cartierSolution6.v

Table of Contents

Proph

https://gitee.com/OOO1337777/cartier/blob/master/cartierSolution6.v
https://gitee.com/OOO1337777/cartier/blob/master/cartierSolution6.v.pdf

solves half of some question of Cartier which is how to program grammatical polymorph non-contextual ("1-weighted") 2-fold ("2-higher") pairing-projections ("product") … (this multi-folding is the foundation of homotopy "algebraic topology"/"fibre functor")

SHORT ::

The ends is to do polymorph mathematics which is 2-folded/enriched ref some indexer which is made of all the graphs as indexes and all the graph-morphisms as arrows . Such indexer admits the generating-views ("generators") subindexer (4.5.17.h) made of : the singleton-object {0} graph (for « morphisms » , possibly interdependent with « transformations of morphisms » via non-grammatical "Yoneda" …) , and the singleton-morphism-between-two-distinct-objects {0 ~> 1} graph (for « transformations of morphisms » , for « left-whisk » composition , for « right-whisk » composition , for « inner » composition along some tight/strict or lax « cut-adherence ») , and the two structural-dividing ("boundary") {0} |- {0 ~> 1} graph-morphims (for « domain-codomain-morphisms-of-each-transformation » type-indexes) , and the structural-multiplying ("degeneracy") {0 ~> 1} |- {0} graph-morphism (for « unit-transformation-on-each-morphism » type-constructor) .

The 2-conversion-for-transformations relation shall convert across two transformations whose domain-codomain-morphisms-computation arguments are not syntactically/grammatically-the-same . But oneself does show that , by logical-deduction [convTransfCoMod_convMorCoMod_dom] [convTransfCoMod_convMorCoMod_cod] , these two domain-codomain-morphisms are indeed 1-convertible ("soundness lemma") .

Finally, some linear total/asymptotic grade is defined on the morphisms/transformations and the tactics-automated degradation lemma shows that each of the conversion indeed degrades the redex morphism/transformation .

For instant first impression , the 2-conversion-relation-for-transformations constructor which says that the first projection morphism/transformation is natural/polyarrowing (commutativity along these structure-arrow-actions : the structural-multiplying-arrow ("degeneracy") action which is the unit-transformation-on-each-morphism ['UnitTransfCoMod] type-constructor , and the two structural-dividing-arrow ("boundary") actions which are the domain-codomain-morphisms-of-each-transformation ['transfCoMod] type-indexes) , is written as :

```
| Project1_UnitTransfCoMod :
    forall (F1 F2 Z1 : obCoMod) (z1 : 'morCoMod(0 F1 ~> Z1 )0),
      ( ~_1 @ F2 _o>CoMod^ ( @'UnitTransfCoMod z1 : 'transfCoMod(0 z1 ~> z1 )0 )
        : 'transfCoMod(0 ~_1 @ F2 o>CoMod z1 ~>
                ( ~_1 @ F2 o>CoMod z1 : 'morCoMod(0 Pair F1 F2 ~> Z1 )0 ) )0 )
        <~~2 ( @'UnitTransfCoMod ( ~_1 @ F2 o>CoMod z1 ) )
```

KEYWORDS :: 1337777.OOO ; COQ ; cut-elimination ; 2-fold functors ; non-contextual 2-fold pairing-projections ; polymorph metafunctors-grammar ; modos

OUTLINE ::

- Indexer metalogic for 2-fold-enrichment
  - Generating data 2-folded-enriched ref the generating-views subindexer
- Grammatical presentation of objects and touched-morphisms 2-folded/enriched ref the generating-views subindexer
- Solution morphisms
  - Solution morphisms without polymorphism
  - Inversion of morphisms with same domain-codomain objects
  - Destruction of morphisms with inner-instantiation of object-indexes
- Grammatical 2-conversion of transformations , which infer the 1-conversions of their domain-codomain morphisms
  - Grammatical 1-conversions of morphisms
  - Linear total/asymptotic morphism-grade and the degradation lemma
- Polymorphism/cut-elimination by computational/total/asymptotic/reduction/(multi-step) resolution
- Grammatical presentation of transformations
  - Inversion of the cut-adherence (here propositional-equality)
  - Outer ("horizontal") left-whisk cut , outer ("horizontal") right-whisk cut , and inner ("vertical") composition cut with cut-adhesive
- Solution transformations
  - Solution transformations without polymorphism
  - Destruction of transformations with inner-instantiation of morphism-indexes or object-indexes
- Grammatical 2-conversion of transformations , which infer the 1-conversions of their domain-codomain morphisms
  - Grammatical 2-conversions of transformations
  - 1-convertibility of the domain/codomain morphisms for 2-convertible transformations
  - Linear total/asymptotic transformation-grade and the degradation lemma
- Polymorphism/cut-elimination by computational/total/asymptotic/reduction/(multi-step) resolution

