Contents

1 Preface

2 Introduction to quivers and category theory

3 Datatype convention of catreps

Since the goal of this thesis is a translation of the package `catreps` by Peter Webb et al. into CAP, this section is a short overview of the package catreps.

> In this package a category is stored as a concrete category (i.e. a category where the objects are sets and morphisms are maps of sets). A category is stored as a record (cat, say) with fields cat.objects, cat.generators, cat.domain, cat.codomain. Each object in the list cat.object is a set, and each morphism in the list of generator morphisms cat.generators is stored as a mapping of sets, which we notate as the list of its values. ([5])

Example
```
gap> c3c3 := ConcreteCategory( [ [2,3,1], [4,5,6], [,,,5,6,4] ] );
rec( codomain := [ 1, 2, 2 ], domain := [ 1, 1, 2 ],
     generators := [ [ 2, 3, 1 ], [ 4, 5, 6 ], [ ,,, 5, 6, 4 ] ],
     objects := [ [ 1, 2, 3 ], [ 4, 5, 6 ] ], operations := rec(  ) )
```

The list of values as seen in the example above may be easy to type in, but does have its disadvantages: If for example you want to store the morphism that maps the set $\{9\}$ to itself, i.e. the identity morphism $1_{\{9\}}$, you first have to write the eight commas that are not part of that morphism definition `[ ,,,,,,,,9 ]` and you might make a mistake by forgetting one comma. Another issue is that the source object of a morphism `gen` is only implicitly given by those list entries `i` for which `IsBound( gen[i] ) = true` .

Using instead `MapOfFinSets` in CAP solves both of these issues, and it lets us use a different model for concrete categories in CAP, i.e. that of a subcategory of `FinSets`, for which we already have an implementation in CAP. Another advantages of this method is that a `MapOfFinSets` can cache known properties about itself:

Example
```
gap> S := FinSet( [1,2,3] );
<An object in FinSets>
gap> T := FinSet( [4,5,6] );
<An object in FinSets>
gap> map1 := MapOfFinSets( S, [ [1,1], [2,2], [3,3] ], S );
<A morphism in FinSets>
gap> IsAutomorphism( map1 );
true
gap> map1;
<An automorphism in FinSets>
```

Going further in the cited example,
"The following constructs a representation:" ([5])

Example
```
gap> one:=One(GF(3));;
gap> d:=[[1,1,0,0,0],[0,1,1,0,0],[0,0,1,0,0],[0,0,0,1,1],[0,0,0,0,1]]*one;;
gap> e:=[[0,1,0,0],[0,0,1,0],[0,0,0,0],[0,1,0,1],[0,0,1,0]]*one;;
gap> f:=[[1,1,0,0],[0,1,1,0],[0,0,1,0],[0,0,0,1]]*one;;
gap> nine:=CatRep(c3c3,[d,e,f],GF(3));
rec(
category := rec( generators := [ [ 2, 3, 1 ], [ 4, 5, 6 ], [ ,,, 5, 6, 4 ] ]
, operations := rec( ), objects := [ [ 1, 2, 3 ], [ 4, 5, 6 ] ],
domain := [ 1, 1, 2 ], codomain := [ 1, 2, 2 ] ),
genimages := [ [ [ Z(3)^0, Z(3)^0, 0*Z(3), 0*Z(3), 0*Z(3) ],
[ 0*Z(3), Z(3)^0, Z(3)^0, 0*Z(3), 0*Z(3) ],
[ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ],
[ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0, Z(3)^0 ],
[ 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ] ],
[ [ 0*Z(3), Z(3)^0, 0*Z(3), 0*Z(3) ], [ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ]
, [ 0*Z(3), 0*Z(3), 0*Z(3), 0*Z(3) ],
[ 0*Z(3), Z(3)^0, 0*Z(3), Z(3)^0 ],
[ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ] ],
[ [ Z(3)^0, Z(3)^0, 0*Z(3), 0*Z(3) ], [ 0*Z(3), Z(3)^0, Z(3)^0, 0*Z(3) ]
, [ 0*Z(3), 0*Z(3), Z(3)^0, 0*Z(3) ],
[ 0*Z(3), 0*Z(3), 0*Z(3), Z(3)^0 ] ] ], field := GF(3),
dimension := [ 5, 4 ] )
```

