Contents

1 Preface

2 Introduction to quivers and category theory

3 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`, `EmbeddingOfSubRepresenta` and `WeakDirectSumDecomposition` are also important to our theory.

3.1 MapOfFinSets

4 The categories Functor-Categories and Cat-Reps

5 Conclusion

References

A Implementation in Cap

Listings

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);

Function 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

Function 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 );
```

Function 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 );
```

Function 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 );
```

Function 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 );
```

Function 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( e

  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 );
```

Function 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 );
```