---

HINT :: free master-engineering-thesis ; program this grammatical polymorph generated-functor-along-reindexing ("Kan extension") :

generatedFunc (I : IndexerCat) (G : GeneratorsCat) := { R : ReIndexerCat & { f : G ~> generatingFunc R | p : reIndexingFunc R |- I } }

---

1 Indexer metalogic for 2-fold-enrichment

The ends is to do polymorph mathematics which is 2-folded/enriched ref some indexer which is made of all the graphs as indexes and all the graph-morphisms as arrows . Such indexer admits the generating-views ("generators") subindexer (4.5.17.h) made of : the singleton-object {0} graph (for « morphisms » , possibly interdependent with « transformations of morphisms » via non-grammatical "Yoneda" …) , and the singleton-morphism-between-two-distinct-objects {0 ~> 1} graph (for « transformations of morphisms » , for « left-whisk » composition , for « right-whisk » composition , for « inner » composition along some tight/strict or lax « cut-adherence ») , and the two structural-dividing ("boundary") {0} |- {0 ~> 1} graph-morphims (for « domain-codomain-morphisms-of-each-transformation » type-indexes) , and the structural-multiplying ("degeneracy") {0 ~> 1} |- {0} graph-morphism (for « unit-transformation-on-each-morphism » type-constructor) .

Again : The ends is to do polymorph mathematics which is 2-folded/enriched ref some indexer (symmetric-associative-monoidal metalogic/metacategory) which is made of all the graphs as indexes and all the graph-morphisms as arrows . Such indexer admits the generating-views ("generators") subindexer made of : the singleton-object {0} graph , and the singleton-morphism-between-two-distinct-objects {0 ~> 1} graph , and the two structural-dividing ("boundary") {0} |- {0 ~> 1} graph-morphims , and the structural-multiplying ("degeneracy") {0 ~> 1} |- {0} graph-morphism . Primo this infers , for the material (as contrasted from metalogical) mathematics , that the morphisms can no longer be touched individually but many morphisms shall be touched at the same time via some indexing/multiplier/shape : when the shape is the singleton-morphism-between-two-distinct-objects {0 ~> 1} graph such touched-morphisms will be named « transformation of morphisms » ; when the shape is the singleton-object {0} graph such touched-morphisms will be named « morphism » . Secondo this infers that the two structural-dividing-arrows ("boundary") actions are represented via the domain-codomain-morphisms-of-each-transformation , and that the structural-multiplying-arrow ("degeneracy") action is represented via the unit-transformation-on-each-morphism . Tertio this infers , regardless that the common operations on the touched-morphisms are multifold/multiplicative , that oneself can avoid the multiplicative/outer/material ("horizontal") composition of transformation-next-transformation (whose output multiplicity is outside the subindexer) and instead it is sufficient to describe the multiplicative/outer/material composition of transformation-next-morphism (« right-whisk ») and the multiplicative/outer/material composition of morphism-next-transformation (« left-whisk ») (whose output multiplicity is the shape {0 ~> 1} inside the subindexer) .

1.1 Indexer metalogic admits some generating-views subindexer , and is non-contextual ("1-weighted")

As common for the more-general multifold-enriched polymorh mathematics , the indexer metalogic/metacategory is symmetric associative monoidal ; but for the 2-fold polymorh mathematics there are 3 contrasts .

Primo contrast : because of the presence of the generating-views subindexer for the 2-fold polymorh mathematics , then the presentation of this subindexer metalogic is blended with the presentation of its action on the material mathematics . This infers that the two structural-dividing-arrows ("boundary") actions are represented via the domain-codomain-morphisms-of-each-transformation type-indexes of the type-family [transfCoMod] , and that the structural-multiplying-arrow ("degeneracy") action is represented via the unit-transformation-on-each-morphism [UnitTransfCoMod] type-constructor (the constructor [UnitTransfCoMod] of the type-family [transfCoMod] , which is elsewhere also hidden/blended in the outer left/right-whisk cut constructors [TransfCoMod_PolyMorCoMod_Pre] [TransfCoMod_PolyMorCoMod_Post]) .

