`CategoriesForHomalg` - category theory based programming

Sebastian Gutsche and Sebastian Posur

TU Kaiserslautern / RWTH Aachen

Aachen, August 27, 2014

Motivation

Given two categories $\mathcal{A}$ and $\mathcal{B}$, one can construct lots of new categories

Motivation

Given two categories $\mathcal{A}$ and $\mathcal{B}$, one can construct lots of new categories, e.g., $\mathrm{ChainComplexes}(\mathcal{A})$, $\mathcal{A}/\mathcal{B}$, or $\mathrm{Hom}(\mathcal{A}, \mathcal{B})$.

Motivation

Given two categories $\mathcal{A}$ and $\mathcal{B}$, one can construct lots of new categories, e.g., $\mathrm{ChainComplexes}(\mathcal{A})$, $\mathcal{A}/\mathcal{B}$, or $\mathrm{Hom}(\mathcal{A},\mathcal{B})$. What is the advantage of having such constructions in `GAP`?

Motivation

Given two categories $\mathcal{A}$ and $\mathcal{B}$, one can construct lots of new categories, e.g., $\mathrm{ChainComplexes}(\mathcal{A})$, $\mathcal{A}/\mathcal{B}$, or $\mathrm{Hom}(\mathcal{A}, \mathcal{B})$.
What is the advantage of having such constructions in `GAP`?

- $R - \mathrm{mod} \rightsquigarrow \mathrm{ChainComplexes}(R - \mathrm{mod})$

Motivation

Given two categories $\mathcal{A}$ and $\mathcal{B}$, one can construct lots of new categories, e.g., $\mathrm{ChainComplexes}(\mathcal{A})$, $\mathcal{A}/\mathcal{B}$, or $\mathrm{Hom}(\mathcal{A}, \mathcal{B})$.
What is the advantage of having such constructions in `GAP`?

- $R - \mathrm{mod} \rightsquigarrow \mathrm{ChainComplexes}(R - \mathrm{mod})$
  (extension of functors, e.g., $- \otimes_R M$)

Motivation

Given two categories $\mathcal{A}$ and $\mathcal{B}$, one can construct lots of new categories, e.g., $\mathrm{ChainComplexes}(\mathcal{A})$, $\mathcal{A}/\mathcal{B}$, or $\mathrm{Hom}(\mathcal{A}, \mathcal{B})$.
What is the advantage of having such constructions in `GAP`?

- $R - \mathrm{mod} \rightsquigarrow \mathrm{ChainComplexes}(R - \mathrm{mod})$
  (extension of functors, e.g., $- \otimes_R M$)
- $k[x_0, \ldots, x_n] - \mathrm{grmod} \longrightarrow \frac{k[x_0,\ldots,x_n]-\mathrm{grmod}}{\langle \text{finite dimensional modules} \rangle}$

Motivation

Given two categories $\mathcal{A}$ and $\mathcal{B}$, one can construct lots of new categories, e.g., $\mathrm{ChainComplexes}(\mathcal{A})$, $\mathcal{A}/\mathcal{B}$, or $\mathrm{Hom}(\mathcal{A},\mathcal{B})$.
What is the advantage of having such constructions in `GAP`?

- $R-\mathrm{mod} \rightsquigarrow \mathrm{ChainComplexes}(R-\mathrm{mod})$
  (extension of functors, e.g., $-\otimes_R M$)
- $k[x_0,\ldots,x_n]-\mathrm{grmod} \longrightarrow \frac{k[x_0,\ldots,x_n]-\mathrm{grmod}}{\langle\text{finite dimensional modules}\rangle} \cong \mathfrak{Coh}(\mathbb{P}^n_k)$

Motivation

Given two categories $\mathcal{A}$ and $\mathcal{B}$, one can construct lots of new categories, e.g., $\mathrm{ChainComplexes}(\mathcal{A})$, $\mathcal{A}/\mathcal{B}$, or $\mathrm{Hom}(\mathcal{A}, \mathcal{B})$.
What is the advantage of having such constructions in GAP?

- $R - \mathrm{mod} \rightsquigarrow \mathrm{ChainComplexes}(R - \mathrm{mod})$
  (extension of functors, e.g., $- \otimes_R M$)
- $k[x_0, \ldots, x_n] - \mathrm{grmod} \longrightarrow \dfrac{k[x_0,\ldots,x_n]-\mathrm{grmod}}{\langle \text{finite dimensional modules} \rangle} \cong \mathfrak{Coh}(\mathbb{P}^n_k)$
  (sheafification functor)

Motivation

Given two categories $\mathcal{A}$ and $\mathcal{B}$, one can construct lots of new categories, e.g., $\mathrm{ChainComplexes}(\mathcal{A})$, $\mathcal{A}/\mathcal{B}$, or $\mathrm{Hom}(\mathcal{A}, \mathcal{B})$.
What is the advantage of having such constructions in `GAP`?

- $R - \mathrm{mod} \rightsquigarrow \mathrm{ChainComplexes}(R - \mathrm{mod})$
  (extension of functors, e.g., $- \otimes_R M$)
- $k[x_0, \ldots, x_n] - \mathrm{grmod} \longrightarrow \frac{k[x_0,\ldots,x_n]-\mathrm{grmod}}{\langle \text{finite dimensional modules} \rangle} \cong \mathfrak{Coh}(\mathbb{P}^n_k)$
  (sheafification functor)

We can derive useful new data structures and methods out of old ones

Motivation

Given two categories $\mathcal{A}$ and $\mathcal{B}$, one can construct lots of new categories, e.g., $\mathrm{ChainComplexes}(\mathcal{A})$, $\mathcal{A}/\mathcal{B}$, or $\mathrm{Hom}(\mathcal{A},\mathcal{B})$. What is the advantage of having such constructions in `GAP`?

- $R-\mathrm{mod} \rightsquigarrow \mathrm{ChainComplexes}(R-\mathrm{mod})$
  (extension of functors, e.g., $-\otimes_R M$)
- $k[x_0,\ldots,x_n]-\mathrm{grmod} \longrightarrow \frac{k[x_0,\ldots,x_n]-\mathrm{grmod}}{\langle \text{finite dimensional modules} \rangle} \cong \mathfrak{Coh}(\mathbb{P}^n_k)$
  (sheafification functor)

We can derive useful new data structures and methods out of old ones in a compatible way.

Motivation

Given two categories $\mathcal{A}$ and $\mathcal{B}$, one can construct lots of new categories, e.g., $\mathrm{ChainComplexes}(\mathcal{A})$, $\mathcal{A}/\mathcal{B}$, or $\mathrm{Hom}(\mathcal{A},\mathcal{B})$.
What is the advantage of having such constructions in `GAP`?

- $R - \mathrm{mod} \rightsquigarrow \mathrm{ChainComplexes}(R - \mathrm{mod})$
  (extension of functors, e.g., $- \otimes_R M$)
- $k[x_0, \dots, x_n] - \mathrm{grmod} \longrightarrow \frac{k[x_0,\dots,x_n]-\mathrm{grmod}}{\langle \text{finite dimensional modules} \rangle} \cong \mathfrak{Coh}(\mathbb{P}^n_k)$
  (sheafification functor)

We can derive useful new data structures and methods out of old ones in a compatible (functorial) way.

CategoriesForHomalg

CategoriesForHomalg

`CategoriesForHomalg` is a `GAP` package providing

CategoriesForHomalg

`CategoriesForHomalg` is a `GAP` package providing

1. data structures for

CategoriesForHomalg

`CategoriesForHomalg` is a `GAP` package providing

1. data structures for
   - categories,

CategoriesForHomalg

`CategoriesForHomalg` is a `GAP` package providing

1. data structures for
   - categories,
   - functors,

CategoriesForHomalg

`CategoriesForHomalg` is a `GAP` package providing

1. data structures for
   - categories,
   - functors,
   - natural transformations,

CategoriesForHomalg

`CategoriesForHomalg` is a `GAP` package providing

1. data structures for
   - categories,
   - functors,
   - natural transformations,
   - opposite categories,

CategoriesForHomalg

`CategoriesForHomalg` is a `GAP` package providing

1. data structures for
   - categories,
   - functors,
   - natural transformations,
   - opposite categories,
   - chain complexes,

CategoriesForHomalg

`CategoriesForHomalg` is a `GAP` package providing

1. data structures for
   - categories,
   - functors,
   - natural transformations,
   - opposite categories,
   - chain complexes,
   - (Serre) quotient categories,

CategoriesForHomalg

`CategoriesForHomalg` is a `GAP` package providing

1. data structures for
   - categories,
   - functors,
   - natural transformations,
   - opposite categories,
   - chain complexes,
   - (Serre) quotient categories,
   - ...

CategoriesForHomalg

`CategoriesForHomalg` is a `GAP` package providing

1. data structures for
   - categories,
   - functors,
   - natural transformations,
   - opposite categories,
   - chain complexes,
   - (Serre) quotient categories,
   - ...
2. methods for

CategoriesForHomalg

`CategoriesForHomalg` is a `GAP` package providing

1. data structures for
   - categories,
   - functors,
   - natural transformations,
   - opposite categories,
   - chain complexes,
   - (Serre) quotient categories,
   - ...
2. methods for
   - pullbacks,

CategoriesForHomalg

`CategoriesForHomalg` is a `GAP` package providing

1. data structures for
   - categories,
   - functors,
   - natural transformations,
   - opposite categories,
   - chain complexes,
   - (Serre) quotient categories,
   - ...
2. methods for
   - pullbacks,
   - pushouts,

CategoriesForHomalg

`CategoriesForHomalg` is a `GAP` package providing

1. data structures for
   - categories,
   - functors,
   - natural transformations,
   - opposite categories,
   - chain complexes,
   - (Serre) quotient categories,
   - ...
2. methods for
   - pullbacks,
   - pushouts,
   - connecting homomorphisms,

CategoriesForHomalg

`CategoriesForHomalg` is a `GAP` package providing

1. data structures for
   - categories,
   - functors,
   - natural transformations,
   - opposite categories,
   - chain complexes,
   - (Serre) quotient categories,
   - ...
2. methods for
   - pullbacks,
   - pushouts,
   - connecting homomorphisms,
   - generalized morphisms,

CategoriesForHomalg

`CategoriesForHomalg` is a `GAP` package providing

1. data structures for
   - categories,
   - functors,
   - natural transformations,
   - opposite categories,
   - chain complexes,
   - (Serre) quotient categories,
   - ...
2. methods for
   - pullbacks,
   - pushouts,
   - connecting homomorphisms,
   - generalized morphisms,
   - spectral sequences,

CategoriesForHomalg

`CategoriesForHomalg` is a `GAP` package providing

1. data structures for
   - categories,
   - functors,
   - natural transformations,
   - opposite categories,
   - chain complexes,
   - (Serre) quotient categories,
   - ...
2. methods for
   - pullbacks,
   - pushouts,
   - connecting homomorphisms,
   - generalized morphisms,
   - spectral sequences,
   - ...

A simple example: QVectorSpaces

`QVectorSpaces`:
The category of finite dimensional vector spaces over $\mathbb{Q}$.

A simple example: QVectorSpaces

`QVectorSpaces`:
The category of finite dimensional vector spaces over $\mathbb{Q}$.

1. Create a HomalgCategory:

```
QVectorSpaces := CreateHomalgCategory( "QVectorSpaces" );
```

A simple example: QVectorSpaces

`QVectorSpaces`:
The category of finite dimensional vector spaces over $\mathbb{Q}$.

1. Create a HomalgCategory:

```
QVectorSpaces := CreateHomalgCategory( "QVectorSpaces" );
```

2. Write constructors for objects and morphisms:

A simple example: QVectorSpaces

`QVectorSpaces`:
The category of finite dimensional vector spaces over $\mathbb{Q}$.

1. Create a HomalgCategory:

```
QVectorSpaces := CreateHomalgCategory( "QVectorSpaces" );
```

2. Write constructors for objects and morphisms:
   - Data structure for objects: $\mathbb{N}_0$

A simple example: QVectorSpaces

`QVectorSpaces`:
The category of finite dimensional vector spaces over $\mathbb{Q}$.

1. Create a HomalgCategory:

```
QVectorSpaces := CreateHomalgCategory( "QVectorSpaces" );
```

2. Write constructors for objects and morphisms:
   - Data structure for objects: $\mathbb{N}_0$ (wrapped integers)

A simple example: QVectorSpaces

`QVectorSpaces`:
The category of finite dimensional vector spaces over $\mathbb{Q}$.

1. Create a HomalgCategory:

```
QVectorSpaces := CreateHomalgCategory( "QVectorSpaces" );
```

2. Write constructors for objects and morphisms:
   - Data structure for objects: $\mathbb{N}_0$ (wrapped integers)
   - Data structure for morphisms: Matrices with entries in $\mathbb{Q}$

A simple example: QVectorSpaces

`QVectorSpaces`:
The category of finite dimensional vector spaces over $\mathbb{Q}$.

1. Create a HomalgCategory:

```
QVectorSpaces := CreateHomalgCategory( "QVectorSpaces" );
```

2. Write constructors for objects and morphisms:
   - Data structure for objects: $\mathbb{N}_0$ (wrapped integers)
   - Data structure for morphisms: Matrices with entries in $\mathbb{Q}$

```
...
ObjectifyWithAttributes( morphism, TypeOfQVectorSpaceMorphisms,
  Source, source,
  Range, range );

Add( QVectorSpaces, morphism );
...
```

A simple example: QVectorSpaces

3. Add some basic algorithms to `QVectorSpaces`

A simple example: QVectorSpaces

3. Add some basic algorithms to `QVectorSpaces`, for example:

A simple example: QVectorSpaces

3. Add some basic algorithms to `QVectorSpaces`, for example:

```
AddKernel( QVectorSpaces,

  function( morphism )
    local matrix;

    matrix := morphism!.underlying_matrix;

    return
      QVectorSpace( NrRows( matrix ) - RankOfMatrix( matrix ) );

end );
```

A simple example: QVectorSpaces

Actually, there is more to say about the kernel:

A simple example: QVectorSpaces

Actually, there is more to say about the kernel: To handle the kernel of φ algorithmically ...

$$M \xrightarrow{\varphi} N$$

A simple example: QVectorSpaces

Actually, there is more to say about the kernel: To handle the kernel of φ algorithmically ...

... one has to construct the object $\ker\varphi$,

$$\begin{tikzcd} \ker\varphi & & \\ & M \arrow[r, "\varphi"] & N \end{tikzcd}$$

A simple example: QVectorSpaces

Actually, there is more to say about the kernel: To handle the kernel of φ algorithmically ...

... one has to construct the object $\ker\varphi$,
its embedding into the object M,

$$\begin{tikzcd} \ker\varphi \arrow[dr, hook, "\kappa"] & & \\ & M \arrow[r, "\varphi"] & N \end{tikzcd}$$

A simple example: QVectorSpaces

Actually, there is more to say about the kernel: To handle the kernel of φ algorithmically ...

... one has to construct the object $\ker\varphi$,
its embedding into the object M,
and for every test object T

$$\begin{tikzcd}
\ker\varphi \arrow[dr, hook, "\kappa"] \arrow[drr, bend left, "0"] & & \\
 & M \arrow[r, "\varphi"] & N \\
T \arrow[ur, "\tau"] \arrow[urr, bend right, "0"'] & &
\end{tikzcd}$$

A simple example: QVectorSpaces

Actually, there is more to say about the kernel: To handle the kernel of φ algorithmically ...

... one has to construct the object kerφ,
its embedding into the object M,
and for every test object T
a morphism given by kerφ's universal property.

$$\begin{tikzcd}
\ker\varphi \arrow[dr, hook, "\kappa"] \arrow[drr, bend left, "0"] & & \\
& M \arrow[r, "\varphi"] & N \\
T \arrow[uu, dotted, "\tau/\kappa"] \arrow[ur, "\tau"] \arrow[urr, bend right, "0"'] & &
\end{tikzcd}$$

A simple example: QVectorSpaces

Actually, there is more to say about the kernel: To handle the kernel of φ algorithmically ...

... one has to construct the object $\ker\varphi$,
its embedding into the object M,
and for every test object T
a morphism given by $\ker\varphi$'s universal property.

$$
\begin{tikzcd}
\ker\varphi \arrow[dr, hook, "\kappa"] \arrow[drr, bend left, "0"] & & \\
& M \arrow[r, "\varphi"] & N \\
T \arrow[uu, dotted, "\tau/\kappa"] \arrow[ur, "\tau"] \arrow[urr, bend right, "0"'] & &
\end{tikzcd}
$$

Thus a proper implementation of the kernel needs **<u>three</u>** algorithms.

A simple example: QVectorSpaces

After having implemented these basic algorithms, `CategoriesForHomalg` provides derived algorithms.

A simple example: QVectorSpaces

After having implemented these basic algorithms, `CategoriesForHomalg` provides derived algorithms.
Example: Factorization of a morphism into an epi and a mono.

$$M \xrightarrow{\varphi} N$$

A simple example: QVectorSpaces

After having implemented these basic algorithms, `CategoriesForHomalg` provides derived algorithms.
Example: Factorization of a morphism into an epi and a mono.

$$\ker(\varphi) \xhookrightarrow{\iota} M \xrightarrow{\quad\quad\varphi\quad\quad} N$$

```
iota := KernelEmb( phi );
```

A simple example: QVectorSpaces

After having implemented these basic algorithms, `CategoriesForHomalg` provides derived algorithms.
Example: Factorization of a morphism into an epi and a mono.

$$\begin{tikzcd} \ker(\varphi) \arrow[r, hook, "\iota"] & M \arrow[rr, "\varphi"] \arrow[dr, two heads] & & N \\ & & \mathrm{coker}(\iota) & \end{tikzcd}$$

```
iota := KernelEmb( phi );
epimorphism := CokernelProj( iota );
```

A simple example: QVectorSpaces

After having implemented these basic algorithms, `CategoriesForHomalg` provides derived algorithms.
Example: Factorization of a morphism into an epi and a mono.

$$
\begin{tikzcd}
\ker(\varphi) \arrow[r, hook, "\iota"] & M \arrow[rr, "\varphi"] \arrow[dr, two heads] & & N \\
& & \mathrm{coker}(\iota) \arrow[ur, hook, dotted] &
\end{tikzcd}
$$

```
iota := KernelEmb( phi );

epimorphism := CokernelProj( iota );

monomorphism := CokernelColift( iota, phi );
```

A simple example: QVectorSpaces

After having implemented these basic algorithms, `CategoriesForHomalg` provides derived algorithms.
Example: Factorization of a morphism into an epi and a mono.

$$\begin{tikzcd}
\ker(\varphi) \arrow[r, hook, "\iota"] & M \arrow[rr, "\varphi"] \arrow[dr, two heads] & & N \\
& & \mathrm{coker}(\iota) \arrow[ur, hook, dotted] &
\end{tikzcd}$$

```
iota := KernelEmb( phi );

epimorphism := CokernelProj( iota );

monomorphism := CokernelColift( iota, phi );

return [ epimorphism, monomorphism ];
```

A simple example: QVectorSpaces

After having implemented these basic algorithms, `CategoriesForHomalg` provides derived algorithms.

A simple example: QVectorSpaces

After having implemented these basic algorithms, `CategoriesForHomalg` provides derived algorithms.
Another example: Pullback.

$$\begin{tikzcd} & M \arrow[d, "\mu"] \\ N \arrow[r, "\nu"] & B \end{tikzcd}$$

A simple example: QVectorSpaces

After having implemented these basic algorithms, `CategoriesForHomalg` provides derived algorithms.
Another example: Pullback.

$$\begin{tikzcd} M \oplus N \arrow[r, "\pi_M"] \arrow[d, "\pi_N"'] & M \arrow[d, "\mu"] \\ N \arrow[r, "\nu"] & B \end{tikzcd}$$

1. Compute $M \oplus N$ and projection maps.

A simple example: QVectorSpaces

After having implemented these basic algorithms, `CategoriesForHomalg` provides derived algorithms.
Another example: Pullback.

$$\begin{tikzcd} M \oplus N \arrow[r, "\pi_M"] \arrow[d, "\pi_N"'] \arrow[dr, "\delta"] & M \arrow[d, "\mu"] \\ N \arrow[r, "\nu"] & B \end{tikzcd}$$

1. Compute $M \oplus N$ and projection maps.
2. Compute $\delta := \mu \circ \pi_M - \nu \circ \pi_N$.

A simple example: QVectorSpaces

After having implemented these basic algorithms, `CategoriesForHomalg` provides derived algorithms.
Another example: Pullback.

$$\begin{tikzcd} \ker(\varphi) \arrow[dr, hook] & & \\ & M \oplus N \arrow[r, "\pi_M"] \arrow[d, "\pi_N"'] \arrow[dr, "\delta"] & M \arrow[d, "\mu"] \\ & N \arrow[r, "\nu"] & B \end{tikzcd}$$

1. Compute $M \oplus N$ and projection maps.
2. Compute $\delta := \mu \circ \pi_M - \nu \circ \pi_N$.
3. Compute the kernel embedding of δ.

A simple example: QVectorSpaces

After having implemented these basic algorithms, `CategoriesForHomalg` provides derived algorithms.
Another example: Pullback.

$$
\begin{tikzcd}
\ker(\varphi) \arrow[dr, hook] \arrow[drr, bend left] \arrow[ddr, bend right] & & \\
& M \oplus N \arrow[r, "\pi_M"] \arrow[d, "\pi_N"'] \arrow[dr, "\delta"] & M \arrow[d, "\mu"] \\
& N \arrow[r, "\nu"] & B
\end{tikzcd}
$$

1. Compute $M \oplus N$ and projection maps.
2. Compute $\delta := \mu \circ \pi_M - \nu \circ \pi_N$.
3. Compute the kernel embedding of δ.

Advantages of categorial programming

Advantages of categorial programming

- Deduce higher algorithms from basic algorithms.

Advantages of categorial programming

- Deduce higher algorithms from basic algorithms.
- Generate new data structures from old ones with low effort.

Advantages of categorial programming

- Deduce higher algorithms from basic algorithms.
- Generate new data structures from old ones with low effort.
- Create a “type-safe” enviroment with objects and morphisms living in exactly one category

Advantages of categorial programming

- Deduce higher algorithms from basic algorithms.
- Generate new data structures from old ones with low effort.
- Create a “type-safe” enviroment with objects and morphisms living in exactly one category and with functors as converters.

Who profits?

Who profits?

- Programmers who wants to embed their code in a categorial setup.

Who profits?

- Programmers who wants to embed their code in a categorial setup.
- Mathematicians/ Physicists experimenting with complex mathematical objects.

Functors: modelling relations between data types

Structure of a functor

- A functor is modeled as a morphism in the category of categories.

Functors: modelling relations between data types

Structure of a functor

- A functor is modeled as a morphism in the category of categories. **They can be composed!**

Functors: modelling relations between data types

Structure of a functor

- A functor is modeled as a morphism in the category of categories. **They can be composed!**
- Each functor can contain functions to be applied to objects, morphisms, etc.

Functors: modelling relations between data types

Structure of a functor

- A functor is modeled as a morphism in the category of categories. **They can be composed!**
- Each functor can contain functions to be applied to objects, morphisms, etc.

```
gap> AddObjectFunction( transpose,
>             obj -> Opposite( obj ) );;
gap> AddMorphismFunction( transpose,
>        func( source_new, mor, range_new )
>            return Transpose( Opposite( mor ) );
>          end );;
gap> ApplyFunctor( transpose, Opposite( tau ) );
<A morphism in the category QVectorSpaces>
```

Why are functors important?

- Functors model relations and translations between categories.

Why are functors important?

- Functors model relations and translations between categories.
- Data structures can be completely modeled by functors.

Why are functors important?

- Functors model relations and translations between categories.
- Data structures can be completely modeled by functors.
- Soon: One data structure for complexes as functors from **Integers** to a category

About the implementation of functors

Crucial for functors: Caching

- If a functor F is applied to two morphisms $A \to B$ and $B \to C$, the resulting morphisms $F(A) \to F(B)$ and $F(B) \to F(C)$ should be composable:

About the implementation of functors

Crucial for functors: Caching

- If a functor F is applied to two morphisms $A \to B$ and $B \to C$, the resulting morphisms $F(A) \to F(B)$ and $F(B) \to F(C)$ should be composable:

We need the two $F(B)$ to be identical.

About the implementation of functors

Crucial for functors: Caching

- If a functor F is applied to two morphisms $A \to B$ and $B \to C$, the resulting morphisms $F(A) \to F(B)$ and $F(B) \to F(C)$ should be composable:

  We need the two $F(B)$ to be identical.

- Therefore functors store their computed values.

Data structures for localization

Generalized morphism

Every category $\mathcal{A}$ defined in `CategoriesForHomalg` has an associated **Generalized morphism category** $\mathrm{G}(\mathcal{A})$.

Data structures for localization

Generalized morphism

Every category $\mathcal{A}$ defined in `CategoriesForHomalg` has an associated **Generalized morphism category** $\mathrm{G}(\mathcal{A})$. In this category every monomorphism or epimorphism of $\mathcal{A}$ is split.

Data structures for localization

Generalized morphism

Every category $\mathcal{A}$ defined in `CategoriesForHomalg` has an associated **Generalized morphism category** $\mathrm{G}(\mathcal{A})$. In this category every monomorphism or epimorphism of $\mathcal{A}$ is split.

Diagram chases become possible

*Example: Snake lemma

Find the snake using generalized morphisms.

$$
\begin{tikzcd}
 & & & \ker \arrow[d] & \\
 & A' \arrow[r] \arrow[d] & B' \arrow[r] \arrow[d] & C' \arrow[r] \arrow[d] & 0 \\
0 \arrow[r] & A \arrow[r] \arrow[d] & B \arrow[r] & C & \\
 & \operatorname{coker} & & &
\end{tikzcd}
$$

*Example: Snake lemma

Find the snake using generalized morphisms.

$$
\begin{tikzcd}
 & & & \mathrm{ker} \arrow[d] & \\
 & A' \arrow[r] \arrow[d] & B' \arrow[r] \arrow[d] & C' \arrow[r] \arrow[d] & 0 \\
0 \arrow[r] & A \arrow[r] \arrow[d] & B \arrow[r] & C & \\
 & \mathrm{coker} & & &
\end{tikzcd}
$$

*Example: Snake lemma

Find the snake using generalized morphisms.

$$\begin{tikzcd}
 & & & \mathrm{ker} \arrow[d] & \\
 & A' \arrow[r] \arrow[d] & B' \arrow[r] \arrow[d] & C' \arrow[r] \arrow[d] \arrow[l, bend right] & 0 \\
0 \arrow[r] & A \arrow[r] \arrow[d] & B \arrow[r] & C & \\
 & \mathrm{coker} & & &
\end{tikzcd}$$

*Example: Snake lemma

Find the snake using generalized morphisms.

$$
\begin{tikzcd}
 & & & \ker \arrow[d] & \\
 & A' \arrow[r] \arrow[d] & B' \arrow[r] \arrow[d] & C' \arrow[r] \arrow[d] \arrow[l, bend right] & 0 \\
0 \arrow[r] & A \arrow[r] \arrow[d] & B \arrow[r] & C & \\
 & \operatorname{coker} & & &
\end{tikzcd}
$$

*Example: Snake lemma

Find the snake using generalized morphisms.

$$\begin{tikzcd}
 & & & \ker \arrow[d] & \\
 & A' \arrow[r] \arrow[d] & B' \arrow[r] \arrow[d] & C' \arrow[r] \arrow[d] \arrow[l, bend right] & 0 \\
0 \arrow[r] & A \arrow[r] \arrow[d] & B \arrow[r] \arrow[l, bend left] & C & \\
 & \operatorname{coker} & & &
\end{tikzcd}$$

*Example: Snake lemma

Find the snake using generalized morphisms.

$$
\begin{tikzcd}
 & & & \ker \arrow[d] & \\
 & A' \arrow[r] \arrow[d] & B' \arrow[r] \arrow[d] & C' \arrow[r] \arrow[d] \arrow[l, bend right] & 0 \\
0 \arrow[r] & A \arrow[r] \arrow[d] & B \arrow[r] \arrow[l, bend left] & C & \\
 & \operatorname{coker} & & &
\end{tikzcd}
$$

*Example: Snake lemma

Find the snake using generalized morphisms.

$$
\begin{tikzcd}
 & & & \mathrm{ker} \arrow[d] \arrow[dddll, rounded corners] & \\
 & A' \arrow[r] \arrow[d] & B' \arrow[r] \arrow[d] & C' \arrow[r] \arrow[d] \arrow[l, bend right] & 0 \\
0 \arrow[r] & A \arrow[r] \arrow[d] & B \arrow[r] \arrow[l, bend left] & C & \\
 & \mathrm{coker} & & &
\end{tikzcd}
$$

ToolsForHomalg - Advanced technical features

`ToolsForHomalg`
Advanced technical features

Data structure for caches

`ToolsForHomalg` offers convenient caches.

Data structure for caches

`ToolsForHomalg` offers convenient caches.

Caching in `ToolsForHomalg`

- Caches are `GAP` objects which take a list of keys and possibly return a stored `GAP` object.

Data structure for caches

`ToolsForHomalg` offers convenient caches.

Caching in `ToolsForHomalg`

- Caches are `GAP` objects which take a list of keys and possibly return a stored `GAP` object.
- All objects are stored in weak pointer lists.

Data structure for caches

`ToolsForHomalg` offers convenient caches.

Caching in `ToolsForHomalg`

- Caches are `GAP` objects which take a list of keys and possibly return a stored `GAP` object.
- All objects are stored in weak pointer lists.

```
gap> cache := CachingObject( 2 );
<A cache with keylength 2, 0 hits, 0 misses>
gap> SetCacheValue( cache, [ S, T ], U );;
gap> GetCacheValue( cache, [ S, T ] );
U
gap> cache;
<A cache with keylength 2, 1 hits, 0 misses>
```

Convenience for caches: InstallMethodWithCache

Install a method with cache

Use `InstallMethodWithCache` instead of `InstallMethod` to install a method which stores its results.

Convenience for caches: InstallMethodWithCache

Install a method with cache

Use `InstallMethodWithCache` instead of `InstallMethod` to install a method which stores its results.

```
InstallMethodWithCache( DirectProductFunctor,
                 [ IsHomalgCategory, IsInt ],

  function( category, number_of_arguments )
    local direct_product_functor;
      ...
    return direct_product_functor;
end );
```

Convenience for caches: InstallMethodWithCache

Install a method with cache

Use `InstallMethodWithCache` instead of `InstallMethod` to install a method which stores its results.

```
InstallMethodWithCache( DirectProductFunctor,
                        [ IsHomalgCategory, IsInt ],

  function( category, number_of_arguments )
    local direct_product_functor;
      ...
    return direct_product_functor;
end );
```

At the second call of this function with identical input, the stored value is returned.

Convenience for caches: InstallMethodWithCacheFromObject

Install a method with cache extracted from argument

- `InstallMethodWithCacheFromObject` works like `InstallMethodWithCache`, but extracts the cache from one of its arguments.

Convenience for caches: InstallMethodWithCacheFromObject

Install a method with cache extracted from argument

- `InstallMethodWithCacheFromObject` works like `InstallMethodWithCache`, but extracts the cache from one of its arguments.
- One needs to implement a function `CachingObject` which can be applied to the argument.

Convenience for caches: InstallMethodWithCacheFromObject

*Install the API function

```
InstallMethod( CachingObject,
               [ IsHomalgCategory, IsString, IsInt ],
  function( category, name, number )
    local cache;

    if IsBound( category!.caches.(name) ) then
        return category!.caches.(name);
    fi;

    cache := CachingObject( number );
    category!.caches.(name) := cache;
    return cache;
end );
```

Convenience for caches: DeclareOperationWithCache

Declare an operation with cache

One can also declare an operation to be cached.

```
DeclareOperationWithCache( "DirectSum",
            [ IsHomalgCategoryObject,
              IsHomalgCategoryObject ] );
```

Convenience for caches: DeclareOperationWithCache

Declare an operation with cache

One can also declare an operation to be cached.

```
DeclareOperationWithCache( "DirectSum",
            [ IsHomalgCategoryObject,
              IsHomalgCategoryObject ] );
```

This defines and installs `SetDirectSum` and `HasDirectSum` which work as usual.

ToDoList

ToDoLists are a tool to keep track of logical implications.

ToDoList

`ToDoLists` are a tool to keep track of logical implications.

What to do with them

- Set possible complex logical implication as a `ToDoListEntry`.

ToDoList

`ToDoLists` are a tool to keep track of logical implications.

What to do with them

- Set possible complex logical implication as a `ToDoListEntry`.
- They can apply theorems and spread knowledge.

`ToDoList` example

Example taken from `ToricVarieties`

If U is an affine toric variety, the PICARD group is trivial.

ToDoList example

Example taken from ToricVarieties

If U is an affine toric variety, the PICARD group is trivial.

```
U := ToricVariety( ... );
D := Divisor( U, ... );
ToDoListEntry( rec( Source := [
                   rec( object := U,
                        attribute := "IsAffine",
                        value := true ),
                   rec( object := D,
                        attribute := "IsCartier",
                        value := true ) ],
          Range := rec( object := D
                        attribute := "IsPrincipal",
                        value := true ) );
```

Advantages of `ToDoLists`

Advantages

- Complex logical relations can be modeled.

Advantages of `ToDoLists`

Advantages

- Complex logical relations can be modeled.
- `ToDoLists` keep track of propagated knowledge.

Advantages of `ToDoLists`

Advantages

- Complex logical relations can be modeled.
- `ToDoLists` keep track of propagated knowledge.
- This can be used to create proofs.

Advantages of `ToDoLists`

Advantages

- Complex logical relations can be modeled.
- `ToDoLists` keep track of propagated knowledge.
- This can be used to create proofs.
- A scheduling system might be possible.

Generic view - a tool to create view methods

What is generic view

- Properties and attributes of objects can be listed in a graph with implications to create generic view methods.

Generic view - a tool to create view methods

What is generic view

- Properties and attributes of objects can be listed in a graph with implications to create generic view methods.
- View and display can then be installed using this graphs.

Generic view - a tool to create view methods

What is generic view

- Properties and attributes of objects can be listed in a graph with implications to create generic view methods.
- View and display can then be installed using this graphs.

```
print_graph :=
    CreatePrintingGraph( IsHomalgCategoryMorphism );

AddRelationToGraph( print_graph,
   rec( Source := [ "IsSplitMonomorphism" ],
        Range := [ rec( Conditions := "IsMonomorphism",
        PrintString := "mono",
        Adjective := true ) ] ) );

InstallPrintFunctionsOutOfPrintingGraph( print_graph );
```

Advantages of generic view

Advantages

- Graph keeps track of relations between printed attributes.

Advantages of generic view

Advantages

- Graph keeps track of relations between printed attributes.
- Additionally, `FullView` and `FullViewWithEverythingComputed` are installed.

Advantages of generic view

Advantages

- Graph keeps track of relations between printed attributes.
- Additionally, `FullView` and `FullViewWithEverythingComputed` are installed.

```
gap> FullView( tau );
Full description:
morphism in the category QVectorSpaces
- iso: not computed yet
- mono: true
- split mono: true
- epi: not computed yet
- split epi: not computed yet
- identity: false
```