Cap Manual

Sebastian Gutsche

Sebastian Posur

CHAPTER 1

Introduction

CAP is an acronym for categories, algorithms, and programming. It is a software project implemented in GAP.

- CAP derives powerful algorithms and data structures from basic categorical constructions.
- CAP serves as a categorical programming language in which you can realize your code in a categorically structured way.
- CAP simplifies complex computations by applying theorems.

We call this concept **categorical programming**. This manual provides a short tutorial for CAP and explains its main features. You can clone the CAP git repository from

https://github.com/homalg-project/CAP_project.

CHAPTER 2

Tutorial and Quickstart

Although CAP offers many features and many possibilities to tweak computations, it is fairly simple to integrate already existing data types and algorithms in GAP into the CAP setup. In this chapter we give you a hand at the very first steps, using some examples and explaining the use and the effect of the commands in it.

1. The category of groups

When implementing a category in CAP, one should always have the corresponding classical category in mind. In this example we start with the category of groups. The objects in this category will be the groups, the morphisms will be the homomorphisms of groups. Every category in CAP is presented by a category GAP object. We start by creating this object.

```
gap> LoadPackage( "CAP" );
true
gap> grps := CreateCapCategory( "groups" );
groups
```

The string is just the name of the category we have created, and the object the variable grps refers to stores information about the category. We can now continue by telling the category how operations on the objects or morphisms are performed by adding functions the category. First, there should be a compose method for morphisms. In GAP morphisms of groups are composed via the * operator, so we use the two argument function * as function for PreCompose.

```
gap> AddPreCompose( grps, \* );
```

Now the composition of two morphisms will be computed by the * operation. Another thing every category needs is a function to compute the identity morphism.

```
gap> identity_func := grp -> GroupHomomorphismByImages( grp, grp );
function( grp ) ... end
gap> AddIdentityMorphism( grps, identity_func );
```

The command used above exactly creates the identity morphism of a group. Next, we tell CAP that every output of a basic operation for the category grps should automatically be added to the category grps. This step is necessary since in this example, we simply integrate already existing algorithms and data structures of GAP to CAP's infrastructure.

```
gap> EnableAddForCategoricalOperations( grps );
```

Since the category has now all the functions we wanted it to have, we can finalize it.

```
gap> Finalize( grps );
true
```

Finalizing a category is necessary after adding all the desired operations and before constructing objects for it. We can now create a group and tell the system that the group should be an object in the category. Note that `AddObject` can only be applied to `GAP` objects lying in the filter `IsAttributeStoringRep`.

```
gap> S3 := SymmetricGroup( 3 );
Sym( [ 1 .. 3 ] )
gap> AddObject( grps, S3 );
gap> S4 := SymmetricGroup( 4 );
Sym( [ 1 .. 4 ] )
gap> AddObject( grps, S4 );
```

Now, CAP considers those groups as objects in the category `grps`, and we can ask them about their category. Also, they are now part of the `GAP` filter `IsCapCategoryObject`.

```
gap> CapCategory( S3 );
groups
gap> IsCapCategoryObject( S4 );
true
```

Now manipulation of the groups is possible using the functions we have already provided to the category. It is possible to construct the identity of a group, and compose it with itself.

```
gap> id_S3 := IdentityMorphism( S3 );
IdentityMapping( Sym( [ 1 .. 3 ] ) )
gap> PreCompose( id_S3, id_S3 );
IdentityMapping( Sym( [ 1 .. 3 ] ) )
```

Of course, one can also create a morphism between `S3` and `S4` and add it to the category. After that, we can also compose it with the identity morphism.

```
gap> S3_S4 := GroupHomomorphismByImages( S3, S4, GeneratorsOfGroup(S3) );
[ (1,2,3), (1,2) ] -> [ (1,2,3), (1,2) ]
gap> AddMorphism( grps, S3_S4 );
gap> PreCompose( id_S3, S3_S4 );
[ (1,2,3), (1,2) ] -> [ (1,2,3), (1,2) ]
```

Please note that the constructors for objects and morphisms used in this example are the ones provided by `GAP` itself, and the only "change" done to the data structure was

adding it to the category. This is one of the design principles of CAP. Already existing data structures and algorithms can be integrated into the system with little to no effort, which makes it possible to integrate CAP into many existing projects.

This example was pretty basic and only a way to show the very basic structure of CAP. The next example will construct a more sophisticated category and some of the computational tools in CAP.

2. The category of rational vector spaces

In this section, we will create the category of finite dimensional rational vector spaces as an example of an abelian category. We are going to realize this category as a skeletal category, i.e., the objects will be the non-negative integers, representing the dimension of the vector space, and we use appropriate matrices as morphisms. We start by loading CAP and the package `MatricesForHomalg` of the `homalg` project, and creating appropriate data structures for the objects and morphisms of the category. The first part of this tutorial is taken from a file and not from an interactive session. Please note that for the sake of completeness, all the necessary code to create the functions is displayed. This might seem confusing or simply too much. If you only want to have a look how a CAP category is initialized, only look at the headers of the functions. Also, once the whole process of creation is finished, there are examples for computations in the category. If you are interested in the capability of CAP, please continue at 3.

In the first part, after loading the packages, we create types and representations for the objects and morphisms of the category. Those are needed since we want to provide the data structure. For simplicity, those representations are derived from the corresponding CAP `GAP` categories. This is not needed in general, so one can use their own data structures without any changes. After that, we declare an attribute for vector spaces, i.e., the dimension, which will be the only needed data of an object, as described above. Then we declare constructors for objects and morphisms. The object constructor only has one attribute, which is the dimension of the vector space. The morphism constructor has three arguments. A vector space, which is the `Source` of the morphism, a matrix representing the morphism, and a second vector space, which is the `Range` of the morphism. Note that every morphism which should be added to the category needs the `GAP` attributes `Source` and `Range`, which will also be added as objects to the category.

```
LoadPackage( "CAP" );

LoadPackage( "MatricesForHomalg" );

DeclareRepresentation( "IsHomalgRationalVectorSpaceRep",
                       IsCapCategoryObjectRep,
                       [ ] );

BindGlobal( "TheTypeOfHomalgRationalVectorSpaces",
        NewType( TheFamilyOfCapCategoryObjects,
                IsHomalgRationalVectorSpaceRep ) );
```

```
DeclareRepresentation( "IsHomalgRationalVectorSpaceMorphismRep",
                       IsCapCategoryMorphismRep,
                       [ ] );

BindGlobal( "TheTypeOfHomalgRationalVectorSpaceMorphism",
        NewType( TheFamilyOfCapCategoryMorphisms,
                IsHomalgRationalVectorSpaceMorphismRep ) );

DeclareAttribute( "Dimension",
                  IsHomalgRationalVectorSpaceRep );

DeclareOperation( "QVectorSpace",
                  [ IsInt ] );

DeclareOperation( "VectorSpaceMorphism",
                  [ IsHomalgRationalVectorSpaceRep, IsObject,
                    IsHomalgRationalVectorSpaceRep ] );

BindGlobal( "vecspaces", CreateCapCategory( "VectorSpaces" ) );

SetIsAbelianCategory( vecspaces, true );

BindGlobal( "VECTORSPACES_FIELD", HomalgFieldOfRationals( ) );
```

After the declarations of the constructors we created a global category GAP object, which will be the category of vector spaces and where we add all the objects and morphisms to, already in their constructors, which we create now. Since we are going to create an Abelian category, we tell the system that we are doing this, to have all the computational properties of an Abelian category.

```
##
InstallMethod( QVectorSpace,
               [ IsInt ],

  function( dim )
    local space;

    space := rec( );

    ObjectifyWithAttributes( space, TheTypeOfHomalgRationalVectorSpaces,
                             Dimension, dim
    );

    Add( vecspaces, space );
```

```
    return space;

end );

##
InstallMethod( VectorSpaceMorphism,
                  [ IsHomalgRationalVectorSpaceRep, IsObject,
                    IsHomalgRationalVectorSpaceRep ],

  function( source, matrix, range )
    local morphism;

    if not IsHomalgMatrix( matrix ) then

      morphism := HomalgMatrix( matrix, Dimension( source ), Dimension( range ),
                                VECTORSPACES_FIELD );

    else

      morphism := matrix;

    fi;

    morphism := rec( morphism := morphism );

    ObjectifyWithAttributes( morphism,
                             TheTypeOfHomalgRationalVectorSpaceMorphism,
                             Source, source,
                             Range, range
    );

    Add( vecspaces, morphism );

    return morphism;

end );
```

Those constructors are straight forward, they create the objects and morphisms with the desired attributes and then add those to our previously created category. Since the objects, because of their type, already imply the GAP filters `IsCapCategoryObject` and `IsCapCategoryMorphism` respectively, we can use the operation `Add` instead of `AddObject` and `AddMorphism`. Since `Add` is called in the constructor, the resulting objects and morphisms are automatically part of the category, and after finishing up the category all the operations from the category will be applicable to them.

We now continue by adding functions to the category, starting with the same operations as above.

If you want to test your implementation function by function, please remember to use the command `Finalize( vecspaces )` before testing.

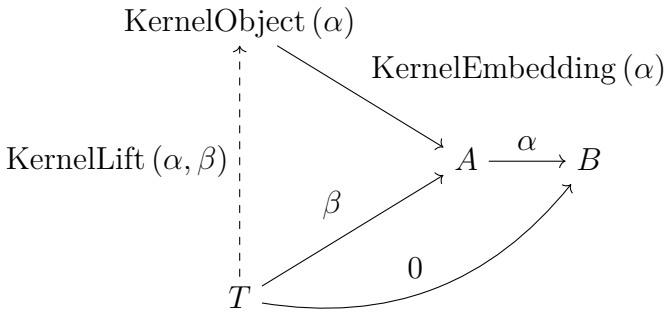
```
AddIdentityMorphism( vecspaces,

  function( obj )

    return VectorSpaceMorphism( obj,
          HomalgIdentityMatrix( Dimension( obj ), VECTORSPACES_FIELD ), obj );

end );

AddPreCompose( vecspaces,

  function( mor_left, mor_right )
    local composition;

    composition := mor_left!.morphism * mor_right!.morphism;

    return VectorSpaceMorphism( Source( mor_left ),
                  composition, Range( mor_right ) );

end );
```

Those functions are straight forward and do not require any additional explanation except the code. Before we continue adding additional operations, there is another thing we have to take care of. The category of vector spaces we wanted to create is a skeletal category, i.e., vector spaces of the same dimension are supposed to be equal. At the moment two consecutive calls of the object constructor will not lead to equal objects, since there is no comparison function. CAP offers facilities to give correct comparison functions for objects and morphisms to the category. If no function is given, the GAP function `IsIdenticalObj` is used. However, giving the right equality function to the category is important, please see section 3 about equalities for details. We will now add equalities for objects and morphisms. Objects will be compared by their dimension, for morphisms the matrices will be compared entry-wise. The equalities in CAP do not need to be `true` or `false`, but are also allowed to return `fail`, which will be interpreted as non decidable. This is used for example for complexes.

```
AddIsEqualForObjects( vecspaces,

  function( vecspace_1, vecspace_2 )

    return Dimension( vecspace_1 ) = Dimension( vecspace_2 );
```

```
end );

AddIsEqualForMorphisms( vecspaces,

  function( a, b )

    return a!.morphism = b!.morphism;

end );
```

Next we implement some additional functions in the category of vector spaces. We start by defining functions for the kernel. A proper implementation needs at least two functions. One for `KernelEmbedding`, which can compute the embedding of the kernel into the source of the given morphism. From this function then automatically a function for `KernelObject` is derived, which computes the embedding and returns the source of this morphism. The second one, `KernelLift`, implements the universal property of the kernel. The diagram of the kernel looks like this:



Please note that it is important that the range of the output morphism of the function we give to `KernelLift`, under equal first argument, must coincide with the source of the output morphism of the function given to `KernelEmbedding`. In our case this simply means they need to have the same dimension.

```
AddKernelEmbedding( vecspaces,

  function( morphism )
    local kernel_emb, kernel_obj;

    kernel_emb := SyzygiesOfRows( morphism!.morphism );

    kernel_obj := QVectorSpace( NrRows( kernel_emb ) );

    return VectorSpaceMorphism( kernel_obj, kernel_emb, Source( morphism ) );

end );
```

```
AddKernelLift( vecspaces,

  function( morphism, test_morphism )
    local kernel_matrix;

    kernel_matrix := SyzygiesOfRows( morphism!.morphism );

    return VectorSpaceMorphism( Source( test_morphism ),
                    RightDivide( test_morphism!.morphism, kernel_matrix ),
                    QVectorSpace( NrRows( kernel_matrix ) ) );

end );
```

There are two things to mention here. First thing, instead of `KernelLift` we could also have installed another method, called `KernelLiftWithGivenKernelObject`. The result would again be an implementation of `KernelLift`, but those `WithGiven` operations offer an advantage. Instead of calling a function with only α and β, there would be a third argument, which should be equal to the output of `KernelObject` and then serves as the range of the output morphism. This can be done to ensure better compatibility if the given equality function for objects might not be complete, or to save computation time. Please have a look at 2.1 for more details about the `WithGiven` functions.

Also, the function given for `KernelLift` does at the beginning the same as the function given for `KernelEmbedding`. This might to a unnecessary computation. So, in this context, it would be better to provide a more general function for `LiftAlongMonomorphism`. The input for this function would be a mono, and some morphism, would then provide a lift. The function for `KernelLift` would then be derived from this operation and `KernelEmbedding`, by computing

$$\text{LiftAlongMonomorphism}\left(\text{KernelEmbedding}\left(\alpha\right), \beta\right).$$

For more information about this system, please refer to the derivations section 4. An implementation of the kernel would then instead look like this.

```
AddKernelEmbedding( vecspaces,

  function( morphism )
    local kernel_emb, kernel_obj;

    kernel_emb := SyzygiesOfRows( morphism!.morphism );

    kernel_obj := QVectorSpace( NrRows( kernel_emb ) );

    return VectorSpaceMorphism( kernel_obj, kernel_emb, Source( morphism ) );

end );

AddLiftAlongMonomorphism( vecspaces,
```

```
  function( monomorphism, test_morphism )

    return VectorSpaceMorphism( Source( test_morphism ),
           RightDivide( test_morphism!.morphism, monomorphism!.morphism ),
           Source( monomorphism ) );

end );
```

We now see the code duplication from above disappear.

We now continue by adding more functionality to the category. Dual to our second implementation of the kernel, we will provide functions for the cokernel.

```
AddCokernelProjection( vecspaces,

  function( morphism )
    local cokernel_proj, cokernel_obj;

    cokernel_proj := SyzygiesOfColumns( morphism!.morphism );

    cokernel_obj := QVectorSpace( NrColumns( cokernel_proj ) );

    return VectorSpaceMorphism( Range( morphism ),
              cokernel_proj, cokernel_obj );

end );

AddColiftAlongEpimorphism( vecspaces,

  function( epimorphism, test_morphism )

    return VectorSpaceMorphism( Range( epimorphism ),
              LeftDivide( epimorphism!.morphism, test_morphism!.morphism ),
              Range( test_morphism ) );

end );
```

The next step is implementing functions for the ZeroObject in the category, i.e. the vector space of dimension 0. A proper implementation here needs three algorithms. One, without any arguments, returns the zero object of the category. The other two should, given one object, compute the unique morphisms from and into the zero object. Those are named `UniversalMorphismIntoZeroObject` and `UniversalMorphismFromZeroObject`. For those there are again two choices, either using the `WithGiven` operations or not. For our example we do both, providing functions for the `WithGiven` operations and for the ones that need to compute their own zero object. Please note that this is again possible because of the installed equality of objects.

```
AddZeroObject( vecspaces,

  function( )

    return QVectorSpace( 0 );

end );

AddUniversalMorphismIntoZeroObject( vecspaces,

  function( source )

    return VectorSpaceMorphism( source,
              HomalgZeroMatrix( Dimension( source ), 0, VECTORSPACES_FIELD ),
              QVectorSpace( 0 ) );

end );

AddUniversalMorphismIntoZeroObjectWithGivenZeroObject( vecspaces,

  function( source, terminal_object )

    return VectorSpaceMorphism( source,
              HomalgZeroMatrix( Dimension( source ), 0, VECTORSPACES_FIELD ),
              terminal_object );

end );

AddUniversalMorphismFromZeroObject( vecspaces,

  function( sink )

    return VectorSpaceMorphism( QVectorSpace( 0 ),
              HomalgZeroMatrix( 0, Dimension( sink ), VECTORSPACES_FIELD ),
              sink );

end );

AddUniversalMorphismFromZeroObjectWithGivenZeroObject( vecspaces,

  function( sink, initial_object )

    return VectorSpaceMorphism( initial_object,
                  HomalgZeroMatrix( 0, Dimension( sink ), VECTORSPACES_FIELD ),
                  sink );
```

```
end );
```

Now we turn the set of homomorphisms between two objects into an Abelian group. For this, we need to define the addition of two morphisms having the same sources and ranges, the additive inverse of a morphism, and the zero morphism between two objects.

```
AddAdditionForMorphisms( vecspaces,

  function( a, b )

    return VectorSpaceMorphism( Source( a ),
               a!.morphism + b!.morphism,
               Range( a ) );

end );

AddAdditiveInverseForMorphisms( vecspaces,

  function( a )

    return VectorSpaceMorphism( Source( a ),
               - a!.morphism,
               Range( a ) );

end );

AddZeroMorphism( vecspaces,

  function( a, b )

    return VectorSpaceMorphism( a,
               HomalgZeroMatrix( Dimension( a ),
                                 Dimension( b ),
                                 VECTORSPACES_FIELD ),
               b );

end );
```

The installation of `ZeroMorphism` is not necessary here, since we have the zero object and the corresponding universal morphisms, this method could also be derived from those. Of course, providing a primitive implementation might increase the speed of some computations.

The last thing left to implement is the direct sum and its universal properties. After that, we can tell the system that the category is abelian, finalize it, and start computations. The direct sum needs, for a proper implementation, at least five functions. At first, the

direct sum of objects needs to be created. Then there are injections of the components and projections to the factors. Finally, since the direct sum is product and coproduct at the same time, there are two universal properties to be implemented. For details please have a look at 5.

```
AddDirectSum( vecspaces,

  function( object_product_list )
    local dim;

    dim := Sum( List( object_product_list, c -> Dimension( c ) ) );

    return QVectorSpace( dim );

end );

AddInjectionOfCofactorOfDirectSum( vecspaces,

  function( object_product_list, injection_number )
    local components, dim, dim_pre, dim_post, dim_cofactor, coproduct,
    number_of_objects, injection_of_cofactor;

    components := object_product_list;

    number_of_objects := Length( components );

    dim := Sum( components, c -> Dimension( c ) );

    dim_pre := Sum( components{ [ 1 .. injection_number - 1 ] },
                    c -> Dimension( c ) );

    dim_post := Sum( components{ [ injection_number + 1 .. number_of_objects ] },
                     c -> Dimension( c ) );

    dim_cofactor := Dimension( object_product_list[ injection_number ] );

    coproduct := QVectorSpace( dim );

    injection_of_cofactor := HomalgZeroMatrix( dim_cofactor,
                                               dim_pre,
                                               VECTORSPACES_FIELD );

    injection_of_cofactor := UnionOfColumns( injection_of_cofactor,
                                        HomalgIdentityMatrix( dim_cofactor,
                                           VECTORSPACES_FIELD ) );
```

```
    injection_of_cofactor := UnionOfColumns( injection_of_cofactor,
                                    HomalgZeroMatrix( dim_cofactor,
                                                      dim_post,
                                                      VECTORSPACES_FIELD ) );

    return VectorSpaceMorphism( object_product_list[ injection_number ],
                                injection_of_cofactor, coproduct );

end );

AddUniversalMorphismFromDirectSum( vecspaces,

  function( diagram, sink )
    local dim, coproduct, components, universal_morphism, morphism;

    components := sink;

    dim := Sum( components, c -> Dimension( Source( c ) ) );

    coproduct := QVectorSpace( dim );

    universal_morphism := sink[1]!.morphism;

    for morphism in components{ [ 2 .. Length( components ) ] } do

      universal_morphism := UnionOfRows( universal_morphism,
                                         morphism!.morphism );

    od;

    return VectorSpaceMorphism( coproduct, universal_morphism,
                                   Range( sink[1] ) );

end );

AddProjectionInFactorOfDirectSum( vecspaces,

  function( object_product_list, projection_number )
    local components, dim, dim_pre, dim_post, dim_factor, direct_product,
          number_of_objects, projection_in_factor;

    components := object_product_list;

    number_of_objects := Length( components );

    dim := Sum( components, c -> Dimension( c ) );
```

```
    dim_pre := Sum( components{ [ 1 .. projection_number - 1 ] },
                    c -> Dimension( c ) );

    dim_post := Sum( components{ [ projection_number + 1
                                   .. number_of_objects ] },
                                   c -> Dimension( c ) );

    dim_factor := Dimension( object_product_list[ projection_number ] );

    direct_product := QVectorSpace( dim );

    projection_in_factor := HomalgZeroMatrix( dim_pre,
                                              dim_factor,
                                              VECTORSPACES_FIELD );

    projection_in_factor := UnionOfRows( projection_in_factor,
                                         HomalgIdentityMatrix( dim_factor,
                                            VECTORSPACES_FIELD ) );

    projection_in_factor := UnionOfRows( projection_in_factor,
                                         HomalgZeroMatrix( dim_post,
                                                   dim_factor,
                                                   VECTORSPACES_FIELD ) );

    return VectorSpaceMorphism( direct_product,
                                projection_in_factor,
                                object_product_list[ projection_number ] );

end );

AddUniversalMorphismIntoDirectSum( vecspaces,

  function( diagram, sink )
    local dim, direct_product, components, universal_morphism, morphism;

    components := sink;

    dim := Sum( components, c -> Dimension( Range( c ) ) );

    direct_product := QVectorSpace( dim );

    universal_morphism := sink[1]!.morphism;

    for morphism in components{ [ 2 .. Length( components ) ] } do
```

```
      universal_morphism := UnionOfColumns( universal_morphism,
                                            morphism!.morphism );

    od;

    return VectorSpaceMorphism( Source( sink[1] ),
                                universal_morphism,
                                direct_product );

end );
```

Now we can finalize the category.

```
Finalize( vecspaces );
```

The finalize step is in this case very important. It triggers further derivations of methods, which depend on the fact that some operations are not installed previously, and will not be installed anymore. Also it makes it possible to construct further categories, like the category of complexes. We now initialize an interactive session with this implementation of vector spaces and start by creating some morphisms.

```
gap> V := QVectorSpace( 2 );
<A rational vector space of dimension 2>
gap> W := QVectorSpace( 3 );
<A rational vector space of dimension 3>
gap> alpha := VectorSpaceMorphism( V, [ [ 1, 1, 1 ], [ -1, -1, -1 ] ], W );
A rational vector space homomorphism with matrix:
[ [   1,   1,   1 ],
  [  -1,  -1,  -1 ] ]
```

Unsurprisingly, we can now compute some of the stuff we told the system how to compute, for example the `KernelEmbedding`, which is the embedding of the kernel. Also, the cokernel and its projection can be computed. Those computations will be carried out using the functions we have given to the system. It is also possible to use the standard arithmetic on the morphism. Even if the added functions have different names, e.g., `AdditionForMorphisms`, it is possible to use the `GAP` arithmetic operations, e.g., `+`, for it.

```
gap> KernelEmbedding( alpha );
A rational vector space homomorphism with matrix:
[ [  1,  1 ] ]
gap> CokernelObject( alpha );
<A rational vector space of dimension 2>
gap> CokernelProjection( alpha );
A rational vector space homomorphism with matrix:
[ [  -1,  -1 ],
  [   1,   0 ],
  [   0,   1 ] ]
gap> alpha + alpha;
```

```
A rational vector space homomorphism with matrix:
[ [   2,   2,   2 ],
  [  -2,  -2,  -2 ] ]
gap> - alpha;
A rational vector space homomorphism with matrix:
[ [  -1,  -1,  -1 ],
  [   1,   1,   1 ] ]
```

The fact that such functions work properly are not surprising at all. But CAP has the power to derive further constructions from the given functions. Here are some examples of methods now applicable.

```
gap> IsMonomorphism( alpha );
false
gap> IsEpimorphism( alpha );
false
gap> alpha_image := ImageEmbedding( alpha );
A rational vector space homomorphism with matrix:
[ [  1,  1,  1 ] ]
```

This ImageEmbedding is computed by taking the KernelEmbedding of the CokernelProjection. There are many more constructions computable right now. One can see a list of all possible operations in the category in the category using the `InstalledMethodsOfCategory` command.

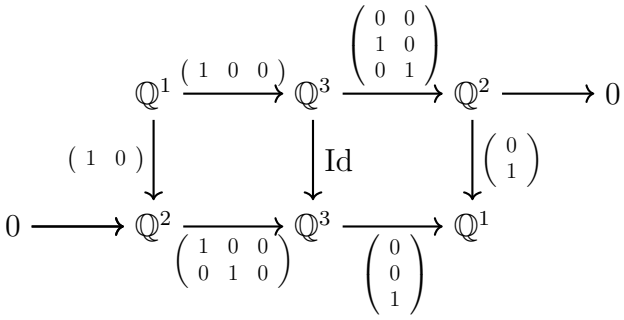
```
gap> InstalledMethodsOfCategory( vecspaces );
Can do the following basic methods at the moment:
* InverseImmutable, weight 201
... <output ommitted> ..
```

One can now also carry out less obvious computations. In the next example we compute the intersection of two 2 dimensional subspaces of a 3 dimensional vector space. Please note that the steps for this computation are completely general, and can also be used for the intersection of arbitrary subobjects, for example ideals or submodules. We present the subspaces as images of two morphisms α and β, compute their FiberProduct and compose its embedding in the source of α with α itself. The resulting morphism represents the intersection of the image of α and β in their range.

```
gap> alpha := VectorSpaceMorphism( V, [ [ 1, 0, 0 ], [ 0, 1, 1 ] ], W );
A rational vector space homomorphism with matrix:
[ [  1,  0,  0 ],
  [  0,  1,  1 ] ]
gap> beta := VectorSpaceMorphism( V, [ [ 1, 1, 0 ], [ 0, 0, 1 ] ], W );
A rational vector space homomorphism with matrix:
[ [  1,  1,  0 ],
  [  0,  0,  1 ] ]
gap> fiberproduct := FiberProduct( alpha, beta );
<A rational vector space of dimension 1>
```

```
gap> projection := ProjectionInFactor( fiberproduct, 1 );
A rational vector space homomorphism with matrix:
[ [  1,  1 ] ]
gap> intersection := PreCompose( projection, alpha );
A rational vector space homomorphism with matrix:
[ [  1,  1,  1 ] ]
```

But the computational powers of CAP do not end at this point. Using the extensions of CAP, like `HomologicalAlgebraForCap`, we can use the given algorithms to compute complicated diagram chases and invariants with very low effort. First, we compute the connecting homomorphism in the Snake lemma applied to the following diagram.



In the example, we are going to name the morphisms in the upper chain α_1 and α_2, in the lower chain β_1 and β_2, and the connecting ones γ_1, γ_2, and γ_3. Please note that for the computation only α_2, β_1 and the connections are needed.

```
gap> LoadPackage( "HomologicalAlgebraForCAP" );
true
gap> V1 := QVectorSpace( 1 );
<A rational vector space of dimension 1>
gap> V2 := QVectorSpace( 2 );
<A rational vector space of dimension 2>
gap> V3 := QVectorSpace( 3 );
<A rational vector space of dimension 3>
gap> alpha2 := VectorSpaceMorphism( V3, [ [ 0, 0 ], [ 1, 0 ], [ 0, 1 ] ], V2 );
A rational vector space homomorphism with matrix:
[ [  0,  0 ],
  [  1,  0 ],
  [  0,  1 ] ]
gap> beta1 := VectorSpaceMorphism( V2, [ [ 1, 0, 0 ], [ 0, 1, 0 ] ], V3 );
A rational vector space homomorphism with matrix:
[ [  1,  0,  0 ],
  [  0,  1,  0 ] ]
gap> gamma1 := VectorSpaceMorphism( V1, [ [ 1, 0 ] ], V2 );
A rational vector space homomorphism with matrix:
[ [  1,  0 ] ]
gap> gamma2 := IdentityMorphism( V3 );
```

```
A rational vector space homomorphism with matrix:
[ [  1,  0,  0 ],
  [  0,  1,  0 ],
  [  0,  0,  1 ] ]
gap> gamma3 := VectorSpaceMorphism( V2, [ [ 0 ], [ 1 ] ], V1 );
A rational vector space homomorphism with matrix:
[ [  0 ],
  [  1 ] ]
gap> snake := SnakeLemmaConnectingHomomorphism( alpha2, gamma1, gamma2,
> gamma3, beta1 );
A rational vector space homomorphism with matrix:
[ [  1 ] ]
```

Another nice feature of CAP are functors. The functors in CAP should represent mathematical functors between categories and are modeled as morphisms in the CAP category of categories, `CapCat`. They provide a good way to communicate between different data structures implemented by someone in the CAP environment. Then they are functors between different categories. Also, it is useful to implement some internal operations in a category as endofunctors, since functors can be manipulated like morphisms, i.e. composed, etc. Also, natural transformations can be implemented in CAP. Generally, functors in CAP need two functions, a function describing the action on the objects, and a function describing the actions on the morphism. Also, we need to tell the system in which component the functor is covariant and in which the functor is contravariant. Covariant is hereby the standard case, so we do not need to specify it. In the next example, we create, as a first example, the identity functor of the previously created category of vector spaces. This is somehow the easiest functor, since it does not require any manipulation at all. We start by creating the functor object, and then adding the object and the morphism function to it.

```
gap> id_functor := CapFunctor( "Identity of vecspaces", vecspaces, vecspaces );
Identity of vecspaces
gap> AddObjectFunction( id_functor, IdFunc );
gap> AddMorphismFunction( id_functor, function( obj1, mor, obj2 )
> return mor; end );
```

Even if this example is trivial, we already see one of the main points to keep in mind when creating functors. Although the functor only takes one argument, the morphism function takes three. This is due to the fact that morphism functions in functors are always kind of `WithGiven` functions (see 2.1). The first argument here is the image of the source(s) of the morphism(s) given to the object function of the functor, the last is the image of the range(s).

It was already mentioned that functors are morphisms in a specific category, and every CAP category has a specific object in this category. Of course, a method for the identity morphism is implemented in `CapCat`, so we can have the same result with less effort.

```
gap> id_functor := IdentityMorphism( AsCatObject( vecspaces ) );
Identity functor of VectorSpaces
```

The identity functor is not of much use, but interesting to show how a functor is created in general. We will now create a second functor. It will map a vector space V to its direct sum with itself, i.e. $V \oplus V$, and provides the diagonal of a morphism. The functor is created as follows.

```
double_functor := CapFunctor( "DoubleOfVecspaces",
                              vecspaces, vecspaces );

AddObjectFunction( double_functor,

  function( obj )

    return QVectorSpace( 2 * Dimension( obj ) );

end );

AddMorphismFunction( double_functor,

  function( new_source, mor, new_range )
    local matr, matr1;

    matr := EntriesOfHomalgMatrixAsListList( mor!.morphism );

    matr := Concatenation( List( matr,
            i -> Concatenation( i,
                 ListWithIdenticalEntries( Length( i ), 0 ) ) ),
                           List( matr,
            i -> Concatenation(
                 ListWithIdenticalEntries( Length( i ), 0 ), i ) ) );

    return VectorSpaceMorphism( new_source, matr, new_range );

end );
```

Since we have given this functor functions about how to act on objects or morphisms, we can actually apply it to such. We can also compose it with itself and then again apply it to objects or morphisms.

```
gap> V2;
<A rational vector space of dimension 2>
gap> ApplyFunctor( double_functor, V2 );
<A rational vector space of dimension 4>
gap> alpha2;
```

```
A rational vector space homomorphism with matrix:
[ [  0,  0 ],
  [  1,  0 ],
  [  0,  1 ] ]
gap> ApplyFunctor( double_functor, alpha2 );
A rational vector space homomorphism with matrix:
[ [  0,  0,  0,  0 ],
  [  1,  0,  0,  0 ],
  [  0,  1,  0,  0 ],
  [  0,  0,  0,  0 ],
  [  0,  0,  1,  0 ],
  [  0,  0,  0,  1 ] ]
```

As said, since functors are morphisms, it is also possible to manipulate functors like morphisms. In the next example, we compose the double functor with itself to get a quadruble functor.

```
gap> quadruple_functor := PreCompose( double_functor, double_functor );
Composition of DoubleOfVecspaces and DoubleOfVecspaces
gap> ApplyFunctor( quadruple_functor, V2 );
<A rational vector space of dimension 8>
gap> ApplyFunctor( quadruple_functor, alpha2 );
A rational vector space homomorphism with matrix:
[ [  0,  0,  0,  0,  0,  0,  0,  0 ],
  [  1,  0,  0,  0,  0,  0,  0,  0 ],
  [  0,  1,  0,  0,  0,  0,  0,  0 ],
  [  0,  0,  0,  0,  0,  0,  0,  0 ],
  [  0,  0,  1,  0,  0,  0,  0,  0 ],
  [  0,  0,  0,  1,  0,  0,  0,  0 ],
  [  0,  0,  0,  0,  0,  0,  0,  0 ],
  [  0,  0,  0,  0,  1,  0,  0,  0 ],
  [  0,  0,  0,  0,  0,  1,  0,  0 ],
  [  0,  0,  0,  0,  0,  0,  0,  0 ],
  [  0,  0,  0,  0,  0,  0,  1,  0 ],
  [  0,  0,  0,  0,  0,  0,  0,  1 ] ]
```

Some functors are more important for applications than others. If some data structure is implemented in the CAP setup, and if some functors are implemented in this setup, one might want them to work like methods, and does not want to use `ApplyFunctor` all the time. This is possible using the `InstallFunctor` command. The example also shows an interesting fact about the caching. Both the functor used with `ApplyFunctor` and the installed version use the same cache. This gives a bit more consistency and might save computation time. Also, it ensures the interchangeability of both ways of applying the functor.

```
gap> InstallFunctor( double_functor, "DoubleFunctor" );
gap> V4 := DoubleFunctor( V2 );
```

```
<A rational vector space of dimension 4>
gap> V4_2 := ApplyFunctor( double_functor, V2 );
<A rational vector space of dimension 4>
gap> IsIdenticalObj( V4, V4_2 );
true
gap> DoubleFunctor( alpha2 );
A rational vector space homomorphism with matrix:
[ [  0,  0,  0,  0 ],
  [  1,  0,  0,  0 ],
  [  0,  1,  0,  0 ],
  [  0,  0,  0,  0 ],
  [  0,  0,  1,  0 ],
  [  0,  0,  0,  1 ] ]
```

At last, we want to show natural transformations in CAP and the manipulation which is possible for them. For this, we create the natural transformation which swaps the components in the double functor, i.e., for some vector space V it creates the morphism $V \oplus V \to V \oplus V$, $(x, y) \mapsto (y, x)$. A natural transformation only needs one function, which returns for an object the correct morphism.

```
double_swap_components := NaturalTransformation( "double swap components",
                              double_functor, double_functor );

AddNaturalTransformationFunction( double_swap_components,

  function( doubled_source, obj, doubled_range )
    local zero_morphism, one_morphism;

    zero_morphism := ZeroMorphism( obj, obj );

    one_morphism := IdentityMorphism( obj );

    return MorphismBetweenDirectSums( [ [ zero_morphism, one_morphism ],
                                        [ one_morphism, zero_morphism ] ] );

end );
```

Of course, we can apply the natural transformation to objects. But it is also possible to compute the vertical and the horizontal composition of natural transformations. In the next and last example we compute both the horizontal and the vertical composition of the natural transformation we created with itself. We leave it to the reader to verify the results.

```
gap> ApplyNaturalTransformation( double_swap_components, V2 );
A rational vector space homomorphism with matrix:
[ [  0,  0,  1,  0 ],
  [  0,  0,  0,  1 ],
```

```
  [  1,  0,  0,  0 ],
  [  0,  1,  0,  0 ] ]
gap> h_composition := HorizontalPreCompose( double_swap_components,
                                                 double_swap_components );
Vertical composition of Horizontal composition of natural
transformation double swap components and functor DoubleOfVecspaces
and Horizontal composition of functor DoubleOfVecspaces
and natural transformation double swap components
gap> ApplyNaturalTransformation( h_composition, V2 );
A rational vector space homomorphism with matrix:
[ [  0,  0,  0,  0,  0,  0,  1,  0 ],
  [  0,  0,  0,  0,  0,  0,  0,  1 ],
  [  0,  0,  0,  0,  1,  0,  0,  0 ],
  [  0,  0,  0,  0,  0,  1,  0,  0 ],
  [  0,  0,  1,  0,  0,  0,  0,  0 ],
  [  0,  0,  0,  1,  0,  0,  0,  0 ],
  [  1,  0,  0,  0,  0,  0,  0,  0 ],
  [  0,  1,  0,  0,  0,  0,  0,  0 ] ]
gap> v_composition := VerticalPreCompose( double_swap_components,
                                                double_swap_components );
Vertical composition of double swap components and double swa
 p components
gap> ApplyNaturalTransformation( v_composition, V2 );
A rational vector space homomorphism with matrix:
[ [  1,  0,  0,  0 ],
  [  0,  1,  0,  0 ],
  [  0,  0,  1,  0 ],
  [  0,  0,  0,  1 ] ]
```

CHAPTER 3

Specifications

In CAP, we want to model categories. In order to do this correctly, we have to implement the basic operations of a category with respect to specifications dictated by CAP. In this chapter, we will see in detail what these specifications are.

This chapter is organized as follows:

In the first section, we define the notion of a category, so that it becomes clear what kind of mathematical object we want to model.

In the second section, we define `GAP` sets and `GAP` maps, in order to have semantics for the basic operation symbols.

Afterwards, we begin with the enumeration of all specifications, which are the equality specifications, typing specifications, and mathematical specifications.

We conclude with a statement about the importance of specifications.

1. Categories

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.

Definition 1.1. A CAP category $\mathbf{C}$ consists of the following data:

(1) A set $\mathrm{Obj}_{\mathbf{C}}$ of **objects**.
(2) For every pair $a, b \in \mathrm{Obj}_{\mathbf{C}}$, a set $\mathrm{Hom}_{\mathbf{C}}(a, b)$ of **morphisms**.
(3) 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**.
(4) For every $a \in \mathrm{Obj}_{\mathbf{C}}$, an **identity morphism** $\mathrm{id}_a \in \mathrm{Hom}_{\mathbf{C}}(a, a)$.
(5) 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 $\alpha, \alpha' \in \mathrm{Hom}_C(a, b)$, $\beta, \beta' \in \mathrm{Hom}_C(b, c)$, $\alpha \sim_{a,b} \alpha'$ and $\beta \sim_{b,c} \beta'$, then $\beta \circ \alpha \sim_{a,c} \beta' \circ \alpha'$.
(6) 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
$$\alpha \sim_{a,b} (\alpha \circ \mathrm{id}_a).$$
(7) 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
$$((\gamma \circ \beta) \circ \alpha) \sim_{a,d} (\gamma \circ (\beta \circ \alpha))$$

So the main differences between a CAP category and a classical category are:

(1) A CAP category has a set of objects, not a class.
(2) A CAP category has as an additional part of its datum a congruence for morphisms, and the axioms are stated with respect to this congruence, and not with respect to equality.

We will see that the congruence for morphisms actually makes the implementation of some categories easier (for example the category of presentations).

REMARK 1.2. Passing to the quotient sets $\mathrm{Hom}_C(a,b)/\sim_{a,b}$ gives rise to a classical category $\mathbf{D}$, because all constructions and axioms respect the congruence. It is usually the case that we actually want to study $\mathbf{D}$, but that it is easier to implement a CAP category $\mathbf{C}$ giving rise to $\mathbf{D}$.

REMARK 1.3. In terms of higher category theory, a CAP category is a 2-category such that the 2-morphism sets are either empty or a singleton.

Convention. Throughout this manual we will use *category* as a short term for a CAP category. If we want to refer to the classical notion of a category (i.e. the one used in [ML71]) we will use the term *classical category*.

2. GAP Sets and GAP Maps

In this section, we will first define GAP sets and GAP maps and then associate to such objects actual sets and maps. Such a translation between objects on the computer and mathematical objects is necessary for modeling mathematics on the computer.

REMARK 2.1. For our definitions to be consistent, we have to fix a set $\mathcal{U}$ containing all GAP objects up to `IsIdenticalObj` which are relevant for the discussion (e.g. within a current session). We call such a $\mathcal{U}$ a **GAP universe**. We further define the set Bool := {`true`, `false`}.

Definition 2.2. A **GAP set** is a pair $(P, =_P)$ consisting of

- a GAP function P
- a binary GAP operation $=_P$

both having values in Bool, such that

(1) P defines a map $\mathcal{U} \to$ Bool,
(2) $=_P$ defines a map $\{(a,b) \in \mathcal{U} \times \mathcal{U} \mid P(a) = P(b) = \texttt{true}\} \to$ Bool,
(3) $=_P$ defines an equivalence relation $\sim_{=_P}$ on $\{a \in \mathcal{U} \mid P(a) = \texttt{true}\}$.

Notation. For a GAP object x such that $P(x) = \texttt{true}$, we simply write $x \in P$.

Definition 2.3. A **GAP map** between GAP sets $(P, =_P)$, $(Q, =_Q)$ is a GAP operation Ω, such that

(1) for $p \in P$, we have $\Omega(p) \in Q$,
(2) for $p =_P p'$, we have $\Omega(p) =_Q \Omega(p')$,
(3) for $y \notin P$, $\Omega(y)$ throws an error.

For a natural number $n \in \mathbb{N}$, we define n-ary GAP maps analogously, i.e., they respect equality component-wise.

REMARK 2.4. [Implementation of a GAP map] Let Ω be a GAP operation installed via the GAP methods $\mu_1, \ldots, \mu_n$. Then Ω defines a GAP map $(P, =_P) \to (Q, =_Q)$ if and only if the following three conditions are satisfied:

(1) $\mu_k(p) =_Q \mu_l(p)$ for $p \in P$ and applicable methods μ_k, μ_l, where $k, l \in \{1 \ldots n\}$.
(2) For every $p \in P$ there exists an $i \in \{1 \ldots n\}$ such that p lies in the filter of μ_i (i.e., the filters of the methods cover P).
(3) $\Omega(y)$ throws an error for $y \notin P$.

The first item can always be technically realized by caching, but note that this is in general not the best strategy. The second item is satisfied in particular if there is a "fallback method" which is applicable for all $p \in P$. The third item will usually be automatically realized by CAP for all basic operations.

In order to model concatenations of maps, we have to define a notion of concatenation of GAP maps.

Definition 2.5. A sequence of GAP maps of length greater then one

$$\Omega_1 : (P_1, =_{P_1}) \to (P_2, =_{P_2}),\ \Omega_2 : (P_2, =_{P_2}) \to (P_3, =_{P_3}), \ldots,$$

is called a **concatenation of GAP maps**.

Example 2.6. If we would rely on weak caching in order to satisfy the first item in remark 2.4, a concatenation of GAP maps would not necessarily yield equal output for equal input. Here is an example of what can go wrong:

Let R be a ring. We take matrices $M \in R^{n \times m}$ for integers n, m as a data structure of modules over R, where the matrix M shall represent the quotient module $R^{1 \times m}/M$. Two modules are considered equal (`IsEqualForObjects`) if their representing matrices are equal.

So for example, if we take $A = (0)$ and $B = (1)$, then A represents a module isomorphic to R, B represents a module isomorphic to the zero module. A canonical way to implement the direct sum of two modules would be to construct a diagonal block matrix, so that applying $A \oplus -$ twice to B yields

$$\texttt{result}_1 := A \oplus (A \oplus B) = A \oplus \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

If $A \oplus -$ works with a weak cache, the result of $A \oplus B$ might get lost, even though we still can access $\texttt{result}_1$. Now assume that the attribute `IsZero` becomes known for B, and assume furthermore that there is a special implementation for $A \oplus -$ which just returns A in that case, we suddenly have

$$\texttt{result}_2 := A \oplus (A \oplus B) = A \oplus A = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix},$$

leaving us with two saved results from syntactically identical expressions having non equal evaluations.

We now associate to every GAP set and every GAP map a corresponding object in set theory.

Definition 2.7. Let $(P, =_P)$ be a GAP set. We define the **associated set** by

$$\overline{P} := \{x \in \mathcal{U} \mid x \in P\}/\sim_{=_P}.$$

The equivalence class of an object $x \in P$ is denoted by $\overline{x}$.

Definition 2.8. Let $\alpha : (P, =_P) \to (Q, =_Q)$ be a GAP map. We define the **associated map** by

$$\overline{\alpha} := \{(\overline{p}, \overline{q}) \mid p \in P, q \in Q, \alpha(p) =_Q q\} \subseteq \overline{P} \times \overline{Q}.$$

The definitions of the associated n-ary maps for $n \in \mathbb{N}$ and for the associated concatenation map are similar.

3. Equality Specifications

Specification. Every basic operation has to yield equal output for given equal input.

Stated in the terminology of section 2: Every basic operation has to be a GAP map.

Realizing a basic operation as a GAP map means that we first have to understand its domain and codomain, which are GAP sets. So let $\mathbf{C}$ be a category object, i.e., a category realized as a GAP object in CAP. Every basic operation of $\mathbf{C}$ takes finitely many arguments as an input and computes one single output. Each input argument has one of the following types:

(1) Int, an integer.
(2) Bool, a Boolean.
(3) $\mathrm{Obj}_{\mathbf{C}}$, an object of $\mathbf{C}$.
(4) $\mathrm{Hom}_{\mathbf{C}}(a, b)$, a morphism with source and range given by objects a and b in $\mathbf{C}$, respectively (this is a dependent type). We write $\mathrm{Mor}_{\mathbf{C}} := \sum_{a,b:\mathrm{Obj}_{\mathbf{C}}} \mathrm{Hom}_{\mathbf{C}}(a, b)$ for the type of all morphisms.
(5) $\mathrm{ListObj}_{\mathbf{C}}$, a finite list of objects.
(6) $\mathrm{ListMor}_{\mathbf{C}}$, a finite list of morphisms which possibly have different sources or ranges.

These are also the possible output types except that an output never is a list.
We now specify what it means for two terms of the same type to be equal.

(1) For Int and Bool, equality is given by the equality operations in GAP.
(2) For $\mathrm{ListObj}_{\mathbf{C}}$ and $\mathrm{ListMor}_{\mathbf{C}}$, equality is given by entry-wise equality.
(3) For $\mathrm{Obj}_{\mathbf{C}}$, equality is given by the basic operation `IsEqualForObjects`.
(4) For $\mathrm{Hom}_{\mathbf{C}}(a, b)$, equality is given by the basic operation `IsEqualForMorphisms`.

Like every basic operation, `IsEqualForObjects` and `IsEqualForMorphisms` can be added to a category object using the corresponding add functions `AddIsEqualForObjects` and `AddIsEqualForMorphisms`.

REMARK 3.1. The basic operations `IsEqualForObjects` and `IsEqualForMorphisms` play the role of the equality functions of the sets $\mathrm{Obj}_{\mathbf{C}}$ and $\mathrm{Hom}_{\mathbf{C}}(a, b)$ in definition 1.1. In particular, `IsEqualForMorphisms` shall not be confused with congruence for morphisms.

REMARK 3.2. The above listed types are differently realized in CAP.

- For Int, we use the realization of `GAP`.
- For Bool, we use the realization of `GAP`, but only the values `true` and `false` are allowed (exception: equality functions, see 3.1 and 3.2).
- For $\mathrm{Obj}_{\mathbf{C}}$, there is a `GAP` filter `ObjectFilter`($\mathbf{C}$).
- The type Mor is realized by `GAP` objects within the filter `MorphismFilter`($\mathbf{C}$). The dependent type $\mathrm{Hom}_{\mathbf{C}}(a, b)$ is realized by objects of type Mor having the objects a, b set for their attributes `Source` and `Range`.
- There are no special `GAP` filters for $\mathrm{ListObj}_{\mathbf{C}}$ and $\mathrm{ListMor}_{\mathbf{C}}$, simply the `GAP` filter `IsList` is used.

3.1. Specifications for `IsEqualForObjects`. The basic operation `IsEqualForObjects` has a special specification.

Specification (`IsEqualForObjects`)**.**

(1) Input: two objects (Obj).
(2) Output: a Boolean (Bool).
(3) `IsEqualForObjects` has to respect the `GAP` function `IsIdenticalObj`, i.e., for objects a, a', b, b' such that `IsIdenticalObj`(a, a') and `IsIdenticalObj`(b, b') holds, `IsEqualForObjects`(a, b) and `IsEqualForObjects`(a', b') yield equal Booleans.
(4) `IsEqualForObjects` has to give rise to an equivalence relation on the set of all objects together with `IsEqualForObjects` as an equality function.

In short: The pair (Obj, `IsEqualForObjects`) has to become a `GAP` set, see section 2.

REMARK 3.3. It is allowed for `IsEqualForObjects` to return `fail`. This is interpreted by CAP as *"I don't know if the two objects are equal"*.

3.2. Specifications for `IsEqualForMorphisms`. The basic operation `IsEqualForMorphisms` has a special specification.

Specification (`IsEqualForMorphisms`)**.** Let a, b : Obj.

(1) Input: two morphisms ($\mathrm{Hom}(a, b)$) (having equal sources and ranges).
(2) Output: a Boolean (Bool).
(3) `IsEqualForMorphisms` has to respect the `GAP` function `IsIdenticalObj`, i.e., for morphisms $\alpha, \alpha', \beta, \beta'$ such that `IsIdenticalObj`(α, α') and `IsIdenticalObj`(β, β') holds, `IsEqualForMorphisms`(α, β) and `IsEqualForMorphisms`(α', β') yield equal Booleans.
(4) `IsEqualForMorphisms` has to give rise to an equivalence relation on the set of all morphisms with prescribed source and range together with `IsEqualForObjects` as an equality function.

The pair ($\mathrm{Hom}(a, b)$, `IsEqualForMorphisms`) has to become a `GAP` set (assuming we restrict `IsEqualForMorphisms` to $\mathrm{Hom}(a, b)$), see section 2.

REMARK 3.4. It is allowed for `IsEqualForMorphisms` to return `fail`. This is interpreted by CAP as *"I don't know if the two morphisms are equal"*.

4. Typing Specifications

Specification. Every basic operation has to match its type.

Every basic operation symbol has a type. To be more precise, it has a (dependent) function type. We refer to the first chapter of [**Uni13**] for a good summary of dependent type theory.

Example 4.1. The type of IsZeroForObjects is

$$\text{Obj} \to \text{Bool}.$$

Given a term of type Obj, the basic operation symbol IsZeroForObjects returns a term of type Bool.

Example 4.2. For $a, b : \text{Obj}$, the type of KernelObject is

$$\text{Hom}(a, b) \to \text{Obj}.$$

Given a term of type $\text{Hom}(a, b)$, the basic operation symbol KernelObject returns a term of type Obj.

Example 4.3. For $a, b : \text{Obj}$, the type of KernelEmbedding is

$$\prod_{\phi:\text{Hom}(a,b)} \text{Hom}(\text{KernelObject}(\phi), a).$$

Given a term of type $\text{Hom}(a, b)$, the basic operation symbol KernelEmbedding returns a term of type $\text{Hom}(\text{KernelObject}(\phi), a)$. Thus KernelEmbedding has a dependent function type, because the type of its output depends on the input term.

For a basic operation to match its typing specification, it has to

(1) accept only input specified by its type, and throw an error otherwise,
(2) compute only output specified by its type.

The first item is (almost) always handled by CAP. If we add a basic operation to CAP via an add function, CAP ensures that every call of that basic operation first triggers a type checking for the input.
The meaning and implications of the second item have to be analyzed carefully.

Example 4.4 (4.1 continued)**.** The output has to be a Boolean (c.f. remark 3.2).

Example 4.5 (4.2 continued)**.** The output has to be an object (c.f. remark 3.2).

Example 4.6 (4.3)**.** The output type depends on the given input. Let a denote the source, b denote the range of the input morphism ϕ. Then the range of the output morphism has to be equal (`IsEqualForObjects`) to a. The source of the output morphism has to be equal (`IsEqualForObjects`) to the output of $\text{KernelObject}(\phi)$.

As seen in example 4.3, basic operation symbols (like KernelObject) can be part of the type of other basic operations (like $\prod_{\phi:\mathrm{Hom}(a,b)} \mathrm{Hom}(\mathrm{KernelObject}(\phi), a)$). To implement a coherent model of all basic operations, we have to take care that such dependencies are fulfilled. Fortunately, CAP provides tools to help us with this task.

Example 4.7. Strategies of an implementation of KernelObject and KernelEmbedding with the correct types:

(1) Only implement KernelEmbedding. CAP will automatically derive (c.f. section 4) the basic operation KernelObject coherently for you (as the source of KernelEmbedding).
(2) Implement KernelObject and the helper basic operation KernelEmbeddingWithGivenKernelObject This is an operation getting as an input a morphism ϕ and the result of KernelObject(ϕ), which we can then use to construct a morphism with KernelObject(ϕ) as its source (c.f. 2.1). CAP will automatically derive (c.f. section 4) the basic operation KernelEmbedding coherently for you (by calling KernelEmbeddingWithGivenKernelObject with the output of KernelObject as an input).
(3) Implement KernelObject, KernelEmbedding, and KernelEmbeddingWithGivenKernelObject, and enable caching for these functions (c.f. chapter 7). There is a redirection mechanism (see 2.1) of CAP that guarantees a correct typing.

Of course, you always can only implement KernelObject and KernelEmbedding, and prove the correct typing manually.

5. Mathematical Specifications

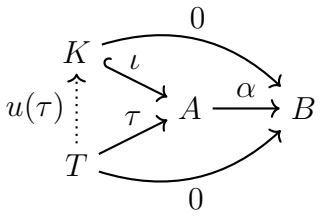
Specification. Every basic operation has to compute what it is mathematically supposed to compute.

In order to implement a coherent model of a category, the basic operations that we provide have to match all mathematical specifications. We give an example to illustrate what this means.

Example 5.1. We want to implement KernelObject, KernelEmbedding, and KernelLift. Let us take a look at the definition of a kernel in a category $\mathbf{C}$.
For a given morphism $\alpha : A \to B$ in $\mathbf{C}$, a kernel of α consists of three parts:

(1) an object $K \in \mathbf{C}$,
(2) a morphism $\iota : K \to A$ such that $\alpha \circ \iota \sim_{K,B} 0$,
(3) 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. The situation can be depicted as follows:



We call $K, \iota, u(\tau)$ a KernelObject, KernelEmbedding, KernelLift, respectively. Thus for implementing these basic operations properly, they have to yield outputs which together form a kernel of ϕ.

Remark 5.2. We note that the universal property of the kernel is formulated with the notion of congruence for morphisms, not with equality of morphisms. This is again due to our notion of a category, see section 1.

6. The Importance of Specifications

Once we have implemented a category matching all specifications, we can benefit from all the constructions Cap automatically performs for us. Cap is able to use categories in order to construct new ones. It is also able to derive algorithms using the basic operations as building blocks. The correctness of all these automatic constructions heavily depends on the correctness of its building blocks.

(1) Equality specifications: necessary to interpret basic operations in set theory. Not matching these specifications means dealing with randomness.
(2) Typing specifications: necessary to safely concatenate basic operations.
(3) Mathematical specifications: necessary because we want to model our mathematical notions correctly.

CHAPTER 4

Constructive Category Theory

In this chapter, we explain the philosophy of constructive category theory underlying our software project CAP. In short: Existential quantifiers have to be turned into algorithms. Using dependent type theory [Uni13], this short statement can be made precise.

1. Propositions as Types

Like ZFC, dependent type theory is a formal language in which we can do mathematics. Every correct mathematical theorem stated in "everyday language" can in principle be "compiled" to a formal statement in dependent type theory, where its correctness can then be verified formally. One of the nice features of dependent type theory is called *proposition as types*. Unlike ZFC, dependent type theory makes no distinction between a proposition and the objects which we want to describe (e.g. sets or types). So we shall think about a proposition like

$$\exists x \in \mathbb{Z} : x^2 = 4$$

as being the type (or for simplicity the set) consisting of all pairs (x, p) where x is a solution of $x^2 = 4$, and p is a witness of this fact. This is why in dependent type theory, such a proposition is written as a dependent sum (or for simplicity a disjoint union)

$$\sum_{x:\mathbb{Z}} x^2 = 4.$$

Proving a proposition P in dependent type theory means exhibiting a term of P. So for proving $\sum_{x:\mathbb{Z}} x^2 = 4$, we have to give an explicit $x : \mathbb{Z}$, e.g. 2 or -2, and have to show that $x^2 = 4$ (by unwrapping the definition of squaring an integer).

2. Models on the Computer

A proof of $\sum_{x:\mathbb{Z}} x^2 = 4$ with a proof assistant uses the constructions that can be done within dependent type theory. This means that once a proof is found, this proof is valid for every model of dependent type theory (there are models interpreting types as sets or groupoids [HS96]). Such models can be realized on the computer. For example, in the formalization of CAP, we use `GAP`-sets and `GAP`-maps. A `GAP`-map essentially is a computer program taking only elements of its source as its arguments and returns a uniquely determined element of its range, respecting a notion of equality of elements. This is exactly what an algorithm is supposed to do.

3. Examples of Constructions in Category Theory

Combining dependent type theory and computer models, we get to the following concept of constructive category theory:

(1) We use dependent type theory for our formulation of category theory.
(2) We use a computer model of dependent type theory to interpret category theory.

A definition of categories within dependent type theory can be found in [Uni13]. We give some examples of the consequences of this concept of constructiveness.

Example 3.1 (Identities)**.** For every object a in a category $\mathbf{C}$, there exists a morphism id_a such that for all objects b and all morphisms $\alpha \in \mathrm{Hom}(a,b)$, $\beta \in \mathrm{Hom}(b,a)$, the equations $\mathrm{id}_a \circ \beta = \beta$ and $\alpha \circ \mathrm{id}_a = \alpha$ hold. Written in the syntax of dependent type theory, this statement looks as follows:

$$\prod_{a:\mathrm{Obj}} \sum_{\mathrm{id}_a:\mathrm{Hom}(a,a)} \prod_{\substack{b:\mathrm{Obj} \\ \alpha:\mathrm{Hom}(a,b) \\ \beta:\mathrm{Hom}(b,a)}} (\mathrm{id}_a \circ \beta = \beta) \times (\alpha \circ \mathrm{id}_b = \alpha).$$

A term of this type is equal to a dependent function f, mapping an object a to a pair (id_a, p), where id_a is a morphism and p is a proof that this morphism acts like a unit. The proof p can be omitted in our computer model because it is a mere proposition, so we end up with a dependent function $a \mapsto \mathrm{id}_a$, which is given in our computer model by an explicit algorithm.

Example 3.2 (Kernels)**.** Universal objects such as kernels are not merely objects. They are a collection of data. Given a morphism $\phi \in \mathrm{Hom}(a,b)$, a kernel of ϕ consists of

- an object k,
- a morphism $\iota : k \to a$ such that $\phi \circ \iota = 0$,
- a universal property, namely: For every other object t and morphism $\tau : t \to a$ such that $\phi \circ \tau = 0$, there exists a unique morphism $l : t \to k$ such that $\tau = \iota \circ l$.

Written in the syntax of dependent type theory, the type of kernels of ϕ looks as follows:

$$\mathrm{KernelsOf}(\phi) :\equiv \sum_{k:\mathrm{Obj}} \sum_{\iota:\mathrm{Hom}(k,a)} (\phi \circ \iota = 0) \times \left(\prod_{\substack{t:\mathrm{Obj} \\ \tau:\mathrm{Hom}(t,a) \\ p:\phi\circ\tau=0}} \sum_{l:\mathrm{Hom}(t,k)} (\tau = \iota \circ l) \times \mathrm{Uniqueness}(l) \right)$$

A term of this type is equal to a tupel (k, ι, q, u) consisting of an object k, a morphism $\iota : k \to a$, a witness q of $\phi \circ \iota = 0$, and a dependent function u. This dependent function takes as arguments an object t, a morphism $\tau : t \to a$, and a witness p of $\phi \circ \tau = 0$. The output is a triple consisting of morphism $l : t \to k$, a witness of $\tau = \iota \circ l$, and a witness of the uniqueness of l. Again by omitting mere propositions in our computer model, u is interpreted by an explicit algorithm $(t, \tau) \mapsto l$.
We say that a category has kernels if for all objects a, b and morphisms $\phi \in \mathrm{Hom}(a,b)$,

ϕ has a kernel. Written in the syntax of dependent type theory, this statement looks as follows:

$$\prod_{\substack{a,b:\mathrm{Obj} \\ \phi:\mathrm{Hom}(a,b)}} \mathrm{KernelsOf}(\phi).$$

A term of this type is equal to a dependent function mapping a triple (a, b, ϕ) to a term of $\mathrm{KernelsOf}(\phi)$. Interpreted in our computer model, such a dependent function is an algorithm $(a, b, \phi) \mapsto (k, \iota, q, u)$. Note that u (the universal property) is an algorithm by itself, here constructed by another algorithm.

In the following examples we omit the descriptions in dependent type theory and focus on their interpretations in a computer model.

Example 3.3 (Functors)**.** A functor F between two categories $\mathbf{C}$ and $\mathbf{D}$ consists of

- a function $F_0 : \mathrm{Obj}_{\mathbf{C}} \to \mathrm{Obj}_{\mathbf{D}}$,
- for $a, b \in \mathrm{Obj}_{\mathbf{C}}$, a function $F_{a,b} : \mathrm{Hom}_{\mathbf{C}}(a, b) \to \mathrm{Hom}_{\mathbf{D}}(F_0(a), F_0(b))$

such that

- $F_{a,a}(\mathrm{id}_a) = \mathrm{id}_{F_0(a)}$,
- for $a, b, c \in \mathrm{Obj}_{\mathbf{C}}$, $\alpha : a \to b$, $\beta : b \to c$, we have $F_{a,c}(\beta \circ \alpha) = F_{b,c}(\beta) \circ F_{a,b}(\alpha)$.

Interpreting this in a computer model, the action of the functor F is given by two algorithms:

- On objects: The argument is an object $a \in \mathbf{C}$. The output is an object $F_0(a) \in \mathbf{D}$.
- On morphisms: The arguments are objects a, b and a morphism $\phi : a \to b$. The output is a morphism $F_{a,b}(\phi) : F_0(a) \to F_0(b)$.

Example 3.4 (Natural Transformations)**.** A natural transformation ν between two functors $F, G : \mathbf{C} \to \mathbf{D}$ consists of

- a dependent function mapping an object $a \in \mathbf{C}$ to a morphism $\nu_a \in \mathrm{Hom}_{\mathbf{D}}(Fa, Ga)$

such that

- for $b \in \mathbf{C}$, $\alpha : a \to b$, there is an equality $G(\alpha) \circ \nu_a = \nu_b \circ F(\alpha)$.

Interpreting this in a computer model, a natural transformation is an algorithm having an object a as an argument. The output is a morphism $\nu_a : F(a) \to G(a)$.

Example 3.5 (Tensor Hom Adjunction)**.** One of the axioms of a closed monoidal category $\mathbf{C}$ (e.g. the category of modules over a commutative ring R) says that

- for each $b \in \mathbf{C}$, the functor $- \otimes b$ has a right adjoint $\underline{\mathrm{Hom}}(b, -)$.

To understand this axiom constructively, we have to unwrap it. So $\underline{\mathrm{Hom}}(b, -)$ being a right adjoint to $- \otimes b$ means that

- $\exists$ a natural transformation $\mathrm{coev}^b_a : a \to \underline{\mathrm{Hom}}(b, a \otimes b)$, called **coevaluation**,
- $\exists$ a natural transformation $\mathrm{ev}^b_a : \underline{\mathrm{Hom}}(b, a) \otimes b \to a$, called **evaluation**,

such that the so called zig-zag relations[1] hold. Interpreting this axiom in a computer model, it says that we have two algorithms:

[1]The zig-zag relations are $\mathrm{ev}^b_{a\otimes b} \circ (\mathrm{coev}^b_a \otimes b) = \mathrm{id}_{a\otimes b}$ and $\underline{\mathrm{Hom}}(b, \mathrm{ev}^b_a) \circ \mathrm{coev}^b_{\underline{\mathrm{Hom}}(b,a)} = \mathrm{id}_{\underline{\mathrm{Hom}}(b,a)}$.

(1) The arguments are objects b, a. The output is a morphism $a \to \underline{\mathrm{Hom}}(b, a \otimes b)$.
(2) The arguments are objects b, a. The output is a morphism $\underline{\mathrm{Hom}}(b, a) \otimes b \to a$.

Having these morphisms interpreted in our computer model, we are able to construct all the natural morphisms on the computer that can be constructed formally on the level of dependent type theory. For example, let a be an object, and denote by $a^\vee = \underline{\mathrm{Hom}}(a, 1)$ its dual. We want to construct the natural map $a \to (a^\vee)^\vee$. For this, we

(1) construct $\mathrm{coev}_a^{(a^\vee)} : a \to \underline{\mathrm{Hom}}(a^\vee, a \otimes a^\vee)$,
(2) construct $\underline{\mathrm{Hom}}(a^\vee, B_{a,a^\vee}) : \underline{\mathrm{Hom}}(a^\vee, a \otimes a^\vee) \to \underline{\mathrm{Hom}}(a^\vee, a^\vee \otimes a)$,
(3) construct $\mathrm{ev}_1^a : a^\vee \otimes a \to 1$,
(4) construct $\underline{\mathrm{Hom}}(a^\vee, \mathrm{ev}_1^a) : \underline{\mathrm{Hom}}(a^\vee, a^\vee \otimes a) \to (a^\vee)^\vee$.

The desired morphism is given by the composition of (1), (2), and (4). Note that this is an example of a derivation implemented in Cap. In general, using the evaluation and the coevaluation, we can explicitly construct the bijection

$$\mathrm{Hom}(a \otimes b, c) \cong \mathrm{Hom}(a, \underline{\mathrm{Hom}}(b, c)),$$

and use this in turn for constructing natural morphisms.

CHAPTER 5

Categorical Constructions

CAP supports categorical constructions for a category **C** of the following types:

- Basic constructions which only depend on the fact that **C** is category, such as composition (PreCompose), identity (IdentityMorphism), decisions whether a morphism is a mono, epi, or iso (IsMonomorphism, IsEpimorphism, IsIsomorphism).
- Some limit and colimit constructions, such as direct product (DirectProduct), coproduct (Coproduct), fiber product (FiberProduct), pushout (Pushout), kernel (KernelObject), cokernel (CokernelObject).
- Constructions for monoidal categories, such as tensor product (TensorProductOnObjects, TensorProductOnMorphisms), unitors (LeftUnitor, RightUnitor), associators (AssociatorLeftToRight, AssociatorRightToLeft), internal hom (InternalHomOnObjects, InternalHomOnMorphisms).

Notation. For the definition of a category, see chapter 3, section 1.

1. Basic Constructions

1.1. Basic Categorical Constructions. The most basic categorical constructions are composition and the creation of identity morphisms, c.f. the definition of a CAP category in section 1.

1.2. Basic Categorical Properties of Morphisms. In CAP, support is given for the implementation of the following categorical properties of morphisms.

Definition 1.1. A morphism $\alpha : b \to c$ is called a **monomorphism** if for every pair of morphisms $\beta, \gamma : a \to b$ such that $\alpha \circ \beta \sim_{a,c} \alpha \circ \gamma$, we have $\beta \sim_{a,b} \gamma$.

Definition 1.2. A morphism $\alpha : a \to b$ is called an **epimorphism** if for every pair of morphisms $\beta, \gamma : b \to c$ such that $\beta \circ \alpha \sim_{a,c} \gamma \circ \alpha$, we have $\beta \sim_{b,c} \gamma$.

Definition 1.3. A morphism $\alpha : a \to b$ is called an **isomorphism** if there is a morphism $\beta : b \to a$ such that $\alpha \circ \beta \sim_{b,b} \mathrm{id}_b$ and $\beta \circ \alpha \sim_{a,a} \mathrm{id}_a$.

Definition 1.4. A morphism $\alpha : a \to b$ is called a **split monomorphism** if there is a morphism $\beta : b \to a$ such that $\beta \circ \alpha \sim_{a,a} \mathrm{id}_a$.

Definition 1.5. A morphism $\alpha : a \to b$ is called a **split epimorphism** if there is a morphism $\beta : b \to a$ such that $\alpha \circ \beta \sim_{b,b} \mathrm{id}_b$.

2. Limits and Colimits

In CAP, support is given for the implementation of some special kinds of limits and colimits. We first give the definition of a limit and a colimit. Let $\mathbf{C}$ be a category. Let $\mathbf{I}$ be another category (called an **index category**) and $D : \mathbf{I} \to \mathbf{C}$ be a functor (called a **diagram**).

Definition 2.1. A **source** of D is a collection of morphisms $(s_i : S \to D_i)_{i \in \mathbf{I}}$ such that $(D(i \to j) \circ s_i) \sim_{S,D_j} s_j$ for every arrow $(i \to j) \in \mathbf{I}$.

Definition 2.2. A triple (L, λ, u) consisting of the following data:

(1) an object $L \in \mathbf{C}$,
(2) a source $\lambda = (\lambda_i : L \to D_i)_{i \in \mathbf{I}}$,
(3) a dependent function u mapping every source $\tau = (\tau_i : T \to D_i)_{i \in \mathbf{I}}$ to a morphism $u(\tau) : T \to L$ such that $\lambda_i \circ u(\tau) \sim_{T,D_i} \tau_i$,

is called a **limit of the diagram** D, if the morphisms $u(\tau)$ are uniquely determined up to congruence of morphisms.

Definition 2.3. A **sink** of D is a collection of morphisms $(s_i : D_i \to S)_{i \in \mathbf{I}}$ such that $(s_i \circ D(i \to j)) \sim_{D_i,S} s_j$ for every arrow $(i \to j) \in \mathbf{I}$.

Definition 2.4. A triple (C, c, u) consisting of the following data:

(1) an object $C \in \mathbf{C}$,
(2) a sink $c = (c_i : D_i \to C)_{i \in \mathbf{I}}$,
(3) a dependent function u mapping every source $\tau = (\tau_i : D_i \to T)_{i \in \mathbf{I}}$ to a morphism $u(\tau) : C \to T$ such that $u(\tau) \circ c_i \sim_{D_i,T} \tau_i$,

is called a **colimit of the diagram** D, if the morphisms $u(\tau)$ are uniquely determined up to congruence of morphisms.

Example 2.5. Examples of limits are kernels, direct products, fiber products.

Example 2.6. Examples of colimits are cokernels, coproducts, pushouts.

2.1. Limits in Cap. For every limit $(L, \lambda = (\lambda_i : L \to D_i)_{i \in \mathbf{I}})$ in CAP, there are three basic operations.

(1) A basic operation returning the limit object. The argument is the diagram:
$$D \mapsto L.$$
(2) A basic operation returning the limit source. The argument is the diagram:
$$D \mapsto \lambda.$$
(3) A basic operation returning the morphism given by the universal property. The arguments are the diagram and a test source:
$$(D, \tau) \mapsto u(\tau).$$

Example 2.7. In the case of a kernel, the three basic operations in question are

(1) KernelObject,

(2) KernelEmbedding,
(3) KernelLift.

In addition, there are two more basic operations which are variants of the second and the third basic operation.

(1) A basic operation returning the limit source. The arguments are the diagram and an object equal (IsEqualForObjects) to the limit object:

$$(D, L) \mapsto \lambda.$$

(2) A basic operation returning the morphism given by the universal property. The arguments are the diagram, a test source, and an object equal (IsEqualForObjects) to the limit object:

$$(D, \tau, L) \mapsto u(\tau).$$

Example 2.8. In the case of a kernel, the two variants in question are

(1) KernelEmbeddingWithGivenKernelObject,
(2) KernelLiftWithGivenKernelObject.

We call such basic operations `WithGiven` operations. We recommend not to call the `WithGiven` operations but only to implement them as helpers in the following sense: Depending on which of the above five basic operations are implemented, CAP automatically tries to fill in the missing operations:

- Given $D \mapsto L$ and $(D, L) \mapsto \lambda$, CAP derives $D \mapsto \lambda$.
- Given $D \mapsto L$ and $(D, \tau, L) \mapsto u(\tau)$, CAP derives $(D, \tau) \mapsto u(\tau)$.

Moreover, if the three basic operations $D \mapsto L$, $D \mapsto \lambda$, and $(D, L) \mapsto \lambda$ are all given, then CAP automatically enhances $D \mapsto \lambda$ with a **redirect function**. The redirect function first looks if the limit object L is already in the cache of $D \mapsto L$ (compared with a user given equality function, default option is IsEqualForObjects). If this is the case, CAP automatically calls the `WithGiven` operation $(D, L) \mapsto \lambda$ with the input L found in the cache.

Analogously, CAP enhances $(D, \tau) \mapsto u(\tau)$ with a redirect function if the three basic operations $D \mapsto L, (D, \tau) \mapsto u(\tau), (D, \tau, L) \mapsto u(\tau)$ are all given.

We will now define the limits and colimits available in CAP.

2.2. Direct Product.

Definition 2.9. For a given list of objects $D = (P_1, \ldots, P_n)$, a direct product of D consists of three parts:

(1) an object P,
(2) a list of morphisms $\pi = (\pi_i : P \to P_i)_{i=1\ldots n}$
(3) 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.

2.3. Coproduct.

Definition 2.10. For a given list of objects $D = (I_1, \ldots, I_n)$, a coproduct of D consists of three parts:

(1) an object I,
(2) a list of morphisms $\iota = (\iota_i : I_i \to I)_{i=1\ldots n}$
(3) 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 $i = 1, \ldots, n$.

The triple (I, ι, u) is called a **coproduct of** D if the morphisms $u(\tau)$ are uniquely determined up to congruence of morphisms.

2.4. Fiber Product.

Definition 2.11. For a given list of morphisms $D = (\beta_i : P_i \to B)_{i=1\ldots n}$, a fiber product of D consists of three parts:

(1) an object P,
(2) a list of morphisms $\pi = (\pi_i : P \to P_i)_{i=1\ldots n}$ such that $\beta_i \circ \pi_i \sim_{P,B} \beta_j \circ \pi_j$ for all pairs i, j.
(3) 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 $i = 1, \ldots, n$.

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.

2.5. Pushout.

Definition 2.12. For a given list of morphisms $D = (\beta_i : B \to I_i)_{i=1\ldots n}$, a pushout of D consists of three parts:

(1) an object I,
(2) a list of morphisms $\iota = (\iota_i : I_i \to I)_{i=1\ldots n}$ such that $\iota_i \circ \beta_i \sim_{B,I} \iota_j \circ \beta_j$ for all pairs i, j,
(3) a dependent function u mapping each list of morphisms $\tau = (\tau_i : I_i \to T)_{i=1\ldots n}$ 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 $i = 1, \ldots, n$.

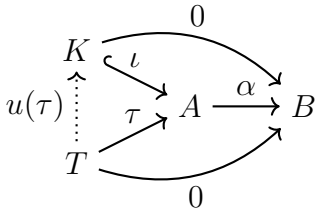
The triple (I, ι, u) is called a **pushout of** D if the morphisms $u(\tau)$ are uniquely determined up to congruence of morphisms.

2.6. Kernel.

Definition 2.13. For a given morphism $\alpha : A \to B$, a kernel of α consists of three parts:

(1) an object K,
(2) a morphism $\iota : K \to A$ such that $\alpha \circ \iota \sim_{K,B} 0$,
(3) 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. The situation can be depicted as follows:



2.7. CokernelObject.

Definition 2.14. For a given morphism $\alpha : A \to B$, a cokernel of α consists of three parts:

(1) an object K,
(2) a morphism $\epsilon : B \to K$ such that $\epsilon \circ \alpha \sim_{A,K} 0$,
(3) 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 $u(\tau) \circ \epsilon \sim_{B,T} \tau$.

The triple (K, ϵ, u) is called a **cokernel of** α if the morphisms $u(\tau)$ are uniquely determined up to congruence of morphisms.

2.8. Terminal Object.

Definition 2.15. A terminal object consists of two parts:

(1) an object T,
(2) 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.

Remark 2.16. The corresponding diagram D is given by the unique functor $\to \mathbf{A}$. Thus the source of a terminal object can be omitted.

2.9. Initial Object.

Definition 2.17. An initial object consists of two parts:

(1) an object I,
(2) 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.

2.10. Zero Object. Roughly speaking, a zero object is both an initial and a terminal object. Of course, the constructive definition of a zero object has to include algorithms (or witnesses) of being initial and being terminal.

Definition 2.18. A zero object consists of three parts:

(1) an object Z,
(2) a function u_{in} mapping each object A to a morphism $u_{\text{in}}(A) : A \to Z$,

(3) a function u_{out} mapping each object A to a morphism $u_{\text{out}}(A) : Z \to A$.

The triple $(Z, u_{\text{in}}, u_{\text{out}})$ is called a **zero object** if the morphisms $u_{\text{in}}(A)$, $u_{\text{out}}(A)$ are uniquely determined up to congruence of morphisms.

2.11. Direct Sum. Roughly speaking, a direct sum is an object in an additive category which is both a direct product and a coproduct such that the injections and projections are compatible. We now state the constructive definition.

Definition 2.19. For a given list $D = (S_1, \ldots, S_n)$, a direct sum consists of five parts:

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

such that

(1) $\sum_{i=1}^{n} \iota_i \circ \pi_i = \mathrm{id}_S$,
(2) $\pi_j \circ \iota_i = \delta_{i,j}$,

where $\delta_{i,j} \in \mathrm{Hom}(S_i, S_j)$ is the identity if $i = j$, and 0 otherwise. The 5-tuple $(S, \pi, \iota, u_{\text{in}}, u_{\text{out}})$ is called a **direct sum of** D if the morphisms $u_{\text{in}}(\tau)$, $u_{\text{out}}(\tau)$ are uniquely determined up to congruence of morphisms.

Remark 2.20. Given a triple (S, π, ι) satisfying the conditions of definition 2.19, we can construct u_{in}, u_{out} in unique way up to congruence of morphisms such that $(S, \pi, \iota, u_{\text{in}}, u_{\text{out}})$ is a direct sum.

2.12. Image.

Definition 2.21. For a given morphism $\alpha : A \to B$, an image of α consists of four parts:

(1) an object I,
(2) a morphism $c : A \to I$,
(3) a monomorphism $\iota : I \hookrightarrow B$ such that $\iota \circ c \sim_{A,B} \alpha$,
(4) 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.

2.13. Coimage.

Definition 2.22. For a given morphism $\alpha : A \to B$, a coimage of α consists of four parts:

(1) an object C,

(2) an epimorphism $\pi : A \twoheadrightarrow C$,
(3) a morphism $a : C \to B$ such that $a \circ \pi \sim_{A,B} \alpha$,
(4) 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.

3. Categories with more Structure

CAP supports various notions of categories with additional structure. If we want to implement such categories, we set the corresponding `GAP` properties to true right after the call of the constructor `CreateCapCategory`.

Example 3.1. Let k be a field. The category of finite dimensional vector spaces over k is an abelian category. It is realized in the CAP package `LinearAlgebraForCAP`. Furthermore, it can be enhanced with a tensor product such that it becomes a rigid symmetric closed monoidal category.

Example 3.2. Let R be a ring. The category of finitely presented modules over R is an abelian category. It is realized in the CAP package `ModulePresentationsForCAP`. Furthermore, if R is commutative, it can be enhanced with a tensor product such that it becomes a symmetric closed monoidal category.

Example 3.3. The generalized morphism category is enriched over commutative regular semigroups. It is realized in the CAP package `GeneralizedMorphismsForCAP`.

3.1. Categories Enriched over Commutative Regular Semigroups.

Documentation 3.4. The corresponding `GAP` property is given by

`IsEnrichedOverCommutativeRegularSemigroup`.

Definition 3.5. The definition of a category enriched over commutative regular semigroups can be found in [BLH14] (where this property is called *enriched over commutative inverse monoids)*.

3.2. Ab-Categories.

Documentation 3.6. The corresponding `GAP` property is given by

`IsAbCategory`.

Definition 3.7 ([ML71])**.** A category **C** enriched over abelian groups is called an **Ab-category**, i.e., every set of homomorphisms is equipped with a structure of an abelian group such that composition is bilinear.

Basic operations for **Ab**-categories coming from the existential quantifiers in the definition:

- `ZeroMorphism`
- `IsZeroForMorphisms`

- `AdditionForMorphisms`
- `AdditiveInverseForMorphisms`

3.3. Additive Categories.

Documentation 3.8. The corresponding `GAP` property is given by

`IsAdditiveCategory`.

Definition 3.9 ([ML71]). An **Ab**-category **C** is called **additive** if it has a zero object and direct sums for all pairs of objects.

Basic operations for additive categories coming from the existential quantifiers in the definition (in addition to the basic operations for **Ab**-categories):

- `ZeroObject`
- `UniversalMorphismFromZeroObject`
- `UniversalMorphismIntoZeroObject`
- `DirectSum`
- `ProjectionInFactorOfDirectSum`
- `InjectionOfCofactorOfDirectSum`
- `UniversalMorphismIntoDirectSum`
- `UniversalMorphismFromDirectSum`

3.4. Pre-abelian Categories.

Documentation 3.10. The corresponding `GAP` property is given by

`IsPreAbelianCategory`.

Definition 3.11. An additive category **C** is called **pre-abelian** if it has kernels and cokernels for every morphism.

Basic operations for preadditive categories coming from the existential quantifiers in the definition (in addition to the basic operations for additive categories):

- `KernelObject`
- `KernelEmbedding`
- `KernelLift`
- `CokernelObject`
- `CokernelProjection`
- `CokernelColift`

3.5. Abelian Categories. [[ML71]]

Documentation 3.12. The corresponding `GAP` property is given by

`IsAbelianCategory`.

Definition 3.13. A pre-abelian category **C** is called **abelian** if every monomorphism can be regarded as a kernel embedding and every epimorphism can be regarded cokernel projection.

Basic operations for abelian categories coming from the existential quantifiers in the definition (in addition to the basic operations for pre-abelian categories):

- `LiftAlongMonomorphism`
- `ColiftAlongEpimorphism`

3.6. Monoidal Categories.

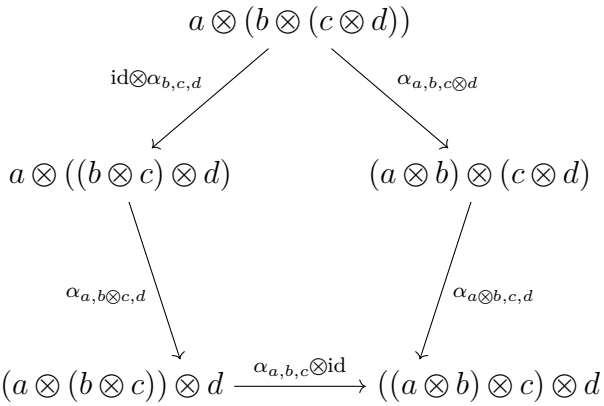
Documentation 3.14. The corresponding `GAP` property is given by

`IsMonoidalCategory`.

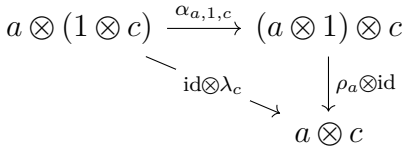
Definition 3.15 ([ML71]). A 6-tuple $(\mathbf{C}, \otimes, 1, \alpha, \lambda, \rho)$ consisting of

- a category $\mathbf{C}$,
- a functor $\otimes : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$,
- an object $1 \in \mathbf{C}$,
- a natural isomorphism $\alpha_{a,b,c} : a \otimes (b \otimes c) \cong (a \otimes b) \otimes c$,
- a natural isomorphism $\lambda_a : 1 \otimes a \cong a$,
- a natural isomorphism $\rho_a : a \otimes 1 \cong a$,

is called a **monoidal category** if the pentagonal diagram



and the triangular diagram



commute.

Basic operations for monoidal categories coming from the existential quantifiers in the definition:

- `TensorProductOnObjects`
- `TensorProductOnMorphisms`
- `TensorUnit`
- `AssociatorLeftToRight`
- `AssociatorRightToLeft`

- `LeftUnitor`
- `LeftUnitorInverse`
- `RightUnitor`
- `RightUnitorInverse`

3.7. Braided Monoidal Categories.

Documentation 3.16. The corresponding `GAP` property is given by

`IsBraidedMonoidalCategory`.

Definition 3.17 ([ML71]). A monoidal category $\mathbf{C}$ equipped with a natural isomorphism

$$B_{a,b} : a \otimes b \cong b \otimes a$$

is called a **braided monoidal category** if the following diagrams commute:

- Compatibility with the tensor unit:

$$\begin{array}{ccc} a \otimes 1 & \xrightarrow{B_{a,1}} & 1 \otimes a \\ & \searrow^{\rho_a} & \downarrow{\scriptstyle \lambda_a} \\ & & a \end{array}$$

- Swapping b, c in the term $(a \otimes b) \otimes c$:

$$\begin{array}{ccc} (a \otimes b) \otimes c & \xrightarrow{B} & c \otimes (a \otimes b) \\ \downarrow{\scriptstyle \alpha^{-1}} & & \downarrow{\scriptstyle \alpha} \\ a \otimes (b \otimes c) & & (c \otimes a) \otimes b \\ \downarrow{\scriptstyle \mathrm{id} \otimes B} & & \downarrow{\scriptstyle B \otimes \mathrm{id}} \\ a \otimes (c \otimes b) & \xrightarrow{\alpha} & (a \otimes c) \otimes b \end{array}$$

- Swapping a, b in the term $a \otimes (b \otimes c)$:

$$\begin{array}{ccc} a \otimes (b \otimes c) & \xrightarrow{B} & (b \otimes c) \otimes a \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \alpha^{-1}} \\ (a \otimes b) \otimes c & & b \otimes (c \otimes a) \\ \downarrow{\scriptstyle B \otimes \mathrm{id}} & & \downarrow{\scriptstyle \mathrm{id} \otimes B} \\ (b \otimes a) \otimes c & \xrightarrow{\alpha^{-1}} & b \otimes (a \otimes c) \end{array}$$

Basic operations for braided monoidal categories coming from the existential quantifiers in the definition (in addition to the basic operations for monoidal categories):

- `Braiding`
- `BraidingInverse`

3.8. Symmetric Monoidal Categories.

Documentation 3.18. The corresponding GAP property is given by

`IsSymmetricMonoidalCategory`.

Definition 3.19 ([ML71]). A braided monoidal category $\mathbf{C}$ is called **symmetric monoidal category** if $B_{a,b}^{-1} = B_{b,a}$.

3.9. Symmetric Closed Monoidal Categories.

Documentation 3.20. The corresponding GAP property is given by

`IsSymmetricClosedMonoidalCategory`.

Definition 3.21 ([ML71]). A symmetric monoidal category $\mathbf{C}$ which has for each functor $- \otimes b : \mathbf{C} \to \mathbf{C}$ a right adjoint (denoted by $\underline{\mathrm{Hom}}(b, -)$) is called a **symmetric closed monoidal category** or simply **closed category**.

REMARK 3.22. The family of right adjoints defines a bifunctor $\underline{\mathrm{Hom}}(-,-) : \mathbf{C}^{\mathrm{op}} \times \mathbf{C} \to \mathbf{C}$ which we consider as a part of the datum of a closed category.

Basic operations for symmetric closed monoidal categories coming from the existential quantifiers in the definition (in addition to the basic operations for symmetric monoidal categories):

- `InternalHomOnObjects`
- `InternalHomOnMorphisms`
- `EvaluationMorphism`
- `CoevaluationMorphism`

3.10. Rigid Symmetric Closed Monoidal Categories.

Documentation 3.23. The corresponding GAP property is given by

`IsRigidSymmetricClosedMonoidalCategory`.

Definition 3.24. [Del90] A symmetric closed monoidal category $\mathbf{C}$ satisfying

(1) the natural morphism $\underline{\mathrm{Hom}}(a_1, b_1) \otimes \underline{\mathrm{Hom}}(a_2, b_2) \to \underline{\mathrm{Hom}}(a_1 \otimes a_2, b_1 \otimes b_2)$ is an isomorphism,
(2) the natural morphism $a \to \underline{\mathrm{Hom}}(\underline{\mathrm{Hom}}(a, 1), 1)$ is an isomorphism

is called a **rigid symmetric closed monoidal category**.

Basic operations for rigid symmetric closed monoidal categories coming from the existential quantifiers in the definition (in addition to the basic operations for symmetric closed monoidal categories):

- `TensorProductInternalHomCompatibilityMorphismInverse`
- `MorphismFromBidual`

3.11. Strict Monoidal Categories.

Documentation 3.25. The corresponding GAP property is given by

`IsStrictMonoidalCategory`.

Definition 3.26. A monoidal category $\mathbf{C}$ is called **strict** if the associator and the unitors are identities.

REMARK 3.27. Note that being a strict monoidal category is a set theoretic property, but not a categorical property since it is not preserved by equivalences of categories.

3.12. Dual Objects. Let $\mathbf{C}$ be symmetric closed monoidal category, $a \in \mathbf{C}$ an object.

Definition 3.28. A dual object of a consists of

- an object $a^\vee \in \mathbf{C}$,
- a morphism $\mathrm{ev}_a : a^\vee \otimes a \to 1$ (evaluation)

such that for every $t \in \mathbf{C}$, the map

$$\begin{aligned} \mathrm{Hom}_{\mathbf{C}}(t, a^\vee) &\to \mathrm{Hom}_{\mathbf{C}}(t \otimes a, 1) \\ \alpha &\mapsto \mathrm{ev}_a \otimes (\alpha \otimes \mathrm{id}_a) \end{aligned}$$

is a bijection. If $\mathbf{C}$ is a rigid tensor category, dual objects admit a coevaluation $\mathrm{coev}_a : 1 \to a \otimes a^\vee$, i.e. a morphism such that the two compositions (given by only using the natural isomorphisms)

$$a \to 1 \otimes a \to (a \otimes a^\vee) \otimes a \to a \otimes (a^\vee \otimes a) \to a \otimes 1 \to a$$

and

$$a^\vee \to a^\vee \otimes 1 \to a^\vee \otimes (a \otimes a^\vee) \to (a^\vee \otimes a) \otimes a \to 1 \otimes a^\vee \to a^\vee$$

are identities.

4. Derivations

CHAPTER 6

Logic

CAP provides two types of logic applied to objects and morphisms of a category. In this chapter, the two layers are described together with some features of the logic. We start by giving some general remarks, then describing the logic and the syntax explicitly. After that, we give a description how to create your own logic files to the category.

1. General remarks to logic in Cap

There are currently two types of logical propagation between `GAP` objects implemented. The first is a relation between different filters.

Example 1.1. Every isomorphism is a monomorphism. So there is an implication between the properties `IsIsomorphism` and `IsMonomorphism`. Every time the property `IsIsomorphism` is set to true, the property `IsMonomorphism` is also true.

In CAP, such a propagation is handled by `TrueMethods`. That means those relations are static and cannot been turn off, but also `GAP` knows about them. We will call this type of implication the **first type**.

The second propagation is the propagation of predicates between input and output objects of a certain method.

Example 1.2. The composition of two monomorphisms is again a monomorphism. So, if for both input morphisms of the method `PreCompose` the property `IsMonomorphism` becomes set to `true`, in the resulting morphism of `PreCompose`, `IsMonomorphism` is also set to `true`.

Internally, such propagations are carried out via `ToDoLists`. This means that the properties do not need to be known when the mentioned operation is executed, but can be propagated later. As a drawback, `GAP` itself does not know about the relation before it is propagated, so it cannot take use of it. We will call this type of implication the **second type**.

1.1. Switchability of Logic. Since the first type of relations are hard wired via `TrueMethods`, there are not meant to be switched on or off depending on the category. Contrary to this, the logic defined by the relations from operations can be turned off or on for a specific CAP category. This can be achieved by using the commands `CapCategorySwitchLogicOn` and `CapCategorySwitchLogicOff`. Generally it is not a good idea to use these switches mid-computation, since the `ToDoListEntries` for the already computed objects remain

and might still be executed. If one wants no logic for a specific category, it is always best to switch off logic at creation time of the category.

1.2. Adding own logic files. For a specific category, additional logic files can be added via the CAP logic API. Those files are plain TeX files, so it is easy to include them into some manual. For the syntax please see 2 and 3. Once a category is created, a logic file can be added to the category. For the first type of implications, this is done via the command `AddTheoremFileToCategory`. The command takes the category as first argument, and the path to the file as second. For the second type of implications, this is done via the command `AddPredicateImplicationFileToCategory`. Again, this command takes the category as first argument and the path to the file as second.

2. Logic by theorems

A logical implication by theorem is simply an implication between `GAP` filters. In the logic file, the implication looks like this:

```
\begin{sequent}
\begin{align*}
  A:\Obj ~|~ \IsZero( A ) \vdash \IsTerminal( A )
\end{align*}
\end{sequent}
```

To clarify the syntax here, please note the following. Each logical part needs to be in a sequent environment, i.e., begin{sequent} is important. Next, it needs to be in an align environment, as seen above. The first part is the declaration of the type of the `GAP` object, possible choices here are `\Obj` or `\Mor`. After that, a `|`, separating the declaration of the objects from the theorem. After that, the implying filter should be separated by `\vdash` from the implied filter. Note that the filters need to be installed as commands or math operators, since it must match the name of the `GAP` filter after the backslash.

3. Logic by Predicate Implication

Syntax for predicate implications along methods is a bit more complex, since CAP has categorical operations having lists and integers as arguments. Those are reflected in the syntax of of the theorems which can be entered. We are not going to discuss the whole syntax here, but refer to the files in the `LogicForCAP` directory in CAP to have a closer look at the possibilities. Instead, we just look at a few examples.

Example 3.1. We start with a simple example.

```
\begin{sequent}
\begin{align*}
  \alpha:\Mor ~&|~ () \\
  &\vdash \IsMonomorphism\big( \KernelEmbedding( \alpha ) \big)
```

```
\end{align*}
\end{sequent}
```

This is one of the easiest declaration possible. There is only one variable defined, namely `\alpha`, and without any conditions, the result of the method `KernelEmbedding` is a monomorphism. No conditions are always marked by `()`.

Example 3.2. We continue with something more serious.

```
\begin{sequent}
\begin{align*}
  L: \ListObj ~|~ \big(\forall x \in L: \IsTerminal(x)\big)
     \vdash \IsTerminal\big( \DirectProduct( L ) \big)
\end{align*}
\end{sequent}
```

Here we see a more advanced type of implication. There is only a list of objects defined, and the precondition needs to hold for all objects in the list. This can be achieved using the syntax above.

Some general remarks on this propositions at the end: It is generally not possible nor necessary to use more then one categorical operation in a theorem. Also, there is also the possibility to define an integer variable at the beginning, and so accessing a particular entry in an input or output list.

CHAPTER 7

Caching

In this chapter, we will have a closer look at the caching which is used in CAP. Almost every function in CAP is cached, this means, consecutive calls with the same arguments lead to identical results.

1. The role of caches

Caching in CAP is mainly done for two reasons: speed and consistency. On the speed side, it is always a good idea to store already computed results. Most computers have a lot of memory right now, so storing is cheap. It also simpifies programming, since commands can be typed multiple times without worrying about potential recomputing.

The consistency part is a bit more sophisticated. Categorical operations in CAP are mathematical functions with respect to some specific equality notion. This means the user has to make sure that the input of equal input is equal. Caching can help if the function implemented for equality is not good enough to ensure this. If equal input is given to a Categorical operation, the caches look up if there already is a result for this input and returns it instead of a newly computed result.

2. Types of caching

In CAP, we generally use two types of caching: The `GAP` internal attribute/property caching and the caches implemented in CAP. They differ mainly in two aspects, namely the way arguments of functions are compared and how many arguments a cached function can have. We want to compare the two ways of caching.

2.1. Internal attribute caching.

(1) Only for one argument functions
(2) Compares arguments by `IsIdenticalObj`
(3) Always stores the result

2.2. Cap caching.

(1) Arbitrary argument function
(2) Compares arguments by `IsEqualForCache`, which can be implemented separately for different types of objects
(3) Stores the result, or just keeps a weak pointer, or can be disabled completely

Those properties of the CAP caches are needed. Of Course, CAP makes use of the GAP caches as well. If some categorical operation is a GAP attribute, it first looks at the GAP attribute, then in its CAP cache.

3. Switchability of caches

In CAP, the caches have three different stages. They can be hard caches, weak caches, and turned off. Hard caches mean that the result of a computation is stored forever, weak means it is stored as long as there is a pointer to the object, and turned off means there is no storing at all. These modes can be switched for every category and every categorical operation at every time, but we suggest to set it at the creation of a category. This setting can be manipulated by the methods `SetCachingToWeak`, `SetCachingToCrisp`, and `DeactivateCaching`. All those methods take the category as first argument and the name of the categorical operation (e.g. D̈irectSum) ̈as second. Also, there are methods which set the ceaches for all operations of the category. Those only take the category as argument. Their names are `SetCachingOfCategoryWeak`, `SetCachingOfCategoryCrisp`, and `DeactivateCachingOfCategory`.

CHAPTER 8

Index and Notation

- **`GAP` *:** interpret * in the context of `GAP`, e.g., a `GAP` filter is a filter in the context of `GAP`.
- **`GAP` function:** function object in `GAP`.
- **`GAP` operation:** operation object in `GAP`.
- **`GAP` method:** function installed for some operation via InstallMethod.
- **Add function:** function with name `Add*`, which adds functions as methods to the operation *.
- **Basic operation:** operation for which functions can be added to the category, e.g., `PreCompose`.
- **Categorical operation:** basic operation which represents a categorical construction. `IsIdenticalToIdentity` is a basic operation which is not a Categorical operation.
- **Basic operation symbol:** The variable name or string which represents a basic operation.
- **`WithGiven` operation:** basic operation which has the string `WithGiven` in its basic operation symbol.
- **Primitive operation:** basic operation in a category for which the functions are installed via Add functions, not via derivations.
- **Derived operation:** basic operation in a category which is installed by the derivation mechanism of CAP.
- **Category:** CAP category.
- CAP **category:** concept of category implemented in CAP, see definition 1.1.
- **Classical category:** Category as described in [ML71].
- **Category object:** `GAP` objects which represents a CAP category, not to be confused with the `GAP` category.
- **Object:** `GAP` object which represents an object of a category.
- **Morphism:** `GAP` object which represents a Morphism of a category.
- **Twocell:** `GAP` object which represents a twocell of a 2-category.
- CAP **:** always means the complete CAP project.

Bibliography

[BLH14] Mohamed Barakat and Markus Lange-Hegermann, *Gabriel morphisms and the computability of Serre quotients with applications to coherent sheaves*, (arXiv:1409.2028), 2014. 45

[Del90] P. Deligne, *Catégories tannakiennes*, The Grothendieck Festschrift, Vol. II, Progr. Math., vol. 87, Birkhäuser Boston, Boston, MA, 1990, pp. 111–195. MR 1106898 (92d:14002) 49

[HS96] Martin Hofmann and Thomas Streicher, *The groupoid interpretation of type theory*, In Venice Festschrift, Oxford University Press, 1996, pp. 83–111. 35

[ML71] Saunders Mac Lane, *Categories for the working mathematician*, Graduate Texts in Mathematics, no. 5, Springer-Verlag, 1971. 28, 45, 46, 47, 48, 49, 57

[Uni13] The Univalent Foundations Program, *Homotopy type theory: Univalent foundations of mathematics*, http://homotopytypetheory.org/book, Institute for Advanced Study, 2013. 32, 35, 36