Secondo contrast : here there are none parameter/customized-arrow action (non-structural reindexing , customized boundary-or-degeneracy) ; is it possible to make sense of such ?

Tertio contrast : for now , oneself shall only describe non-contextual ("1-weigthed") pairing-projections , this infers that there is no grammatically-distinguished context constructor in the metalogic and consequently-for-the-material-mathematics that each projection outputs (as minimum-factor-weight as) some singleton-morphism and that the pairing ingets/inputs (as minimum-factor-weight as) two singleton-morphisms . In the future , the description of contextual pairing-projections would require some interdependence between the presentation of morphisms and the presentation of transformations , BUT THIS DEPENDENCE NEED-NOT BE GRAMMATICAL ! (inductive-inductive types) , as long as the generating morphisms-data are actually polymorph generating-views … This dependence could be expressed via the sense-decoding ("Yoneda") of the grammatical transformations .

```
From mathcomp
    Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq choice fintype tuple.
Require Omega Psatz. (* Omega.omega is too weak for degradeMor at
Pairing_Mor_morphism , also degradeTransf *)
Require Coq.Logic.Eqdep_dec.

Module TWOFOLD.

Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensive.

Arguments Nat.sub : simpl nomatch.
Arguments Nat.add : simpl nomatch.

Delimit Scope poly_scope with poly.
Open Scope poly.
```

1.2 Generating data 2-folded-enriched ref the generating-views subindexer

As common , oneself shall start from some generating data which is 2-folded/enriched ref the generating-views subindexer . But because this subindexer is non-contextual ("1-weighted") there will be no-surprise and no-contrast from the more-general multifold-enriched polymorph mathematics ; therefore this part is not described for now .

In the future , the description of contextual pairing-projections would require some interdependence between the presentation of morphisms and the presentation of transformations , BUT THIS DEPENDENCE NEED-NOT BE GRAMMATICAL ! (inductive-inductive types) , as long as the generating morphisms-data are actually polymorph generating-views … This dependence could be expressed via the sense-decoding ("Yoneda") of the grammatical transformations .

```
Parameter obCoMod_Gen : Type.
Parameter morCoMod_Gen : forall (F G : obCoMod_Gen), Type.
Parameter transfCoMod_Gen : forall (F G : obCoMod_Gen) (g g' : morCoMod_Gen F G), Type.
```

2 Grammatical presentation of objects and touched-morphisms 2-folded/enriched ref the generating-views subindexer

For 2-folded/enriched polymorph mathematics , each object can be touched individually but the morphisms can no longer be touched individually , and many morphisms shall be touched at the same time via some indexing/multiplier/shape : when the shape is the singleton-morphism-between-two-distinct-objects {0 ~> 1} graph such touched-morphisms will be named « transformation of morphisms » ; when the shape is the singleton-object {0} graph such touched-morphisms will be named « morphism » .

Each decoding ("Yoneda") of some index-for-touched-morphisms which encodes all the touched-morphisms-at-some-domain-codomain is some metafunctor-on-the-subindexer , which is therefore programmed by some inductive-family-presentation [morCoMod] for the shape {0} (morphisms) together with some inductive-family-presentation [transfCoMod] for the shape {0 ~> 1} (transformations-of-morphisms) , which could possibly be interdependent (non-grammatically "Yoneda" …) . Now the inductive-family-presentation [transfCoMod] has some additional/embedded type-indexes and type-constructor : the domain-codomain-morphisms-of-each-transformation [transfCoMod] type-indexes to represent the two structural-dividing-arrow ("boundary") actions, and the unit-transformation-on-each-morphism [UnitTransfCoMod] type-constructor to represent the structural-multiplying-arrow ("degeneracy") action .

Each decoding ("Yoneda") of the whatever-is-interesting arrows between the indexes-for-touched-morphisms are metatransformations which are programmed as some grammatical-constructors of the inductive-family-presentations [morCoMod] and [transfCoMod] .

Memo that the functoriality ("arrows-action") of each metafunctor (decoded index-for-touched-morphisms) and the naturality ("arrows-action") of each metatransformation (decoded arrow-between-indexes) is signified via the additional/embedded type-indexes of [transfCoMod] and type-constructor [UnitTransfCoMod] of [transfCoMod] . All this is effected via the two conversion relations [convMorCoMod] [convTransfCoMod] which relate those grammatical-touched-morphisms : [convMorCoMod] is for morphisms and [convTransfCoMod] is for transformations .

For 2-folded-enriched polymorph mathematics , the common operations on the touched-morphisms are multiplicative ; this contrast from internal polymorph mathematics where many morphisms are touched at the same time and moreover many objects are touched at the same time and moreover the common operations on the objects or touched-morphisms are coordinatewise/dimensional/pointwise . Memo that here the (multiplicative) outer/material ("horizontal") composition [PolyMorCoMod] [TransfCoMod_PolyMorCoMod_Pre] [TransfCoMod_PolyMorCoMod_Post] is some common operation , but there is also some uncommon operation [PolyTransfCoMod] which is the (coordinatewise/dimensional/pointwise) inner/(structure-logical) ("vertical") composition of transformation-later-transformation (along some tight/strict or lax « cut-adherence ») inside each enrichment/indexer-graph ; and both compositions cut-constructors shall be eliminated/erased .

Memo that , for the material mathematics , the decidable equality [obCoMod_eq] on the objects will enable to do any logical-inversion of the very-dependently-typed propositional-equality-across-any-two-morphisms [Inversion_Project1] [Inversion_Exfalso] , and will also enable to do the logical-inversion of any morphism whose domain-codomain-objects are the same [Inversion_domEqcod] [Inversion_toPolyMor] .

```
Inductive obCoMod : Type :=
(** | ObCoMod_Gen : obCoMod_Gen -> obCoMod *)
| Pair : obCoMod -> obCoMod -> obCoMod .

Module ObCoMod_eq.

Definition obCoMod_eq : forall F G : obCoMod, {F = G} + { ~ F = G} .
Proof.
  (** decide equality. *)
  induction F.
  destruct G.
  destruct (IHF1 G1).
  - { destruct (IHF2 G2).
      + left. clear IHF1 IHF2. subst; reflexivity.
      + right. clear IHF1 IHF2. abstract (subst; simplify_eq; done).
    }
  - right. clear IHF1 IHF2. abstract (subst; simplify_eq; done).
Defined.

Definition obCoMod_eqP : forall F : obCoMod, obCoMod_eq F F = left (Logic.eq_refl F).
Proof. induction F. simpl. rewrite IHF1 IHF2. reflexivity. Qed.

Definition Eqdep_dec_inj_pair2_eq_dec
  : forall (P : obCoMod -> Type) (p : obCoMod) (x y : P p),
    existT P p x = existT P p y -> x = y
  := Eqdep_dec.inj_pair2_eq_dec _ ObCoMod_eq.obCoMod_eq.

End ObCoMod_eq.

Reserved Notation "''morCoMod' (0 F' ~> F )0"
         (at level 0, format "''morCoMod' (0  F'  ~>  F  )0").

Inductive morCoMod : obCoMod -> obCoMod -> Type :=

| PolyMorCoMod : forall (F F' : obCoMod),
      'morCoMod(0 F' ~> F )0 -> forall (F'' : obCoMod),
        'morCoMod(0 F'' ~> F' )0 -> 'morCoMod(0 F'' ~> F )0

| UnitMorCoMod : forall (F : obCoMod),
    'morCoMod(0 F ~> F )0

(** | MorCoMod_Gen : forall (F G : obCoMod_Gen),
    morCoMod_Gen F G -> 'morCoMod(0 (ObCoMod_Gen F) ~> (ObCoMod_Gen G) )0 *)
```

```
| Project1_Mor : forall (F1 F2 : obCoMod) (Z1 : obCoMod),
    'morCoMod(0 F1 ~> Z1 )0 ->
    'morCoMod(0 (Pair F1 F2) ~> Z1 )0

| Project2_Mor : forall (F1 F2 : obCoMod) (Z2 : obCoMod),
    'morCoMod(0 F2 ~> Z2 )0 ->
    'morCoMod(0 (Pair F1 F2) ~> Z2 )0

| Pairing_Mor : forall (L : obCoMod) (F1 F2 : obCoMod),
    'morCoMod(0 L ~> F1 )0 -> 'morCoMod(0 L ~> F2 )0 ->
    'morCoMod(0 L ~> (Pair F1 F2) )0

where "''morCoMod' (0 F' ~> F )0" := (@morCoMod F' F) : poly_scope.

Notation "ff_ o>CoMod ff'" :=
  (@PolyMorCoMod _ _ ff' _ ff_) (at level 40 , ff' at next level) : poly_scope.

Notation "@ ''UnitMorCoMod' F" := (@UnitMorCoMod F)
                                   (at level 10, only parsing) : poly_scope.

Notation "''UnitMorCoMod'" := (@UnitMorCoMod _) (at level 0) : poly_scope.

(** Notation "''MorCoMod_Gen' ff" :=
      (@MorCoMod_Gen _ _ _ ff) (at level 3) : poly_scope. **)

(*  @  in  ~_1 @   says argument *)
Notation "~_1 @ F2 o>CoMod z1" :=
  (@Project1_Mor _ F2 _ z1) (at level 4, F2 at next level) : poly_scope.

Notation "~_1 o>CoMod z1" :=
  (@Project1_Mor _ _ _ z1) (at level 4) : poly_scope.

Notation "~_2 @ F1 o>CoMod z2" :=
  (@Project2_Mor F1 _ _ z2) (at level 4, F1 at next level) : poly_scope.

Notation "~_2 o>CoMod z2" :=
  (@Project2_Mor _ _ _ z2) (at level 4) : poly_scope.

Notation "<< f1 ,CoMod f2 >>" :=
  (@Pairing_Mor _ _ _ f1 f2) (at level 4, f1 at next level, f2 at next level,
                              format "<<  f1  ,CoMod  f2  >>" ) : poly_scope.
```

3 Solution morphisms

As common, the purely-grammatical polymorphism cut-constructors , for (multiplicative) outer/material composition [PolyMorCoMod] [TransfCoMod_PolyMorCoMod_Pre] [TransfCoMod_PolyMorCoMod_Post] and (coordinatewise) inner/structural composition [PolyTransfCoMod] , are not part of the solution terminology .

3.1 Solution morphisms without polymorphism

```
Module Sol.

Section Section1.
Delimit Scope sol_scope with sol.
Open Scope sol_scope.

Inductive morCoMod : obCoMod -> obCoMod -> Type :=

| UnitMorCoMod : forall (F : obCoMod),
    'morCoMod(0 F ~> F )0

| Project1_Mor : forall (F1 F2 : obCoMod) (Z1 : obCoMod),
    'morCoMod(0 F1 ~> Z1 )0 ->
```

```
    'morCoMod(0 (Pair F1 F2) ~> Z1 )0

| Project2_Mor : forall (F1 F2 : obCoMod) (Z2 : obCoMod),
    'morCoMod(0 F2 ~> Z2 )0 ->
    'morCoMod(0 (Pair F1 F2) ~> Z2 )0

| Pairing_Mor : forall (L : obCoMod) (F1 F2 : obCoMod),
    'morCoMod(0 L ~> F1 )0 -> 'morCoMod(0 L ~> F2 )0 ->
    'morCoMod(0 L ~> (Pair F1 F2) )0

where "''morCoMod' (0 F' ~> F )0" := (@morCoMod F' F) : sol_scope.

End Section1.

Module Export Ex_Notations.
  Delimit Scope sol_scope with sol.

  Notation "''morCoMod' (0 F' ~> F )0" := (@morCoMod F' F) : sol_scope.

  Notation "@ ''UnitMorCoMod' F" := (@UnitMorCoMod F)
                                          (at level 10, only parsing) : sol_scope.

  Notation "''UnitMorCoMod'" := (@UnitMorCoMod _) (at level 0) : sol_scope.

  (*  @  in  ~_1 @   says argument *)
  Notation "~_1 @ F2 o>CoMod z1" :=
    (@Project1_Mor _ F2 _ z1) (at level 4, F2 at next level) : sol_scope.

  Notation "~_1 o>CoMod z1" :=
    (@Project1_Mor _ _ _ z1) (at level 4) : sol_scope.

  Notation "~_2 @ F1 o>CoMod z2" :=
    (@Project2_Mor F1 _ _ z2) (at level 4, F1 at next level) : sol_scope.

  Notation "~_2 o>CoMod z2" :=
    (@Project2_Mor _ _ _ z2) (at level 4) : sol_scope.

  Notation "<< f1 ,CoMod f2 >>" :=
    (@Pairing_Mor _ _ _ f1 f2) (at level 4, f1 at next level, f2 at next level,
                                format "<<  f1  ,CoMod  f2  >>" ) : sol_scope.

End Ex_Notations.

Fixpoint toPolyMor (F G : obCoMod) (g : 'morCoMod(0 F ~> G )0 %sol)
         {struct g} : 'morCoMod(0 F ~> G )0 %poly .
Proof.
  refine
    match g with
    | ( @'UnitMorCoMod F )%sol => ( @'UnitMorCoMod F )%poly
    | ( ~_1 @ F2 o>CoMod z1 )%sol => ( ~_1 @ F2 o>CoMod (toPolyMor _ _ z1) )%poly
    | ( ~_2 @ F1 o>CoMod z2 )%sol => ( ~_2 @ F1 o>CoMod (toPolyMor _ _ z2) )%poly
    | ( << f1 ,CoMod f2 >> )%sol =>
      ( << (toPolyMor _ _ f1) ,CoMod (toPolyMor _ _ f2) >> )%poly
    end.
Defined.
```

3.2 Inversion of morphisms with same domain-codomain objects

In contrast to some dependent-destruction of morphisms , this dependent-inversion of morphisms with same domain-codomain objects , is logical/propositional , therefore it is only usable during deductions/proofs . But memo that it is also possible to program some nondependent-destruction of morphisms with same domain-codomain objects which is usable during both programming/data and deductions/proofs .

```
Module Inversion_domEqcod.

Inductive morCoMod_domEqcod : forall (F : obCoMod), 'morCoMod(0 F ~> F )0 %sol -> Prop :=

| UnitMorCoMod : forall (F : obCoMod),
    morCoMod_domEqcod ( @'UnitMorCoMod F )%sol

| Project1_Mor : forall (F1 F2 : obCoMod), forall (z1 : 'morCoMod(0 F1 ~> Pair F1 F2 )0%sol),
      morCoMod_domEqcod ( ~_1 @ F2 o>CoMod z1 )%sol

| Project2_Mor : forall (F1 F2 : obCoMod), forall (z2 : 'morCoMod(0 F2 ~> Pair F1 F2 )0%sol),
      morCoMod_domEqcod ( ~_2 @ F1 o>CoMod z2 )%sol

| Pairing_Mor : forall (F1 F2 : obCoMod) (f1 : 'morCoMod(0 (Pair F1 F2) ~> F1 )0 %sol)
                  (f2 : 'morCoMod(0 (Pair F1 F2) ~> F2 )0 %sol),
    morCoMod_domEqcod ( << f1 ,CoMod f2 >> )%sol .

Definition morCoMod_domEqcodP_Type
           (F G : obCoMod) (g : 'morCoMod(0 F ~> G )0 %sol ) : Type.
Proof.
  destruct (ObCoMod_eq.obCoMod_eq F G).
  - destruct e. refine (morCoMod_domEqcod g).
  - intros; refine (unit : Type).
Defined.

Lemma morCoMod_domEqcodP
  : forall (F G : obCoMod) (g : 'morCoMod(0 F ~> G )0 %sol), morCoMod_domEqcodP_Type g .
Proof.
  intros. case: F G / g.
  - intros F. unfold morCoMod_domEqcodP_Type. simpl.
    rewrite ObCoMod_eq.obCoMod_eqP.
    constructor 1.
  - intros ? ? Z1. destruct Z1 as [Z1_1 Z1_2]. intros.
    unfold morCoMod_domEqcodP_Type. simpl.
    { destruct (ObCoMod_eq.obCoMod_eq F1 Z1_1).
      * { destruct (ObCoMod_eq.obCoMod_eq F2 Z1_2).
          - simpl. subst. simpl. constructor 2.
          - intros; exact: tt.
        }
      * intros; exact: tt.
    }
  - intros ? ? Z2 * . destruct Z2 as [Z2_1 Z2_2].
    unfold morCoMod_domEqcodP_Type. simpl.
    { destruct (ObCoMod_eq.obCoMod_eq F1 Z2_1).
      * { destruct (ObCoMod_eq.obCoMod_eq F2 Z2_2).
          - simpl. subst. simpl. constructor 3.
          - intros; exact: tt.
        }
      * intros; exact: tt.
    }
  - intros L *. destruct L as [L1 L2]. unfold morCoMod_domEqcodP_Type. simpl.
    { destruct (ObCoMod_eq.obCoMod_eq L1 F1).
      + { destruct (ObCoMod_eq.obCoMod_eq L2 F2).
          - simpl. subst. simpl. constructor 4.
          - intros; exact: tt.
        }
      + intros; exact: tt.
    }
Qed.

End Inversion_domEqcod.
```

3.3 Destruction of morphisms with inner-instantiation of object-indexes

For the [morCoMod] inductive-family-presentation , there are no extra-argument/parameter (for example , the domain-codomain morphisms in [transfCoMod]) beyond the domain-codomain-objects , therefore this is the common dependent-destruction of morphisms with inner-instantiation of object-indexes

```
Module Destruct_domPair.

Inductive morCoMod_domPair
: forall (F1 F2 : obCoMod), forall (G : obCoMod),
      'morCoMod(0 (Pair F1 F2) ~> G )0 %sol -> Type :=

| UnitMorCoMod : forall (F1 F2 : obCoMod),
    morCoMod_domPair ( @'UnitMorCoMod (Pair F1 F2) )%sol

| Project1_Mor : forall (F1 F2 : obCoMod),
    forall (Z1 : obCoMod) (z1 : 'morCoMod(0 F1 ~> Z1 )0 %sol),
      morCoMod_domPair ( ~_1 @ F2 o>CoMod z1 )%sol

| Project2_Mor : forall (F1 F2 : obCoMod),
    forall (Z2 : obCoMod) (z2 : 'morCoMod(0 F2 ~> Z2 )0 %sol),
      morCoMod_domPair ( ~_2 @ F1 o>CoMod z2 )%sol

| Pairing_Mor :
    forall (M M' : obCoMod) (F1 F2 : obCoMod) (f1 : 'morCoMod(0 (Pair M M') ~> F1 )0 %sol)
      (f2 : 'morCoMod(0 (Pair M M') ~> F2 )0 %sol),
    morCoMod_domPair ( << f1 ,CoMod f2 >> )%sol .

Definition morCoMod_domPairP_Type
           (F G : obCoMod) (g : 'morCoMod(0 F ~> G )0 %sol ) :=
  ltac:( destruct F; [ (*intros; refine (unit : Type)
                         | *) refine (morCoMod_domPair g) ] ).

Lemma morCoMod_domPairP
  : forall (F G : obCoMod) (g : 'morCoMod(0 F ~> G )0 %sol), morCoMod_domPairP_Type g .
Proof.
  intros. case: F G / g.
  - destruct F; [ (*intros; exact: tt |*) ].
    constructor 1.
  - constructor 2.
  - constructor 3.
  - destruct L; [ (* intros; exact: tt | *) ].
    constructor 4.
Defined.

End Destruct_domPair.

Module Destruct_codPair.

Inductive morCoMod_codPair
: forall (F : obCoMod), forall (G1 G2 : obCoMod),
      'morCoMod(0 F ~> (Pair G1 G2) )0 %sol -> Type :=

| UnitMorCoMod : forall (F1 F2 : obCoMod),
    morCoMod_codPair ( @'UnitMorCoMod (Pair F1 F2) )%sol

| Project1_Mor : forall (F1 F2 : obCoMod),
    forall (Z1 Z1' : obCoMod) (z1 : 'morCoMod(0 F1 ~> (Pair Z1 Z1') )0 %sol),
      morCoMod_codPair ( ~_1 @ F2 o>CoMod z1 )%sol

| Project2_Mor : forall (F1 F2 : obCoMod),
    forall (Z2 Z2' : obCoMod) (z2 : 'morCoMod(0 F2 ~> (Pair Z2 Z2') )0 %sol),
      morCoMod_codPair ( ~_2 @ F1 o>CoMod z2 )%sol

| Pairing_Mor : forall (L : obCoMod) (F1 F2 : obCoMod),
    forall (f1 : 'morCoMod(0 L ~> F1 )0 %sol) (f2 : 'morCoMod(0 L ~> F2 )0 %sol),
      morCoMod_codPair ( << f1 ,CoMod f2 >> )%sol .
```

```
Definition morCoMod_codPairP_Type
           (F G : obCoMod) (g : 'morCoMod(0 F ~> G )0 %sol ) :=
  ltac:( destruct G; [ (*intros; refine (unit : Type)
                         | *) refine (morCoMod_codPair g) ] ).

Lemma morCoMod_codPairP
  : forall (F G : obCoMod) (g : 'morCoMod(0 F ~> G )0 %sol ), morCoMod_codPairP_Type g .
Proof.
  intros. case: F G / g.
  - destruct F; [ (*intros; exact: tt |*) ].
    constructor 1.
  - destruct Z1; [ (*intros; exact: tt |*) ].
    constructor 2.
  - destruct Z2; [ (*intros; exact: tt |*) ].
    constructor 3.
  - constructor 4.
Defined.

End Destruct_codPair.

End Sol.
```

4 Grammatical 2-conversion of transformations , which infer the 1-conversions of their domain-codomain morphisms

As common , the grammatical 1-conversions-for-morphisms [convMorCoMod] ans 2-conversions-for-transformations [convTransfCoMod] are classified into : the total/(multi-step) conversions , and the congruences conversions , and the constant conversions which are used in the polymorphism/cut-elimination lemma , and the constant conversions which are only for the wanted sense of pairing-projections-grammar , and the constant conversions which are only for the confluence lemma , and the constant conversions which are derivable by using the finished cut-elimination lemma .

In contrast , because of the structural-multiplying-arrow ("degeneracy") action which is the unit-transformation-on-each-morphism [UnitTransfCoMod] type-constructor (which is elsewhere also hidden/blended in the outer left/right-whisk cut constructors [TransfCoMod_PolyMorCoMod_Pre] [TransfCoMod_PolyMorCoMod_Post]) , then the 2-conversions-for-transformations [convTransfCoMod] depends/uses of the 1-conversions-for-morphisms [convMorCoMod] , via the conversion-constructors [UnitTransfCoMod_cong] [TransfCoMod_PolyMorCoMod_Pre_cong] [TransfCoMod_PolyMorCoMod_Post_cong] .

Also in contrast , because of the embedded/computed domain-codomain morphisms extra-argument/parameter in the inductive-family-presentation of the transformations , the 2-conversion-for-transformations relation shall convert across two transformations whose domain-codomain-morphisms-computation arguments are not syntactically/grammatically-the-same . But oneself does show that , by logical-deduction [convTransfCoMod_convMorCoMod_dom] [convTransfCoMod_convMorCoMod_cod] , these two domain-codomain-morphisms are indeed 1-convertible ("soundness lemma") .

Finally , some linear total/asymptotic morphism-grade [gradeMor] is defined on the morphisms and another linear total/asymptotic transformation-grade [gradeTransf] , which depends/uses of the morphism-grade [gradeMor] , is defined on the transformations ; and the tactics-automated degradation lemmas shows that each of the 1-conversions-for-morphisms or 2-conversions-for-transformations indeed degrades the redex morphism or transformation . (ERRATA: Memo that this new grade function is simplified in comparison from earlier attempts , because strict-degrading-of-the-conversions is not really required but some form of strict-degrading occurs during the computational/total/asymptotic cut-elimination …)

4.1 Grammatical 1-conversions of morphisms

```
Reserved Notation "g' <~~1 g" (at level 70).

Inductive convMorCoMod :
  forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 %poly), Prop :=
```

```
(**  ----- the total/(multi-step) conversions -----  **)

| convMorCoMod_Refl : forall (F G : obCoMod) (g : 'morCoMod(0 F ~> G )0 ),
    g <~~1 g

| convMorCoMod_Trans : forall (F G : obCoMod) (g : 'morCoMod(0 F ~> G )0 )
                         (uTrans : 'morCoMod(0 F ~> G )0 ),
    uTrans <~~1 g -> forall (g00 : 'morCoMod(0 F ~> G )0 ),
      g00 <~~1 uTrans -> g00 <~~1 g

(**  ----- the congruences conversions -----  **)

| PolyMorCoMod_cong : forall (F F' : obCoMod) (f' f'0 : 'morCoMod(0 F' ~> F )0),
    forall (F'' : obCoMod) (f_ f_0 : 'morCoMod(0 F'' ~> F' )0),
      f'0 <~~1 f' -> f_0 <~~1 f_ -> ( f_0 o>CoMod f'0 ) <~~1 ( f_ o>CoMod f' )


| Project1_Mor_cong : forall (F1 F2 : obCoMod) (Z1 : obCoMod)
                        (z1 z1' : 'morCoMod(0 F1 ~> Z1 )0),
  z1' <~~1 z1 -> ( ~_1 @ F2 o>CoMod z1' ) <~~1 ( ~_1 @ F2 o>CoMod z1 )

| