which is currently used to implement the operation named `op_name`. If the operation is not implemented by a derivation (that is, either implemented directly or not implemented at all), then `fail` is returned.

8.4.8 TriggerDerivationsUsingOperation (for IsOperationWeightList, IsString)

▷ `TriggerDerivationsUsingOperation(owl, op_name)` (operation)

Performs a search from the operation *op_name*, and triggers all derivations that give improvements over the current state. This is used internally by `InstallDerivations` and `Reevaluate`. It should normally not be necessary to call this function directly.

8.4.9 Reevaluate (for IsOperationWeightList)

▷ `Reevaluate(owl)` (operation)

Reevaluate the installed derivations, installing better derivations if possible. This should be called if new derivations become available for the category, either because the category has acquired more knowledge about itself (e.g. it is told that it is abelian) or because new derivations have been added to the graph.

8.4.10 Saturate (for IsOperationWeightList)

▷ `Saturate(owl)` (operation)

Saturates the derivation graph, i.e., calls reevaluate until no more changes in the derivation graph occur.

8.4.11 AddPrimitiveOperation (for IsOperationWeightList, IsString, IsInt)

▷ `AddPrimitiveOperation(owl, op_name, weight)` (operation)

Add the operation named *op_name* to the operation weight list *owl* with weight *weight*.

8.4.12 PrintDerivationTree (for IsOperationWeightList, IsString)

▷ `PrintDerivationTree(owl, op_name)` (operation)

Print a tree representation of the way the operation named *op_name* is implemented in the category of the operation weight list *owl*.

8.4.13 PrintTree (for IsObject, IsFunction, IsFunction)

▷ `PrintTree(arg1, arg2, arg3)` (operation)

Prints a tree structure.

Chapter 9

Technical Details

9.1 The Category Cat

9.1.1 ObjectCache (for IsCapFunctor)

▷ `ObjectCache(`*`functor`*`)` (attribute)

Returns: IsCachingObject

Retuns the caching object which stores the results of the functor *`functor`* applied to objects.

9.1.2 MorphismCache (for IsCapFunctor)

▷ `MorphismCache(`*`functor`*`)` (attribute)

Returns: IsCachingObject

Retuns the caching object which stores the results of the functor *`functor`* applied to morphisms.

9.2 Tools

9.2.1 FunctionWithNamedArguments

▷ `FunctionWithNamedArguments(`*`specification, func`*`)` (function)

Returns: a function

Simulates named arguments in GAP as follows:

- *`specification`* is a list of pairs with first entry the name of the argument and second entry a default value (which must be immutable).
- *`func`* must be a function with first argument `CAP_NAMED_ARGUMENTS`.
- The return value is a function with one argument fewer than *`func`*.

When calling the returned function, the arguments are passed on to *`func`*. To simulate named arguments, any GAP options appearing in *`specification`* are consumed and put into the record `CAP_NAMED_ARGUMENTS`.

9.2.2 CAP_INTERNAL_GET_DATA_TYPE_FROM_STRING

▷ `CAP_INTERNAL_GET_DATA_TYPE_FROM_STRING(`*`filter_string[, category]`*`)` (function)

Returns: a record

The function takes one of the strings listed under `filter_list` in 7.3 as input and returns the corresponding data type (see `CapJitInferredDataTypes` (**CompilerForCAP: CapJitInferredDataTypes**) for details). If no category is given, data types with generic filters (`IsCapCategoryObject`, `IsCapCategoryMorphism` etc.) are returned. However, those cannot be used in the context of `CompilerForCAP` because the component `category` cannot be set in this case.

9.2.3 CAP_INTERNAL_GET_DATA_TYPES_FROM_STRINGS

▷ `CAP_INTERNAL_GET_DATA_TYPES_FROM_STRINGS(`*`list_of_strings[, category]`*`)` (function)

Returns: a list

Applies `CAP_INTERNAL_GET_DATA_TYPE_FROM_STRING` (9.2.2) to all elements of *`list`* and returns the result.

9.2.4 CAP_INTERNAL_REPLACED_STRING_WITH_FILTER

▷ `CAP_INTERNAL_REPLACED_STRING_WITH_FILTER(`*`filter_string[, category]`*`)` (function)

Returns: a filter

The function takes one of the strings listed under `filter_list` in 7.3 as input. The corresponding filter of the category *`category`* is returned. If no category is given, generic filters (`IsCapCategoryObject`, `IsCapCategoryMorphism` etc.) are used.

9.2.5 CAP_INTERNAL_REPLACED_STRINGS_WITH_FILTERS

▷ `CAP_INTERNAL_REPLACED_STRINGS_WITH_FILTERS(`*`list[, category]`*`)` (function)

Returns: Replaced list

Applies `CAP_INTERNAL_REPLACED_STRING_WITH_FILTER` (9.2.4) to all elements of *`list`* and returns the result.

9.2.6 CAP_INTERNAL_RETURN_OPTION_OR_DEFAULT

▷ `CAP_INTERNAL_RETURN_OPTION_OR_DEFAULT(`*`string, value`*`)` (function)

Returns: option value

Returns the value of the option with name string, or, if this value is fail, the object value.

9.2.7 CAP_INTERNAL_FIND_APPEARANCE_OF_SYMBOL_IN_FUNCTION

▷ `CAP_INTERNAL_FIND_APPEARANCE_OF_SYMBOL_IN_FUNCTION(`*`function, symbol_list, loop_multiple, replacement_record`*`)` (function)

Returns: a list of symbols with multiples

The function searches for the appearance of the strings in symbol list on the function function and returns a list of pairs, containing the name of the symbol and the number of appearance. If the symbol appears in a loop, the number of appearance is counted times the loop multiple. Moreover, if appearances of found strings should be replaced by collections of other strings, then these can be specified in the replacement record.

9.2.8 CAP_INTERNAL_MERGE_PRECONDITIONS_LIST

▷ `CAP_INTERNAL_MERGE_PRECONDITIONS_LIST(list1, list2)` (function)

Returns: merge list

The function takes two lists containing pairs of symbols (strings) and multiples. The lists are merged that pairs where the string only appears in one list is then added to the return list, if a pair with a string appears in both lists, the resulting lists only contains this pair once, with the higher multiple from both lists.

9.2.9 CAP_INTERNAL_ASSERT_VALUE_IS_OF_TYPE_GETTER

▷ `CAP_INTERNAL_ASSERT_VALUE_IS_OF_TYPE_GETTER(data_type, human_readable_identifier_getter)` (function)

Returns: a function

Returns a function `f` which throws an error if its first argument is not of type *data_type*. *human_readable_identifier_getter* is a function returning a string which is used to refer to the first argument of `f` in the error message. The arguments of `f` except the first argument are passed on to *human_readable_identifier_getter*.

9.2.10 CAP_INTERNAL_ASSERT_IS_CELL_OF_CATEGORY

▷ `CAP_INTERNAL_ASSERT_IS_CELL_OF_CATEGORY(cell, category, human_readable_identifier_getter)` (function)

The function throws an error if *cell* is not a cell of *category*. If *category* is the boolean `false`, only general checks not specific to a concrete category are performed. *human_readable_identifier_getter* is a 0-ary function returning a string which is used to refer to *cell* in the error message.

9.2.11 CAP_INTERNAL_ASSERT_IS_OBJECT_OF_CATEGORY

▷ `CAP_INTERNAL_ASSERT_IS_OBJECT_OF_CATEGORY(object, category, human_readable_identifier_getter)` (function)

The function throws an error if *object* is not an object of *category*. If *category* is the boolean `false`, only general checks not specific to a concrete category are performed. *human_readable_identifier_getter* is a 0-ary function returning a string which is used to refer to *cell* in the error message.

9.2.12 CAP_INTERNAL_ASSERT_IS_MORPHISM_OF_CATEGORY

▷ `CAP_INTERNAL_ASSERT_IS_MORPHISM_OF_CATEGORY(morphism, category, human_readable_identifier_getter)` (function)

The function throws an error if *morphism* is not a morphism of *category*. If *category* is the boolean `false`, only general checks not specific to a concrete category are performed. *human_readable_identifier_getter* is a 0-ary function returning a string which is used to refer to *cell* in the error message.

9.2.13 CAP_INTERNAL_ASSERT_IS_TWO_CELL_OF_CATEGORY

▷ `CAP_INTERNAL_ASSERT_IS_TWO_CELL_OF_CATEGORY(`*`two_cell, category, human_readable_identifier_getter`*`)` (function)

The function throws an error if *`two_cell`* is not a 2-cell of *`category`*. If *`category`* is the boolean `false`, only general checks not specific to a concrete category are performed. *`human_readable_identifier_getter`* is a 0-ary function returning a string which is used to refer to *`cell`* in the error message.

9.2.14 CachingStatistic

▷ `CachingStatistic(`*`category[, operation]`*`)` (function)

Prints statistics for all caches in *`category`*. If *`operation`* is given (as a string), only statistics for the given operation cache is stored.

9.2.15 BrowseCachingStatistic

▷ `BrowseCachingStatistic(`*`category`*`)` (function)

Displays statistics for all caches in *`category`*. in a Browse window. Here "status" indicates if the cache is weak, strong, or inactive, "hits" is the number of successful cache accesses, "misses" the number of unsuccessful cache accesses, and "stored" the number of objects currently stored in the cache.

9.2.16 InstallDeprecatedAlias

▷ `InstallDeprecatedAlias(`*`alias_name, function_name, deprecation_date`*`)` (function)

Makes the function given by *`function_name`* available under the alias *`alias_name`* with a deprecation warning including the date *`deprecation_date`*.

9.2.17 IsSpecializationOfFilter

▷ `IsSpecializationOfFilter(`*`filter1, filter2`*`)` (function)

Checks if *`filter2`* is more special than *`filter1`*, i.e. if *`filter2`* implies *`filter1`*. *`filter1`* and/or *`filter2`* can also be one of the strings listed under `filter_list` in 7.3 and in this case are replaced by the corresponding filters (e.g. `IsCapCategory`, `IsCapCategoryObject`, `IsCapCategoryMorphism`, ...).

9.2.18 IsSpecializationOfFilterList

▷ `IsSpecializationOfFilterList(`*`filter_list1, filter_list2`*`)` (function)

Checks if *`filter_list2`* is more special than *`filter_list1`*, i.e. if both lists have the same length and any element of *`filter_list2`* is more special than the corresponding element of *filter_list1* in the sense of `IsSpecializationOfFilter` (9.2.17). *filter_list1* and *filter_list2* can also be the string `"any"`, respresenting a most general filter list of any length.

9.2.19 InstallMethodForCompilerForCAP

▷ `InstallMethodForCompilerForCAP(same, as, for, InstallMethod)` (function)

Installs a method via `InstallMethod` and adds it to the list of methods known to the compiler. See `CapJitAddKnownMethod` (9.2.21) for requirements.

9.2.20 InstallOtherMethodForCompilerForCAP

▷ `InstallOtherMethodForCompilerForCAP(same, as, for, InstallOtherMethod)` (function)

Installs a method via `InstallOtherMethod` and adds it to the list of methods known to the compiler. See `CapJitAddKnownMethod` (9.2.21) for requirements.

9.2.21 CapJitAddKnownMethod

▷ `CapJitAddKnownMethod(operation, filters, method)` (function)

Adds a method to the list of methods known to the compiler. If the first filter implies `IsCapCategory`, method selection only takes the number of arguments and the first filter into account. This allows to resolve operations even in the case that the syntax tree cannot fully be typed. If the first filter does not imply `IsCapCategory`, method selection takes all filters into account. To strictly distinguish between the two cases, `IsCapCategory` must not imply the first filter (except if the first filter is equal to `IsCapCategory`). Method selection is strict in the sense that two different methods for the same operation must not be comparable. That is, they must have a different number of filters or the filters at at least one position must not be related via implication. In particular, adding two methods with a CAP category as first argument (or a convenience method for a CAP operation) with the same number of arguments and one category filter implying the other is not supported.

9.2.22 CapJitAddTypeSignature

▷ `CapJitAddTypeSignature(name, input_filters, output_data_type)` (function)

(experimental) Adds a type signature for the global function or operation given by *name* to the compiler. *input_filters* must be a list of filters, or the string '"any"' representing a most general filter list of any length. *output_data_type* must be a filter, a data type, or a function. If it is a function with one argument, it must accept a list of input types and return the corresponding data type of the output. If it is a function with two arguments, it must accept the arguments of a function call of *name* (as syntax trees) and the function stack and return a record with components `args` (the possibly modified arguments) and `output_type` (the data type of the output). See `CapJitInferredDataTypes` (**CompilerForCAP: CapJitInferredDataTypes**) for more details on data types.

9.2.23 CapJitDataTypeOfListOf

▷ `CapJitDataTypeOfListOf(element_type)` (function)

(experimental) Returns the data type of a list whose elements are of type *element_type*. *element_type* must be a filter or a data type.

9.2.24 CapJitDataTypeOfNTupleOf

▷ `CapJitDataTypeOfNTupleOf(n, element_types...)` (function)

(experimental) Returns the data type of an *n*-tuple whose entries are of types corresponding to *element_types*. *element_types...* must be filters or data types.

9.2.25 CapJitDataTypeOfGroup

▷ `CapJitDataTypeOfGroup(group)` (function)
▷ `CapJitDataTypeOfSubgroup(group)` (function)
▷ `CapJitDataTypeOfElementOfGroup(group)` (function)

(experimental) Returns the data type of the group (or elements of the group) *group*.

9.2.26 CapJitDataTypeOfRing

▷ `CapJitDataTypeOfRing(ring)` (function)
▷ `CapJitDataTypeOfElementOfRing(ring)` (function)

(experimental) Returns the data type of the ring (or elements of the ring) *ring*.

9.2.27 CapJitDataTypeOfCategory

▷ `CapJitDataTypeOfCategory(category)` (function)
▷ `CapJitDataTypeOfObjectOfCategory(category)` (function)
▷ `CapJitDataTypeOfMorphismOfCategory(category)` (function)
▷ `CapJitDataTypeOfTwoCellOfCategory(category)` (function)

(experimental) Returns the data type of the category (or objects, morphisms, or two cells in the category) *category*.

9.2.28 CapJitTypedExpression

▷ `CapJitTypedExpression(value, data_type_getter)` (function)

(experimental) Simply returns *value*, but allows to specify the data type of *value* for CompilerForCAP. *data_type_getter* must be a literal function or a global variable pointing to a function. The function must accept no arguments or a single argument, and return a valid data type. If the function accepts a single argument, it must be inside a CAP operation or method known to CompilerForCAP (for example, see `InstallMethodForCompilerForCAP` (9.2.19)), and the current category (i.e. the first argument of the CAP operation or method known to CompilerForCAP) will be passed to the function. IMPORTANT: If *data_type_getter* is a literal function, it must not contain references to variables in its context. Otherwise the code might access random memory locations. See CapJitInferredDataTypes (**CompilerForCAP: CapJitInferredDataTypes**) for more details on data types.

9.2.29 CapFixpoint

▷ CapFixpoint(*predicate, func, initial_value*) (function)

Computes a fixpoint of *func* with regard to equality given by *predicate*, starting with *initial_value*. More precisely, CapFixpoint(*predicate*, *func*, *initial_value*) terminates and returns y once *predicate*(x, y) returns true, where y := *func*(x). If no such fixpoint exists, the execution does not terminate.

9.2.30 Iterated (for IsList, IsFunction, IsObject)

▷ Iterated(*list, func, initial_value*) (operation)

This input is a homogeneous list of elements, i.e., having the same type, a bivariate function *func*, and an object *initial_value*. The type of the first argument of *func* must be equal to the type of *initial_value*. The type of the second argument of *func* must be equal to the type of the entries of *list*. The type of the output of *func* must be equal to the type of *initial_value*. The installed method simply calls Iterated(Concatenation([<A>initial_value</A>], <A>list</A>), <A>func</A>). In particular, if *list* is empty, the output will be *initial_value*. This allows for the implementation of compilable initialized for-loops in CAP.

9.2.31 Iterated (for IsList, IsFunction, IsObject, IsObject)

▷ Iterated(*list, func, initial_value, terminal_value*) (operation)

This input is a homogeneous list of elements, i.e., having the same type, a bivariate function *func*, an object *initial_value*, and another object *terminal_entry*. The type of the *terminal_entry* must be equal to the type of the entries of *list*. The type of the first argument of *func* must be equal to the type of *initial_value*. The type of the second argument of *func* must be equal to the type of the entries of *list*. The type of the output of *func* must be equal to the type of *initial_value*. The installed method simply calls Iterated(Concatenation([<A>initial_value</A>], <A>list</A>, [<A>terminal_entry</A>]), <A>func</A>).

9.2.32 TransitivelyNeededOtherPackages

▷ TransitivelyNeededOtherPackages(*package_name*) (function)

Returns a list of package names which are transitively needed other packages of the package *package_name*.

9.2.33 PackageOfCAPOperation

▷ `PackageOfCAPOperation(`*`operation_name`*`)` (function)

Returns the name of the package to which the CAP operation given by *`operation_name`* belongs or fail if the package is not known.

9.2.34 SafePosition (for IsList, IsObject)

▷ `SafePosition(`*`list, obj`*`)` (operation)

Returns: an integer

Returns `Position( <A>list</A>, <A>obj</A> )` while asserting that this value is not `fail`.

9.2.35 SafeUniquePosition (for IsList, IsObject)

▷ `SafeUniquePosition(`*`list, obj`*`)` (operation)

Returns: an integer

Returns `Position( <A>list</A>, <A>obj</A> )` while asserting that this value is not `fail` and the position is unique.

9.2.36 SafePositionProperty (for IsList, IsFunction)

▷ `SafePositionProperty(`*`list, func`*`)` (operation)

Returns: an integer

Returns `PositionProperty( <A>list</A>, <A>func</A> )` while asserting that this value is not `fail`.

9.2.37 SafeUniquePositionProperty (for IsList, IsFunction)

▷ `SafeUniquePositionProperty(`*`list, func`*`)` (operation)

Returns: an integer

Returns a position in *`list`* for which *`func`* returns `true` when applied to the corresponding entry while asserting that there exists exactly one such position.

9.2.38 SafeFirst (for IsList, IsFunction)

▷ `SafeFirst(`*`list, func`*`)` (operation)

Returns: an element of the list

Returns `First( <A>list</A>, <A>func</A> )` while asserting that this value is not `fail`.

9.2.39 SafeUniqueEntry (for IsList, IsFunction)

▷ `SafeUniqueEntry(`*`list, func`*`)` (operation)

Returns: an element of the list

Returns a value in *`list`* for which *`func`* returns `true` while asserting that there exists exactly one such entry.

9.2.40 NTuple

▷ `NTuple(n, args...)` (function)

Returns: a list

Returns *args* while asserting that its length is *n*.

9.2.41 Pair

▷ `Pair(first, second)` (function)

Returns: a list

Alias for `NTuple( 2, <A>first</A>, <A>second</A> )`.

9.2.42 Triple

▷ `Triple(first, second, third)` (function)

Returns: a list

Alias for `NTuple( 3, <A>first</A>, <A>second</A>, <A>third</A> )`.

9.2.43 TransposedMatWithGivenDimensions

▷ `TransposedMatWithGivenDimensions(nr_rows, nr_cols, listlist)` (function)

Returns: a list (of lists)

The arguments are two integers *nr_rows*, *nr_cols* and a list of lists *listlist* such that *nr_rows* = Length(*listlist*) and *nr_cols* = Length(*listlist*[i]) for $i = 1$ to *nr_rows*. The output is the transpose of *listlist* as a list consisting of *nr_cols* rows and *nr_rows* columns.

9.2.44 HandlePrecompiledTowers

▷ `HandlePrecompiledTowers(category, underlying_category, constructor_name)` (function)

Handles the information stored in `<A>underlying_category</A>!.compiler_hints.precompiled_towers` (if bound) which is a list of records with components:

- `remaining_constructors_in_tower`: a non-empty list of strings (names of category constructors)
- `precompiled_functions_adder`: a function accepting a CAP category as input

If *constructor_name* is the only entry of `remaining_constructors_in_tower`, `precompiled_functions_adder` is applied to *category* (except if the option `no_precompiled_code` is set to `true`) and should add precompiled code. Else, if *constructor_name* is the first entry of `remaining_constructors_in_tower`, the information is attached to `<A>category</A>!.compiler_hints.precompiled_towers` after removing *constructor_name* from `remaining_constructors_in_tower`. Note: Currently, there is no logic for finding the "optimal" code to install if *constructor_name* is the only entry of `remaining_constructors_in_tower` of multiple entries.

9.2.45 CAP_JIT_INCOMPLETE_LOGIC

▷ CAP_JIT_INCOMPLETE_LOGIC(*value*) (function)

Simply returns *value*. Used to signify that the argument is not fully run through all logic functions/templates by CompilerForCAP.

9.2.46 CAP_JIT_EXPR_CASE_WRAPPER

▷ CAP_JIT_EXPR_CASE_WRAPPER(*func*) (function)

Simply returns *func*, which must be a literal function without arguments only containing an if-elif-else statement with each branch consisting of a single return statement. Used to write expressions of the form function() if-elif-else end() as CAP_JIT_EXPR_CASE_WRAPPER(function() if-elif-else end)() because the former is not valid in Julia.

9.2.47 ListWithKeys

▷ ListWithKeys(*list, func*) (function)

Returns: a list

Same as List(<A>list</A>, <A>func</A>) but *func* gets both the key i and <A>list</A>[i] as arguments.

9.2.48 SumWithKeys

▷ SumWithKeys(*list, func*) (function)

Returns: a list

Same as Sum(<A>list</A>, <A>func</A>) but *func* gets both the key i and <A>list</A>[i] as arguments.

9.2.49 ProductWithKeys

▷ ProductWithKeys(*list, func*) (function)

Returns: a list

Same as Product(<A>list</A>, <A>func</A>) but *func* gets both the key i and <A>list</A>[i] as arguments.

9.2.50 ForAllWithKeys

▷ ForAllWithKeys(*list, func*) (function)

Returns: a list

Same as ForAll(<A>list</A>, <A>func</A>) but *func* gets both the key i and <A>list</A>[i] as arguments.

9.2.51 ForAnyWithKeys

▷ `ForAnyWithKeys(list, func)` (function)

Returns: a list

Same as `ForAny( <A>list</A>, <A>func</A> )` but *func* gets both the key `i` and `<A>list</A>[i]` as arguments.

9.2.52 NumberWithKeys

▷ `NumberWithKeys(list, func)` (function)

Returns: a list

Same as `Number( <A>list</A>, <A>func</A> )` but *func* gets both the key `i` and `<A>list</A>[i]` as arguments.

9.2.53 FilteredWithKeys

▷ `FilteredWithKeys(list, func)` (function)

Returns: a list

Same as `Filtered( <A>list</A>, <A>func</A> )` but *func* gets both the key `i` and `<A>list</A>[i]` as arguments.

9.2.54 FirstWithKeys

▷ `FirstWithKeys(list, func)` (function)

Returns: a list

Same as `First( <A>list</A>, <A>func</A> )` but *func* gets both the key `i` and `<A>list</A>[i]` as arguments.

9.2.55 LastWithKeys

▷ `LastWithKeys(list, func)` (function)

Returns: a list

Same as `Last( <A>list</A>, <A>func</A> )` but *func* gets both the key `i` and `<A>list</A>[i]` as arguments.

9.2.56 CreateGapObjectWithAttributes

▷ `CreateGapObjectWithAttributes(type[, attribute1, value1, ...])` (function)

Shorthand for `ObjectifyWithAttributes( rec( ), type, [attribute1, value1, ...] )`.

Chapter 10

Limits and Colimits

This section describes the support for limits and colimits in CAP. All notions defined in the following are considered with regard to limits, not colimits, except if explicitly stated otherwise. In particular, the diagram specification specifies a diagram over which the limit is taken. The colimit in turn is taken over the opposite diagram.

10.1 Specification of Limits and Colimits

A record specifying a limit in CAP has the following entries:

- object_specification: see below
- morphism_specifiation: see below
- limit_object_name: the name of the method returning the limit object, e.g. `DirectProduct` or `KernelObject`
- limit_projection_name (optional): the name of the method returning the projection(s) from the limit object, e.g. `ProjectionInFactorOfDirectProduct` or `KernelEmbedding`. Defaults to `Concatenation( "ProjectionInFactorOf", limit_object_name )`.
- limit_universal_morphism_name (optional): the name of the method returning the universal morphism into the limit object, e.g. `UniversalMorphismIntoDirectProduct` or `KernelLift`. Defaults to `Concatenation( "UniversalMorphismInto", limit_object_name )`.
- colimit_object_name: the name of the method returning the colimit object, e.g. `Coproduct` or `CokernelObject`
- colimit_injection_name (optional): the name of the method returning the injection(s) into the colimit object, e.g. `InjectionOfCofactorOfCoproduct` or `CokernelProjection`. Defaults to `Concatenation( "InjectionOfCofactorOf", colimit_object_name )`.
- colimit_universal_morphism_name (optional): the name of the method returning the universal morphism from the colimit object, e.g. `UniversalMorphismFromCoproduct` or `CokernelColift`. Defaults to `Concatenation( "UniversalMorphismFrom", colimit_object_name )`.

limit_object_name and colimit_object_name can be the same, e.g. for `DirectSum` or `ZeroObject`.

The object_specification and morphism_specification together specify the shape of the diagram defining the limit or colimit. The syntax is the following:

- object_specification is a list of strings. Only the strings "fixedobject" and "varobject" are allowed as entries of the list. These are called "types" in the following.
- morphism_specification is a list of triples. The first and third entry of a triple are integers greater or equal to 1 and less or equal to `Length( object_specification )`. The second entry is one of the following strings: "fixedmorphism", "varmorphism", "zeromorphism". This entry is called "type" in the following.

Semantics is given as follows:

- The type "fixedobject" specifies a single object. The type "varobject" specifies arbitrarily many objects.
- The first and the third entry of a triple specify the source and range of a morphism (or multiple morphisms) encoded by the position in object_specification respectively. The type "fixedmorphism" specifies a single morphism. In this case, source and range can only be of type "fixedobject", not of type "varobject". The type "varmorphism" specifies arbitrarily many morphisms. In this case, if the source (resp. range) is of type "fixedobject" all the morphisms must have the same source (resp. range). On the contrary, if the source (resp. range) is of the type "varobject", the objects correspond one-to-one to the sources (resp. ranges) of the morphisms. The type "zeromorphism" is currently ignored but will be endowed with semantics in the future.

For example, a FiberProduct diagram consists of arbitrarily many morphisms which have arbitrary sources but the same common range. This can be expressed as follows:

Code
```
rec(
  object_specification := [ "fixedobject", "varobject" ],
  morphism_specification := [ [ 2, "varmorphism", 1 ] ],
  limit_object_name := "FiberProduct",
  colimit_object_name := "Pushout",
)
```

Note that not all diagrams which can be expressed with the above are actually supported. For now, at most one unbound object (see below for the definition of "unbound") may be of type "varobject", and if there is such an unbound object it must be the last one among the unbound objects. Similarly, at most one unbound morphism may be of type "varmorphism", and if there is such an unbound morphism it must be the last one among the unbound morphisms.

10.2 Enhancing Limit Specifications

The function CAP_INTERNAL_ENHANCE_NAME_RECORD_LIMITS takes a list of limits (given by records as explained above), and computes some additional properties. For example, the number of so-called unbound objects, unbound morphisms and (non-)targets is computed. The term "unbound" signifies that for creating a concrete diagram, these objects or morphisms have to be specified by the user because they cannot be derived by CAP:

- Unbound morphisms are the triples which are of type "fixedmorphism" or "varmorphism".
- Unbound objects are the objects which are not source or range of an unbound morphism.

Finally, targets are the objects which are not the range of a morphism. These are of interest for the following reason: for limits, only projections into targets are relevant because the projections into other objects can simply be computed by composition. Similarly, one only has to give morphisms into these targets to compute a universal morphism.

The number of unbound objects, unbound morphisms and (non-)targets is expressed by the integers 0, 1 and 2:

- 0: no such object/morphism/target exists
- 1: there exists exactly one such object/target of type "fixedobject" respectively exactly one such morphism of type "fixedmorphism"
- 2: else

10.3 Functions

10.3.1 CAP_INTERNAL_GENERATE_CONVENIENCE_METHODS_FOR_LIMITS

▷ `CAP_INTERNAL_GENERATE_CONVENIENCE_METHODS_FOR_LIMITS(`*`package_name, method_name_record, limits`*`)` (function)

This function takes a package name, a method name record and a list of enhanced limits, and generates convenience methods for the limits as a string of GAP code. The result is compared to the content of the file *`package_name`*`/gap/LimitConvenienceOutput.gi`. If a difference is found, a warning is raised and the generated string is written to a temporary file for manual inspection.

10.3.2 CAP_INTERNAL_VALIDATE_LIMITS_IN_NAME_RECORD

▷ `CAP_INTERNAL_VALIDATE_LIMITS_IN_NAME_RECORD(`*`method_name_record, limits`*`)` (function)

This function takes a method name record and a list of enhanced limits, and validates the entries of the method name record. Prefunctions, full prefunctions and postfunctions are excluded from the validation.

Chapter 11

The Category Constructor

11.1 Info class

11.1.1 InfoCategoryConstructor

▷ `InfoCategoryConstructor` (info class)

Info class controlling the debugging output of `CategoryConstructor` (11.2.1).

11.2 Constructors

11.2.1 CategoryConstructor (for IsRecord)

▷ `CategoryConstructor(`*options*`)` (operation)

Returns: a CAP category

Creates a CAP category subject to the options given via *options*, which is a record with the following keys:

- `name` (optional): name of the category
- `category_filter`, `category_object_filter`, `category_morphism_filter` (mandatory): passed to `CreateCapCategoryWithDataTypes` (1.3.4)
- `object_datum_type`, `morphism_datum_type` (optional): passed to `CreateCapCategoryWithDataTypes` (1.3.4)
- `commutative_semiring_of_linear_category` (optional): ring attached as `CommutativeSemiringOfLinearCategory` (1.4.9) to the category
- `range_category_of_homomorphism_structure` (optional): category attached as `RangeCategoryOfHomomorphismStructure` (1.4.10) to the category (or the string "self" to attach the category to itself)
- `properties` (optional): list of categorical properties the category will have, see `CAP_INTERNAL_CATEGORICAL_PROPERTIES_LIST`
- `object_constructor` (optional): function added as an installation of `ObjectConstructor` (2.7.1) to the category

- `object_datum` (optional): function added as an installation of `ObjectDatum` (2.7.3) to the category
- `morphism_constructor` (optional): function added as an installation of `MorphismConstructor` (3.3.1) to the category
- `morphism_datum` (optional): function added as an installation of `MorphismDatum` (3.3.2) to the category
- `list_of_operations_to_install` (mandatory): a list of names of CAP operations which should be installed for the category
- `is_computable` (optional): whether the category can decide `IsCongruentForMorphisms`
- `supports_empty_limits` (optional): whether the category supports empty lists in inputs to operations of limits and colimits
- `underlying_category_getter_string` (optional): see below
- `underlying_category` (optional): see below
- `underlying_object_getter_string` (optional): see below
- `underlying_morphism_getter_string` (optional): see below
- `top_object_getter_string` (optional): see below
- `top_morphism_getter_string` (optional): see below
- `generic_output_source_getter_string` (optional): see below
- `generic_output_range_getter_string` (optional): see below
- `create_func_bool`: see below
- `create_func_object`: see below
- `create_func_morphism`: see below
- `create_func_list_of_objects`: see below

The values of the keys `create_func_*` should be either the string `"default"` or functions which accept the category and the name of a CAP operation of the corresponding `return_type`. Values for return types occuring for operations in `list_of_operations_to_install` are mandatory. The functions must return either strings or pairs of a string and an integer: The strings (after some replacements described below) will be evaluated and added as installations of the corresponding operations to the category with weights given by the integer (if provided). The value `"default"` chooses a suitable default string (see the implementation for details) and gets the weights from `underlying_category`. The following placeholders may be used in the strings and are replaced automatically:

- `operation_name` will be replaced by the name of the operation
- `input_arguments...` will be replaced by the `input_arguments_names` specified in the method name record (see 7.3)

- `underlying_arguments...`: If the constructed category is created from another category, `underlying_category_getter_string`, `underlying_object_getter_string`, and `underlying_morphism_getter_string` may be strings of functions computing the underlying category (when applied to the constructed category) and the underlying object resp. morphism (when applied to the constructed category and an object resp. morphism in the constructed category). These functions are applied to `input_arguments` and `underlying_arguments` is replaced by the result.

- `number_of_arguments` will be replaced by the number of input/underlying arguments

- `top_source` and `top_range`: If the return type is `morphism`, source and range are computed if possible and `top_source` and `top_range` are replaced by the results. For computing source and range, the `output_source_getter_string` and `output_range_getter_string` from the method name record are used if available (see 7.3). In some categories, source and range can always be obtained in a generic way (e.g. from the morphism datum). In this case, `generic_output_source_getter_string` and `generic_output_range_getter_string` can be set and are used if the required information is not available in the method name record.

- `top_object_getter` and `top_morphism_getter` are used in the `"default"` strings and are replaced by `top_object_getter_string` and `top_morphism_getter_string`, respectively.

Chapter 12

Reinterpretations of categories

12.1 Introduction

The support for building towers of category constructors is one of the main design features of CAP. Many categories that appear in the various applications can be modeled by towers of multiple category constructors. The category constructor `ReinterpretationOfCategory` (12.6.1) allows adding one last layer on top which allows expressing the desired (re)interpretation of such a modeling tower. In particular, this category constructor allows specifying the name of the category together with customized methods for the operations

- ObjectConstructor
- MorphismConstructor
- ObjectDatum
- MorphismDatum

in order to reflect the desired (re)interpretation with a user-interface that is independent of the modeling tower (see below for details). Note that the same tower might have multiple interpretations.

`R := ReinterpretationOfCategory( cat_n )`
`cat_n := CategoryConstructor_n( cat_{n-1} )`
...
`cat_1 := CategoryConstructor_1( non_categorical_input )`

Table: A tower of categories modeling the category `R`

The reinterpretation `R` is isomorphic to the top category `cat_n` in the tower. In practice, the word "tower" stands more generally for a finite poset with a greatest element.

12.2 Tutorial

We will show how one can reinterpret a category with the following guiding example: We reinterpret `Opposite( CategoryOfRows( R ) )` as `CategoryOfColumns( R )` using `ReinterpretationOfCategory` (12.6.1) with the options described in the following (see `CategoryOfColumns_as_Opposite_CategoryOfRows.gi` in `AdditiveClosuresForCAP` for a full implementation).

1. Set the options `category_filter`, `category_object_filter`, and `category_morphism_filter` to the filters corresponding to the data structure of the desired reinterpretation, e.g. `IsCategoryOfColumns`, `IsCategoryOfColumsObject`, and `IsCategoryOfColumsMorphism`.

2. Set `object_constructor`, `object_datum`, `morphism_constructor`, and `morphism_datum` to the functions one would write for `ObjectConstructor` and so on for a primitive implementation of the desired reinterpretation. In our example, `object_constructor` takes the reinterpretation `R` (which lies in `IsCategoryOfColumns` due to the filter set in the first step) and an integer, and returns a CAP object in the category with attribute `RankOfObject` set to the integer, just like a primitive implementation of `CategoryOfColumns` would do.

3. Set `modeling_tower_object_constructor`, `modeling_tower_object_datum`, `modeling_tower_morphism_constructor`, and `modeling_tower_morphism_datum`: `modeling_tower_object_constructor` gets the same input as `object_constructor` but must return the corresponding object in the tower `cat_n`. `modeling_tower_object_datum` has the same output as `object_datum` but gets the reinterpretation `R` and an object in the tower `cat_n` as an input. In our example, `modeling_tower_object_constructor` gets the reinterpretation `R` and an integer as in step 2 and wraps the integer as a `CategoryOfRowsObject` and the result as an object in the opposite category. `modeling_tower_object_datum` gets the reinterpretation `R` and an object in `Opposite( CategoryOfRows( R ) )` (that is, an integer boxed as a category of rows object boxed as an object in the opposite category) and returns the underlying integer. `modeling_tower_morphism_constructor` and `modeling_tower_morphism_datum` are given analogously.

By composing `modeling_tower_object_datum` with `object_constructor` and `modeling_tower_morphism_datum` with `morphism_constructor` (with suitable source and range), `ReinterpretationOfCategory` defines a functor "Reinterpretation" from `cat_n` to `R`. Similarly, it defines a functor "Model" from `R` to `cat_n` by composing `object_datum` with `modeling_tower_object_constructor` and `morphism_datum` with `modeling_tower_morphism_constructor` (with suitable source and range). "Reinterpretation" should be an isomorphism of categories with inverse "Model". More precisely, one has to take care of the following things:

- Since `R` should just be a reinterpretation of `cat_n` with a nicer data structure, we certainly want "Reinterpretation" to be an equivalence of categories with pseudo-inverse "Model".

- `ReinterpretationOfCategory` copies all properties from `cat_n` to `R`, including properties like `IsSkeletalCategory` which are not necessarily preserved by mere equivalences.

- To fulfill the specification of WithGiven operations, reinterpreting a WithGiven object `A` in `cat_n` as an object in `R` and taking its model again must give an object equal to `A`. So we require applying "Reinterpretation" and then "Model" to give the identity. Conversely, let `B_1` be an object in `R`. We take its model and reinterpret this again to form an object `B_2`. By construction of `R`, `B_1` and `B_2` are equal if and only if their models are equal. But since applying "Reinterpretation" and then "Model" gives the identity, taking the model of `B_2` simply gives an object equal to the model of `B_1`. Thus, also `B_1` and `B_2` are equal. Hence, "Reinterpretation" has to be an equivalence which is a bijection on objects, and hence an isomorphism (although "Model" is not necessarily its inverse). Note: Alternatively, one can make sure that

WithGiven operations in `cat_n` are only called via the corresponding non-WithGiven operation in `cat_n`. This can be achieved by reinterpreting all operations of `cat_n` (i.e. creating `R` with `only_primitive_operations := false`), disabling redirect functions (i.e. creating `R` with the option `overhead := false`) and not calling WithGiven operations of `R` manually.

12.3 Implementation details

Operations in `ReinterpretationOfCategory` are implemented as follows:

1. Apply `object_datum` and `morphism_datum` to the input to get the underlying data.
2. Apply `modeling_tower_object_constructor` and `modeling_tower_morphism_constructor` to the underlying data to get objects and morphisms in the tower `cat_n`.
3. Apply the operation of the tower `cat_n`.
4. Apply `modeling_tower_object_datum` or `modeling_tower_morphism_datum` to the result to get the underlying data.
5. Apply `object_constructor` or `morphism_constructor` to the underlying data to get an object or a morphism in the reinterpretation `R`.

The first two steps define the functor "Model" and the last two steps define the functor "Reinterpretation". "Reinterpretation" on objects and morphisms is called `ReinterpretationOfObject` and `ReinterpretationOfMorphism` in the code. "Model" on objects and morphisms is called `ModelingObject` and `ModelingMorphism` in the code.

12.4 Relation to `CompilerForCAP`

The operation of the tower `cat_n` (step 3 above) usually unboxes objects and morphisms, operates on the underlying data, and boxes the result. The unboxing usually cancels with step 2 above, and boxing the result usually cancels with step 4 above. If one now compiles the operations of the reinterpretation `R`, only the following steps remain:

1. Apply `object_datum` and `morphism_datum` to the input to get the underlying data.
2. Operate on the underlying data. (previously part of step 3)
3. Apply `object_constructor` or `morphism_constructor` to the underlying data to get an object or a morphism in the reinterpretation `R`. (previously step 5)

This is exactly what a primitive implementation would look like. Thus, in many cases compiling a reinterpretation immediately gives a primitive implementation with no remaining references to the tower `cat_n`.

12.5 Attributes

12.5.1 ModelingCategory (for IsCapCategory)

▷ `ModelingCategory(R)` (attribute)

Returns: a category

The tower `cat_n` modeling the reinterpretation *R*.

12.6 Constructors

12.6.1 ReinterpretationOfCategory (for IsCapCategory, IsRecord)

▷ `ReinterpretationOfCategory(category, options)` (operation)

Returns: a category

Reinterprets a category *category* (the "modeling category") to form a new category `R` (the "reinterpretation") subject to the options given via *options*, which is a record with the following keys:

- `name`: the name of the reinterpretation `R`,
- `category_filter`, `category_object_filter`, `category_morphism_filter`, `object_datum_type`, `morphism_datum_type`, `object_constructor`, `object_datum`, `morphism_constructor`, `morphism_datum`: same meaning as for `CategoryConstructor` (11.2.1), which is used to create the reinterpretation `R`,
- `modeling_tower_object_constructor`: a function which gets the reinterpretation `R` and an object datum (in the sense of `object_datum`) and returns the corresponding modeling object in the modeling category,
- `modeling_tower_object_datum`: a function which gets the reinterpretation `R` and an object in the modeling category and returns the corresponding object datum (in the sense of `object_datum`),
- `modeling_tower_morphism_constructor`: a function which gets the reinterpretation `R`, a source in the modeling category, a morphism datum (in the sense of `morphism_datum`), and a range in the modeling category and returns the corresponding modeling morphism in the modeling category,
- `modeling_tower_morphism_datum`: a function which gets the reinterpretation `R` and a morphism in the modeling category and returns the corresponding morphism datum (in the sense of `morphism_datum`),
- `only_primitive_operations` (optional, default `false`): whether to only reinterpret primitive operations or all operations.

12.6.2 ReinterpretationFunctor (for IsCapCategory)

▷ `ReinterpretationFunctor(R)` (attribute)

Returns: a functor

Returns the functor from the modeling category `ModelingCategory(R)` to the reinterpretation *R* which maps each object/morphism to its reinterpretation.

12.6.3 ModelingObject (for IsCapCategory, IsCapCategoryObject)

▷ `ModelingObject(R, obj)` (operation)

Returns: a CAP category object

Returns the object in `ModelingCategory(R)` modeling the object *obj* in the reinterpretation *R*.

12.6.4 ReinterpretationOfObject (for IsCapCategory, IsCapCategoryObject)

▷ `ReinterpretationOfObject(R, obj)` (operation)

Returns: a CAP category object

Returns the reinterpretation in `R` of an object *obj* in `ModelingCategory(R)`.

12.6.5 ModelingMorphism (for IsCapCategory, IsCapCategoryMorphism)

▷ `ModelingMorphism(R, mor)` (operation)

Returns: a CAP category morphism

Returns the morphism in `ModelingCategory(R)` modeling the morphism *mor* in the reinterpretation *R*.

12.6.6 ReinterpretationOfMorphism (for IsCapCategory, IsCapCategoryObject, IsCapCategoryMorphism, IsCapCategoryObject)

▷ `ReinterpretationOfMorphism(R, source, mor, range)` (operation)

Returns: a CAP category morphism

Returns the reinterpretation in `R` with given source and range in `R` of a morphism *mor* in `ModelingCategory(R)`.

Chapter 13

Create wrapper hulls of a category

13.1 GAP categories

13.1.1 IsWrapperCapCategory (for IsCapCategory)

▷ `IsWrapperCapCategory(`*arg*`)` (Category)

Returns: `true` or `false`

The GAP category of a wrapper CAP category.

13.1.2 IsWrapperCapCategoryObject (for IsCapCategoryObject)

▷ `IsWrapperCapCategoryObject(`*arg*`)` (Category)

Returns: `true` or `false`

The GAP category of objects in a wrapper CAP category.

13.1.3 IsWrapperCapCategoryMorphism (for IsCapCategoryMorphism)

▷ `IsWrapperCapCategoryMorphism(`*arg*`)` (Category)

Returns: `true` or `false`

The GAP category of morphisms in a wrapper CAP category.

13.2 Attributes

13.2.1 UnderlyingCell (for IsWrapperCapCategoryObject)

▷ `UnderlyingCell(`*object*`)` (attribute)

Returns: a category object

The cell underlying the wrapper category object *object*.

13.2.2 UnderlyingCell (for IsWrapperCapCategoryMorphism)

▷ `UnderlyingCell(`*morphism*`)` (attribute)

Returns: a category morphism

The cell underlying the wrapper category morphism *morphism*.

13.3 Constructors

13.3.1 AsObjectInWrapperCategory (for IsWrapperCapCategory, IsCapCategoryObject)

▷ `AsObjectInWrapperCategory(`*`category, object`*`)` (operation)

Returns: an object

Wrap an object *`object`* (in the category underlying the wrapper category *`category`*) to form an object in *`category`*.

13.3.2 AsMorphismInWrapperCategory (for IsWrapperCapCategoryObject, IsCapCategoryMorphism, IsWrapperCapCategoryObject)

▷ `AsMorphismInWrapperCategory(`*`source, morphism, range`*`)` (operation)

Returns: a morphism

Wrap a morphism *`morphism`* (in the category underlying the wrapper category `CapCategory(`*`source`*`)`) to form a morphism in `CapCategory(`*`source`*`)` with given source and range.

13.3.3 AsMorphismInWrapperCategory (for IsWrapperCapCategory, IsCapCategoryMorphism)

▷ `AsMorphismInWrapperCategory(`*`category, morphism`*`)` (operation)

Returns: a morphism

Wrap a morphism *`morphism`* (in the category underlying the wrapper category *`category`*) to form a morphism in *`category`*.

13.3.4 / (for IsCapCategoryCell, IsWrapperCapCategory)

▷ `/(`*`cell, category`*`)` (operation)

Convenience method for `AsObjectInWrapperCategory` (13.3.1) and `AsMorphismInWrapperCategory` (13.3.3).

13.3.5 WrapperCategory (for IsCapCategory, IsRecord)

▷ `WrapperCategory(`*`category, options`*`)` (operation)

Returns: a category

Wraps a category *`category`* to form a new category subject to the options given via *`options`*, which is a record with the following keys:

- `name` (optional): the name of the wrapper category
- `only_primitive_operations` (optional, default `false`): whether to only wrap primitive operations or all operations

Additionally, the following options of `CategoryConstructor` (11.2.1) are supported: `category_filter`, `category_object_filter`, `category_morphism_filter`. The filters must imply `IsWrapperCapCategory`, `IsWrapperCapCategoryObject`, and `IsWrapperCapCategoryMorphism`, respectively.

13.3.6 WrappingFunctor (for IsWrapperCapCategory)

▷ `WrappingFunctor(W)` (attribute)

Returns: a functor

Return the functor from the wrapped category `ModelingCategory(W)` to the wrapper category `W` which simply wraps objects/morphisms.

Chapter 14

Dummy implementations

A dummy implementation of a concept seems to provide an interface for the concept, but calling any operation in this interface will simply signal an error. Hence, when using a dummy implementation, we can be sure that we only rely on the abstract interface but not on any implementation details, for the simple reason that there is no actual implementation. This is useful for testing or compilation against a generic implementation of a concept.

14.1 Dummy rings

14.1.1 IsDummySemiring

▷ `IsDummySemiring(arg)` (filter)

Returns: `true` or `false`

The GAP filter of dummy semirings.

14.1.2 IsDummySemiringElement

▷ `IsDummySemiringElement(arg)` (filter)

Returns: `true` or `false`

The GAP filter of elements of a dummy semiring.

14.1.3 IsDummyCommutativeSemiring

▷ `IsDummyCommutativeSemiring(arg)` (filter)

Returns: `true` or `false`

The GAP filter of dummy commutative semirings.

14.1.4 IsDummyCommutativeSemiringElement

▷ `IsDummyCommutativeSemiringElement(arg)` (filter)

Returns: `true` or `false`

The GAP filter of elements of a dummy commutative semiring.

14.1.5 IsDummyRing

▷ `IsDummyRing(arg)` (filter)

Returns: `true` or `false`

The GAP filter of dummy rings.

14.1.6 IsDummyRingElement

▷ `IsDummyRingElement(arg)` (filter)

Returns: `true` or `false`

The GAP filter of elements of a dummy ring.

14.1.7 IsDummyCommutativeRing

▷ `IsDummyCommutativeRing(arg)` (filter)

Returns: `true` or `false`

The GAP filter of dummy commutative rings.

14.1.8 IsDummyCommutativeRingElement

▷ `IsDummyCommutativeRingElement(arg)` (filter)

Returns: `true` or `false`

The GAP filter of elements of a dummy commutative ring.

14.1.9 IsDummyField

▷ `IsDummyField(arg)` (filter)

Returns: `true` or `false`

The GAP filter of dummy fields.

14.1.10 IsDummyFieldElement

▷ `IsDummyFieldElement(arg)` (filter)

Returns: `true` or `false`

The GAP filter of elements of a dummy field.

14.1.11 DummySemiring

▷ `DummySemiring()` (function)

Returns: a dummy semiring

14.1.12 DummyCommutativeSemiring

▷ `DummyCommutativeSemiring()` (function)

Returns: a dummy commutative semiring

14.1.13 DummyRing

▷ `DummyRing()` (function)

Returns: a dummy ring

14.1.14 DummyCommutativeRing

▷ `DummyCommutativeRing()` (function)

Returns: a dummy commutative ring

14.1.15 DummyField

▷ `DummyField()` (function)

Returns: a dummy field

14.2 Dummy categories

14.2.1 IsDummyCategory (for IsCapCategory)

▷ `IsDummyCategory(`*arg*`)` (Category)

Returns: `true` or `false`

The GAP category of a dummy CAP category.

14.2.2 IsDummyCategoryObject (for IsCapCategoryObject)

▷ `IsDummyCategoryObject(`*arg*`)` (Category)

Returns: `true` or `false`

The GAP category of objects in a dummy CAP category.

14.2.3 IsDummyCategoryMorphism (for IsCapCategoryMorphism)

▷ `IsDummyCategoryMorphism(`*arg*`)` (Category)

Returns: `true` or `false`

The GAP category of morphisms in a dummy CAP category.

14.2.4 DummyCategory (for IsRecord)

▷ `DummyCategory(`*options*`)` (operation)

Returns: a category

Creates a dummy category subject to the options given via *options*, which is a record passed on to `CategoryConstructor` (11.2.1). Note that the options `{category,object,morphism}_filter` will be set to `IsDummyCategory{,Object,Morphism}` and the options `{object,morphism}_{constructor,datum}` and `create_func_*` will be set to dummy implementations (throwing errors when actually called). The dummy category will pretend to support empty limits by default.

14.3 Dummy homalg rings

The operations in this section require `MatricesForHomalg` to be loaded.

14.3.1 IsDummyHomalgRing

▷ `IsDummyHomalgRing(arg)` (filter)

Returns: `true` or `false`

The GAP filter of dummy homalg rings.

14.3.2 IsDummyHomalgRingElement

▷ `IsDummyHomalgRingElement(arg)` (filter)

Returns: `true` or `false`

The GAP filter of elements of a dummy homalg ring.

14.3.3 IsDummyCommutativeHomalgRing

▷ `IsDummyCommutativeHomalgRing(arg)` (filter)

Returns: `true` or `false`

The GAP filter of dummy commutative homalg rings.

14.3.4 IsDummyCommutativeHomalgRingElement

▷ `IsDummyCommutativeHomalgRingElement(arg)` (filter)

Returns: `true` or `false`

The GAP filter of elements of a dummy commutative homalg ring.

14.3.5 IsDummyHomalgField

▷ `IsDummyHomalgField(arg)` (filter)

Returns: `true` or `false`

The GAP filter of dummy homalg fields.

14.3.6 IsDummyHomalgFieldElement

▷ `IsDummyHomalgFieldElement(arg)` (filter)

Returns: `true` or `false`

The GAP filter of elements of a dummy homalg field.

14.3.7 DummyHomalgRing

▷ `DummyHomalgRing()` (function)

Returns: a dummy homalg ring

14.3.8 DummyCommutativeHomalgRing

▷ `DummyCommutativeHomalgRing()` (function)

Returns: a dummy commutative homalg ring

14.3.9 DummyHomalgField

▷ `DummyHomalgField()` (function)

Returns: a dummy homalg field

Chapter 15

Terminal category

15.1 GAP Categories

15.1.1 IsCapTerminalCategoryWithSingleObject (for IsCapCategory)

▷ `IsCapTerminalCategoryWithSingleObject(`*T*`)` (Category)

Returns: `true` or `false`

The GAP type of a terminal category with a single object.

15.1.2 IsObjectInCapTerminalCategoryWithSingleObject (for IsCapCategoryObject)

▷ `IsObjectInCapTerminalCategoryWithSingleObject(`*T*`)` (Category)

Returns: `true` or `false`

The GAP type of an object in a terminal category with a single object.

15.1.3 IsMorphismInCapTerminalCategoryWithSingleObject (for IsCapCategoryMorphism)

▷ `IsMorphismInCapTerminalCategoryWithSingleObject(`*T*`)` (Category)

Returns: `true` or `false`

The GAP type of a morphism in a terminal category with a single object.

15.1.4 IsCapTerminalCategoryWithMultipleObjects (for IsCapCategory)

▷ `IsCapTerminalCategoryWithMultipleObjects(`*T*`)` (Category)

Returns: `true` or `false`

The GAP type of a terminal category with multiple objects.

15.1.5 IsObjectInCapTerminalCategoryWithMultipleObjects (for IsCapCategoryObject)

▷ `IsObjectInCapTerminalCategoryWithMultipleObjects(`*T*`)` (Category)

Returns: `true` or `false`

The GAP type of an object in a terminal category with multiple objects.

15.1.6 IsMorphismInCapTerminalCategoryWithMultipleObjects (for IsCapCategoryMorphism)

▷ `IsMorphismInCapTerminalCategoryWithMultipleObjects(T)` (Category)

Returns: `true` or `false`

The GAP type of a morphism in a terminal category with multiple objects.

15.1.7 IsTerminalCategory (for IsCapCategory)

▷ `IsTerminalCategory(C)` (property)

Returns: `true` or `false`

The property of the category *C* being terminal.

15.2 Constructors

15.2.1 TerminalCategoryWithSingleObject

▷ `TerminalCategoryWithSingleObject(arg)` (function)

Construct a terminal category with a single object.

15.2.2 TerminalCategoryWithMultipleObjects

▷ `TerminalCategoryWithMultipleObjects(arg)` (function)

Construct a terminal category with multiple objects.

15.2.3 CAP_INTERNAL_CONSTRUCTOR_FOR_TERMINAL_CATEGORY

▷ `CAP_INTERNAL_CONSTRUCTOR_FOR_TERMINAL_CATEGORY(options)` (function)

Returns: a CAP category

This function takes a record of options suited for CategoryConstructor. It makes common adjustments for TerminalCategoryWithSingleObject and TerminalCategoryWithMultipleObjects to the list of operations to install and the categorical properties of the given record, before passing it on to CategoryConstructor.

15.3 Attributes

15.3.1 TerminalCategoryWithSingleObjectUniqueObject (for IsCapTerminalCategoryWithSingleObject)

▷ `TerminalCategoryWithSingleObjectUniqueObject(arg)` (attribute)

Returns: a CAP object

The unique object in a terminal category with a single object.

15.3.2 TerminalCategoryWithSingleObjectUniqueMorphism (for IsCapTerminalCategoryWithSingleObject)

▷ `TerminalCategoryWithSingleObjectUniqueMorphism(`*arg*`)` (attribute)

Returns: a CAP morphism

The unique morphism in a terminal category with a single object.

15.3.3 FunctorFromTerminalCategory (for IsCapCategoryObject)

▷ `FunctorFromTerminalCategory(`*object*`)` (attribute)

Returns: a CAP functor

A functor from `AsCapCategory( TerminalObject( CapCat ) )` mapping the unique object to *object*.

Chapter 16

Finite skeletal discrete categories

16.1 GAP Categories

16.1.1 IsFiniteSkeletalDiscreteCategory (for IsCapCategory)

▷ `IsFiniteSkeletalDiscreteCategory(C)` (Category)

Returns: `true` or `false`

The GAP type of a finite skeletal discrete category.

16.1.2 IsObjectInFiniteSkeletalDiscreteCategory (for IsCapCategoryObject)

▷ `IsObjectInFiniteSkeletalDiscreteCategory(O)` (Category)

Returns: `true` or `false`

The GAP type of an object in a finite skeletal discrete category.

16.1.3 IsMorphismInFiniteSkeletalDiscreteCategory (for IsCapCategoryMorphism)

▷ `IsMorphismInFiniteSkeletalDiscreteCategory(M)` (Category)

Returns: `true` or `false`

The GAP type of a morphism in a finite skeletal discrete category.

16.2 Constructors

16.2.1 FiniteSkeletalDiscreteCategory

▷ `FiniteSkeletalDiscreteCategory(arg)` (function)

Returns: a CAP category

Construct a finite skeletal discrete category.

16.3 Attributes

16.3.1 UnderlyingListOfGapObjects (for IsFiniteSkeletalDiscreteCategory)

▷ `UnderlyingListOfGapObjects(arg)` (attribute)

Returns: a list of CAP objects

The underlying GAP objects of a finite skeletal discrete category.

16.3.2 UnderlyingGapObject (for IsObjectInFiniteSkeletalDiscreteCategory)

▷ `UnderlyingGapObject(`*arg*`)` (attribute)

Returns: a CAP object

The underlying GAP object of an object in a finite skeletal discrete category.

16.4 Properties

16.4.1 IsDiscreteCategory (for IsCapCategory)

▷ `IsDiscreteCategory(`*C*`)` (property)

Returns: `true` or `false`

The property of *C* being a discrete CAP category, i.e., equivalent to a category in which any morphism is an identity.

Chapter 17

Examples and Tests

17.1 Dummy implementations

17.1.1 Dummy categories

Example

```
gap> LoadPackage( "CAP", false );
true
gap> list_of_operations_to_install := [
>     "ObjectConstructor",
>     "MorphismConstructor",
>     "ObjectDatum",
>     "MorphismDatum",
>     "IsCongruentForMorphisms",
>     "PreCompose",
>     "IdentityMorphism",
>     "DirectSum",
> ];;
gap> dummy := DummyCategory( rec(
>     list_of_operations_to_install := list_of_operations_to_install,
>     properties := [ "IsAdditiveCategory" ],
> ) );;
gap> ForAll( list_of_operations_to_install, o -> CanCompute( dummy, o ) );
true
gap> IsAdditiveCategory( dummy );
true
```

17.1.2 Dummy rings

Example

```
gap> LoadPackage( "CAP", false );
true
gap> DummyRing( );
Dummy ring 1
gap> DummyRing( );
Dummy ring 2
gap> IsRing( DummyRing( ) );
true
gap> DummyCommutativeRing( );
```

```
Dummy commutative ring 1
gap> DummyCommutativeRing( );
Dummy commutative ring 2
gap> IsRing( DummyCommutativeRing( ) );
true
gap> IsCommutative( DummyCommutativeRing( ) );
true
gap> DummyField( );
Dummy field 1
gap> DummyField( );
Dummy field 2
gap> IsSemiring( DummyField( ) );
true
gap> IsRing( DummyField( ) );
true
gap> IsField( DummyField( ) );
true
gap> IsCommutative( DummyField( ) );
true
```

17.1.3 Dummy semirings

Example

```
gap> LoadPackage( "CAP", false );
true
gap> DummyCommutativeSemiring( );
Dummy commutative semiring 1
gap> DummyCommutativeSemiring( );
Dummy commutative semiring 2
gap> IsSemiring( DummyCommutativeSemiring( ) );
true
gap> IsCommutative( DummyCommutativeSemiring( ) );
true
```

17.2 Finite skeletal discrete categories

Example

```
gap> LoadPackage( "CAP", false );
true
gap> D := FiniteSkeletalDiscreteCategory( [ 1 .. 5 ] );
FiniteSkeletalDiscreteCategory( [ 1 .. 5 ] )
gap> one := ObjectConstructor( D, 1 );
<An object in FiniteSkeletalDiscreteCategory( [ 1 .. 5 ] )>
gap> IsWellDefinedForObjects( one );
true
gap> ObjectDatum( one ) = 1;
true
gap> Display( one );
1
gap> IsEqualForObjects( one, D[1] );
true
gap> id_one := IdentityMorphism( D, one );
```

```
<An identity morphism in FiniteSkeletalDiscreteCategory( [ 1 .. 5 ] )>
gap> IsWellDefinedForMorphisms( id_one );
true
gap> MorphismDatum( id_one ) = fail;
true
gap> Display( id_one );
1
|
| A morphism in FiniteSkeletalDiscreteCategory( [ 1 .. 5 ] )
v
1
gap> IsEqualForMorphisms( PreCompose( id_one, id_one ), id_one );
true
gap> Length( SetOfObjectsOfCategory( D ) );
5
gap> Length( SetOfMorphismsOfFiniteCategory( D ) );
5
```

17.3 Functors

We create a binary functor F with one covariant and one contravariant component in two ways. Here is the first way to model a binary functor:

Example
```
gap> ring := HomalgRingOfIntegers( );;
gap> vec := LeftPresentations( ring );;
gap> F := CapFunctor( "CohomForVec", [ vec, [ vec, true ] ], vec );;
gap> obj_func := function( A, B ) return TensorProductOnObjects( A, DualOnObjects( B ) ); end;;
gap> mor_func := function( source, alpha, beta, range ) return TensorProductOnMorphismsWithGivenT
gap> AddObjectFunction( F, obj_func );;
gap> AddMorphismFunction( F, mor_func );;
```

CAP regards F as a binary functor on a technical level, as we can see by looking at its input signature:

Example
```
gap> InputSignature( F );
[ [ Category of left presentations of Z, false ], [ Category of left presentations of Z, true ] ]
```

We can see that `ApplyFunctor` works both on two arguments and on one argument (in the product category).

Example
```
gap> V1 := TensorUnit( vec );;
gap> V3 := DirectSum( V1, V1, V1 );;
gap> pi1 := ProjectionInFactorOfDirectSum( [ V1, V1 ], 1 );;
gap> pi2 := ProjectionInFactorOfDirectSum( [ V3, V1 ], 1 );;
gap> value1 := ApplyFunctor( F, pi1, pi2 );;
gap> input := Product( pi1, Opposite( pi2 ) );;
gap> value2 := ApplyFunctor( F, input );;
gap> IsCongruentForMorphisms( value1, value2 );
true
```

Here is the second way to model a binary functor:

Example
```
gap> F2 := CapFunctor( "CohomForVec2", Product( vec, Opposite( vec ) ), vec );;
gap> AddObjectFunction( F2, a -> obj_func( a[1], Opposite( a[2] ) ) );;
gap> AddMorphismFunction( F2, function( source, datum, range ) return mor_func( source, datum[1],
gap> value3 := ApplyFunctor( F2,input );;
gap> IsCongruentForMorphisms( value1, value3 );
true
```

CAP regards $F2$ as a unary functor on a technical level, as we can see by looking at its input signature:

Example
```
gap> InputSignature( F2 );
[ [ Product of: Category of left presentations of Z, Opposite( Category of left presentations of
```

Installation of the first functor as a GAP-operation. It will be installed both as a unary and binary version.

Example
```
gap> InstallFunctor( F, "F_installation" );;
gap> F_installation( pi1, pi2 );;
gap> F_installation( input );;
gap> F_installationOnObjects( V1, V1 );;
gap> F_installationOnObjects( Product( V1, Opposite( V1 ) ) );;
gap> F_installationOnMorphisms( pi1, pi2 );;
gap> F_installationOnMorphisms( input );;
```

Installation of the second functor as a GAP-operation. It will be installed only as a unary version.

Example
```
gap> InstallFunctor( F2, "F_installation2" );;
gap> F_installation2( input );;
gap> F_installation2OnObjects( Product( V1, Opposite( V1 ) ) );;
gap> F_installation2OnMorphisms( input );;
```

17.4 HandlePrecompiledTowers

Example
```
gap> LoadPackage( "CAP", false );
true
gap> dummy1 := CreateCapCategory( );;
gap> dummy2 := CreateCapCategory( );;
gap> dummy3 := CreateCapCategory( );;
gap> PrintAndReturn := function ( string )
>     Print( string, "\n" ); return string; end;;
gap> dummy1!.compiler_hints := rec( );;
gap> dummy1!.compiler_hints.precompiled_towers := [
>   rec(
>     remaining_constructors_in_tower := [ "Constructor1" ],
>     precompiled_functions_adder := cat ->
>       PrintAndReturn( "Adding precompiled operations for Constructor1" ),
>   ),
>   rec(
>     remaining_constructors_in_tower := [ "Constructor1", "Constructor2" ],
>     precompiled_functions_adder := cat ->
```

```
>       PrintAndReturn( "Adding precompiled operations for Constructor2" ),
>   ),
> ];;
gap> HandlePrecompiledTowers( dummy2, dummy1, "Constructor1" );
Adding precompiled operations for Constructor1
gap> HandlePrecompiledTowers( dummy3, dummy2, "Constructor2" );
Adding precompiled operations for Constructor2
```

17.5 Terminal category

Example

```
gap> LoadPackage( "CAP", false );
true
gap> T := TerminalCategoryWithMultipleObjects( );
TerminalCategoryWithMultipleObjects( )
gap> i := InitialObject( T );
<A zero object in TerminalCategoryWithMultipleObjects( )>
gap> t := TerminalObject( T );
<A zero object in TerminalCategoryWithMultipleObjects( )>
gap> z := ZeroObject( T );
<A zero object in TerminalCategoryWithMultipleObjects( )>
gap> Display( i );
InitialObject(  )
gap> Display( t );
TerminalObject(  )
gap> Display( z );
ZeroObject(  )
gap> IsIdenticalObj( i, z );
false
gap> IsIdenticalObj( t, z );
false
gap> id_z := IdentityMorphism( z );
<A zero, identity morphism in TerminalCategoryWithMultipleObjects( )>
gap> fn_z := ZeroObjectFunctorial( T );
<A zero, isomorphism in TerminalCategoryWithMultipleObjects( )>
gap> IsEqualForMorphisms( id_z, fn_z );
false
gap> IsCongruentForMorphisms( id_z, fn_z );
true
gap> a := "a" / T;
<A zero object in TerminalCategoryWithMultipleObjects( )>
gap> Display( a );
a
gap> IsWellDefined( a );
true
gap> aa := ObjectConstructor( T, "a" );
<A zero object in TerminalCategoryWithMultipleObjects( )>
gap> Display( aa );
a
gap> IsEqualForObjects( a, aa );
true
gap> IsIsomorphicForObjects( a, aa );
```

```
true
gap> IsIsomorphism( SomeIsomorphismBetweenObjects( a, aa ) );
true
gap> b := "b" / T;
<A zero object in TerminalCategoryWithMultipleObjects( )>
gap> Display( b );
b
gap> IsEqualForObjects( a, b );
false
gap> IsIsomorphicForObjects( a, b );
true
gap> mor_ab := SomeIsomorphismBetweenObjects( a, b );
<A zero, isomorphism in TerminalCategoryWithMultipleObjects( )>
gap> IsIsomorphism( mor_ab );
true
gap> Display( mor_ab );
a
|
| SomeIsomorphismBetweenObjects
v
b
gap> Hom_ab := MorphismsOfExternalHom( a, b );;
gap> Length( Hom_ab );
1
gap> Hom_ab[1];
<A zero, isomorphism in TerminalCategoryWithMultipleObjects( )>
gap> Display( Hom_ab[1] );
a
|
| InterpretMorphismFromDistinguishedObjectToHomomorphismStructureAsMorphism
v
b
gap> Hom_ab[1] = mor_ab;
true
gap> HomStructure( mor_ab );
<A zero, identity morphism in TerminalCategoryWithSingleObject( )>
gap> m := MorphismConstructor( a, "m", b );
<A zero, isomorphism in TerminalCategoryWithMultipleObjects( )>
gap> Display( m );
a
|
| m
v
b
gap> IsWellDefined( m );
true
gap> n := MorphismConstructor( a, "n", b );
<A zero, isomorphism in TerminalCategoryWithMultipleObjects( )>
gap> Display( n );
a
|
| n
```

```
v
b
gap> IsEqualForMorphisms( m, n );
false
gap> IsCongruentForMorphisms( m, n );
true
gap> m = n;
true
gap> hom_mn := HomStructure( m, n );
<A zero, identity morphism in TerminalCategoryWithSingleObject( )>
gap> id := IdentityMorphism( a );
<A zero, identity morphism in TerminalCategoryWithMultipleObjects( )>
gap> Display( id );
a
|
| IdentityMorphism( a )
v
a
gap> m = id;
false
gap> id = MorphismConstructor( a, "xyz", a );
true
gap> zero := ZeroMorphism( a, a );
<A zero, isomorphism in TerminalCategoryWithMultipleObjects( )>
gap> Display( zero );
a
|
| ZeroMorphism( a, a )
v
a
gap> id = zero;
true
gap> IsLiftable( m, n );
true
gap> lift := Lift( m, n );
<A zero, isomorphism in TerminalCategoryWithMultipleObjects( )>
gap> Display( lift );
a
|
| Lift( m, n )
v
a
gap> IsColiftable( m, n );
true
gap> colift := Colift( m, n );
<A zero, isomorphism in TerminalCategoryWithMultipleObjects( )>
gap> Display( colift );
b
|
| Colift( m, n )
v
b
```

```
gap> DirectProduct( T, [ ] );
<A zero object in TerminalCategoryWithMultipleObjects( )>
gap> Equalizer( T, z, [ ] );
<A zero object in TerminalCategoryWithMultipleObjects( )>
gap> Coproduct( T, [ ] );
<A zero object in TerminalCategoryWithMultipleObjects( )>
gap> Coequalizer( T, z, [ ] );
<A zero object in TerminalCategoryWithMultipleObjects( )>
```

Example

```
gap> LoadPackage( "CAP", false );
true
gap> T := TerminalCategoryWithSingleObject( );
TerminalCategoryWithSingleObject( )
gap> i := InitialObject( T );
<A zero object in TerminalCategoryWithSingleObject( )>
gap> t := TerminalObject( T );
<A zero object in TerminalCategoryWithSingleObject( )>
gap> z := ZeroObject( T );
<A zero object in TerminalCategoryWithSingleObject( )>
gap> Display( i );
A zero object in TerminalCategoryWithSingleObject( ).
gap> Display( t );
A zero object in TerminalCategoryWithSingleObject( ).
gap> Display( z );
A zero object in TerminalCategoryWithSingleObject( ).
gap> IsIdenticalObj( i, z );
false
gap> IsIdenticalObj( t, z );
false
gap> IsWellDefined( z );
true
gap> Length( SetOfObjects( T ) );
1
gap> id_z := IdentityMorphism( z );
<A zero, identity morphism in TerminalCategoryWithSingleObject( )>
gap> fn_z := ZeroObjectFunctorial( T );
<A zero, identity morphism in TerminalCategoryWithSingleObject( )>
gap> IsWellDefined( fn_z );
true
gap> IsEqualForMorphisms( id_z, fn_z );
true
gap> IsCongruentForMorphisms( id_z, fn_z );
true
gap> Length( SetOfMorphisms( T ) );
1
gap> IsLiftable( id_z, fn_z );
true
gap> Lift( id_z, fn_z );
<A zero, identity morphism in TerminalCategoryWithSingleObject( )>
gap> IsColiftable( id_z, fn_z );
true
gap> Colift( id_z, fn_z );
```

```
<A zero, identity morphism in TerminalCategoryWithSingleObject( )>
gap> DirectProduct( T, [ ] );
<A zero object in TerminalCategoryWithSingleObject( )>
gap> Equalizer( T, z, [ ] );
<A zero object in TerminalCategoryWithSingleObject( )>
gap> Coproduct( T, [ ] );
<A zero object in TerminalCategoryWithSingleObject( )>
gap> Coequalizer( T, z, [ ] );
<A zero object in TerminalCategoryWithSingleObject( )>
```

Index