we see that `catreps` works with GAP matrices directly whereas with CAP we use `HomalgMatrix` and `RingsForHomalg` which lets us delegate computation to faster computer algebra systems like `Singular` or `Magma`. What is also noticable is the big chunk of output we get as a result of `CatRep(c3c3,[d,e,f],GF(3))`. In CAP we hide the output and give a short description of the resulting object or morphism, and use the `Display` function to display the whole result.

All in all, there are plenty of reasons to change to CAP. In the meantime, in order to still support inputs in the convention of `catreps`, I wrote a converter function `ConvertToMapOfFinSets`.

4 The category FinSets

There are algorithms whose sole purpose is to convert data structures, so they are not of much interest to the mathematical theory, and then there are algorithms that implement our category theoretical calculations, so they are important to our theory.

The algorithms `ConvertToMapOfFinSets`,
`ConcreteCategoryForCAP` and `RightQuiverFromConcreteCategory`
are more of the data structure conversion type, while `RelationsOfEndomorphisms`,
`Algebroid`,
`EmbeddingOfSubRepresentation`
and `WeakDirectSumDecomposition` are also important to our theory.

4.1 MapOfFinSets

Algorithm 1: `ConvertToMapOfFinSets`

Input : a list *objects* of objects in FinSets and a morphism *gen* given as a list of images in the convention of catreps

Output : the corresponding map of finite sets from source S to target T

let T be the first object $O \in objects$ such that $gen \cap O \neq \emptyset$;
if $gen \cap O = \emptyset \, \forall O \in objects$ **then**
  Error "unable to find target set"
end
let fl be the flattening of *objects* as a list;
let S be the sublist of fl according to positions i such that $gen[i]$ is bound;
set S to be the first object $O \in objects$ such that $O = S$;
if $S \neq O \, \forall O \in objects$ **then**
  Error "unable to find source set"
end
let G be the list of pairs $[i, gen[i]], i \in S$;
return MapOfFinSets(S, G, T);

5 The categories Functor-Categories and Cat-Reps

6 Conclusion

References

[1] `https://web.northeastern.edu/martsinkovsky/p/Parnu2019/slides-facchini.pdf`

[2] `https://www.math.uni-bielefeld.de/~sek/kau/leit4.pdf`

[3] Jan Geuenich. `https://hss.ulb.uni-bonn.de/2017/4681/4681.pdf`

[4] Mohamed Barakat, Julia Mickisch and Fabian Zickgraf, `FinSetsForCAP`, `https://github.com/mohamed-barakat/FinSetsForCAP/`

[5] Peter Webb, Dan Christensen, Fan Zhang, and Moriah Elkin, `catrepstutorialMarch11`, `http://www-users.math.umn.edu/~webb/GAPfiles/catrepstutorial.html`

A Implementation in Cap

Listings

Procedure 1: ConvertToMapOfFinSets

```
 function( objects, gen )
  local O, T, fl, S, G, i;

  T := First( objects, O -> Length(
  Intersection( gen, AsList( O ) ) ) > 0 );

  if T = fail then
      Error( "unable to find target set\n" );
  fi;

  fl := Flat( List( objects, O -> AsList( O ) ));
  S := fl{PositionsProperty( fl , i -> IsBound( gen[i] ))};

  S := First( objects, O -> AsList( O ) = S );

  if S = fail then
      Error( "unable to find source set\n" );
  fi;

  G := [ ];

  G := List( S, i -> [ i, gen[i] ] ); # gen[i] is sure to be bound

  return MapOfFinSets( S, G, T );

end );
```

Back to Algorithm 1.
Back to Index

Procedure 2: ConcreteCategoryForCAP

```
 function( L )
  local C, c, objects;

  DeactivateCachingOfCategory( FinSets );
  CapCategorySwitchLogicOff( FinSets );
  DisableSanityChecks( FinSets );
```

```
  C := Subcategory( FinSets, "A finite concrete category" :
  overhead := false, FinalizeCategory := false );

  DeactivateCachingOfCategory( C );
  CapCategorySwitchLogicOff( C );
  DisableSanityChecks( C );

  SetFilterObj( C, IsFiniteConcreteCategory );

  AddIsAutomorphism( C,
    function( alpha )
      return IsAutomorphism( UnderlyingCell( alpha ) );
  end );

  AddInverse( C,
    function( alpha )
      return Inverse( UnderlyingCell( alpha ) ) / CapCategory( alpha );
  end );

  c := ConcreteCategory( L );

  C!.ConcreteCategoryRecord := c;

  objects := List( c.objects, FinSet );

  SetSetOfObjects( C, List( objects, o -> o / C ) );

  SetSetOfGeneratingMorphisms( C, List(
  c.generators, g -> ConvertToMapOfFinSets( objects, g ) / C ) );

  Finalize( C );

  return C;

end );
```

Procedure 3: RightQuiverFromConcreteCategory

```
 function( C )
  local objects, gmorphisms, arrows, i, mor, q;

  objects := SetOfObjects( C );
  gmorphisms := SetOfGeneratingMorphisms( C );
  arrows := [];

  i := 1;

  for mor in gmorphisms do
      arrows[i] :=[
                  PositionProperty( objects,
                          o -> IsEqualForObjects( Source( mor ), o ) ),
                  PositionProperty( objects,
                          o -> IsEqualForObjects( Range( mor ), o ) )
```

```
                    ];
        i := i+1;
    od;

    q := RightQuiver( "q(1)[a]", Length( objects ), arrows );

    return q;

end );
```

Procedure 4: `RelationsOfEndomorphisms`

```
 function( k, C )
    local objects, gmorphisms, q, kq, relation_of_endomorphism,
          arrows, endos, vertices, i, mor, mpowers, m, npowers, n, foundEqual,
          relsEndo;

    objects := SetOfObjects( C );
    gmorphisms := SetOfGeneratingMorphisms( C );
    q := RightQuiverFromConcreteCategory( C );
    kq := PathAlgebra( k, q );

    relation_of_endomorphism := function(kq, a, m, n)
        local rel, one;
        rel := [];
        if m = 0 then
            one := Source( a );
            rel := PathAsAlgebraElement( kq, a )^n
                   - PathAsAlgebraElement( kq, one );
        else
            rel := PathAsAlgebraElement( kq, a )^(m+n)
                   - PathAsAlgebraElement( kq, a )^m;
        fi;
        return rel;
    end;

    arrows := Arrows( q );
    endos := Filtered( arrows, a -> Source( a ) = Target( a ) );

    vertices := Collected( List( endos, Source ) );

    if ForAny( vertices, l -> l[2] > 1 ) then
        Error( "we assume at most 1 generating endomorphism per vertex\n" );
    fi;

    relsEndo := [];

    for i in [ 1 .. Length( gmorphisms ) ] do
        mor := gmorphisms[i];
        if not IsEndomorphism( mor ) then
            continue;
        fi;
        mpowers := [];
```

```
        m := 0;
        # sigma lemma
        foundEqual := false;
        while not mor^m in mpowers do
            n := 1;
            npowers := [];
            while not mor^(m+n) in npowers and
              not foundEqual do
                if IsCongruentForMorphisms( mor^(m+n), mor^m ) then
                    Add( relsEndo,
                         relation_of_endomorphism( kq, arrows[i], m, n ) );
                    foundEqual := true;
                fi;
                Add( npowers, mor^(m+n) );
                n := n+1;
            od;
            Add( mpowers, mor^m );
            m := m+1;
        od;
    od;

    return relsEndo;

end );
```

Procedure 5: `Algebroid`

```
  function( k, C )
    local objects, gmorphisms, q, kq, relEndo, A, F, vertices, rel,
          func, st, s, t, homST, list, p, pos;

    objects := SetOfObjects( C );
    gmorphisms := SetOfGeneratingMorphisms( C );
    q := RightQuiverFromConcreteCategory( C );
    kq := PathAlgebra( k, q );
    relEndo := RelationsOfEndomorphisms( k, C );
    A := Algebroid( kq, relEndo );
    kq := UnderlyingQuiverAlgebra( A );
    F := CapFunctor( A, objects, gmorphisms, C );

    vertices := List( SetOfObjects(A), UnderlyingVertex );

    rel := [];
    func :=
      function( p, l )
        return ForAny( l, p1->
                    IsCongruentForMorphisms(
                           ApplyToQuiverAlgebraElement( F, p ),
                           ApplyToQuiverAlgebraElement( F, p1 ) )
                    );
    end;

    for st in Cartesian(vertices,vertices) do
```

```
      s := st[1];
      t := st[2];
      if s = t then
         continue;
      fi;
      homST := BasisPathsBetweenVertices( kq, s, t );
      homST := List( homST, p -> PathAsAlgebraElement( kq, p ) );

      list := [];

      for p in homST do
         pos := PositionProperty( list, l->func(p,l) );
         if IsInt(pos) then
            Add( list[pos], p );
         else
            Add( list, [p] );
         fi;
      od;
      list := List( list, l-> List( l, p -> p!.representative ) );
      Append( rel, list );
   od;

   rel := Filtered( rel, l -> Length(l)>1 );
   rel := List( rel, l -> List( l{[ 2 .. Length(l) ]}, p -> l[1]-p ) );
   rel := Flat( rel );
   rel := Concatenation( relEndo, rel );

   kq := PathAlgebra( kq ) / rel;

   kq := PathAlgebra( kq ) / GroebnerBasis( IdealOfQuotient( kq ) );

   kq := Algebroid( kq );

   SetUnderlyingCategory( kq, C );

   SetIsLinearClosureOfACategory( kq, true );

   return kq;

end );
```

Procedure 6: EmbeddingOfSubRepresentation

```
 function( eta, F )
   local kq, objects, morphisms, subrep, embedding;

   kq := Source( CapCategory( F ) );

   eta := List( eta, function( eta_o ) if IsMonomorphism( eta_o ) then
      return eta_o; fi; return ImageEmbedding( eta_o ); end );

   objects := List( eta, Source );
   morphisms := List(
```

```
                    SetOfGeneratingMorphisms( kq ),
                    m ->
                    LiftAlongMonomorphism(
                      eta[VertexIndex( UnderlyingVertex( Range( m ) ) )],
                      PreCompose( eta[VertexIndex( UnderlyingVertex( Source( m ) ) )],
                          F( m ) ) ) );

    subrep := AsObjectInHomCategory( kq, objects, morphisms );

    embedding := AsMorphismInHomCategory( subrep, eta, F );

    SetIsMonomorphism( embedding, true );

    return embedding;

end );
```

Procedure 7: WeakDirectSumDecomposition

```
  function( F )
    local f, d, kq, k, objects, morphisms, summands, embeddings;

    f := RecordOfCatRep( F );

    d := Decompose( f );

    kq := Source( CapCategory( F ) );

    k := CommutativeRingOfLinearCategory( kq );

    d := List( d, eta -> List( [ 1 .. Length( eta ) ],
        i -> VectorSpaceMorphism(
            VectorSpaceObject( Length( eta[i] ), k ),
            eta[i],
            F( kq.(i) ) ) ) );

    return List( d, eta -> EmbeddingOfSubRepresentation( eta, F ) );

end );
```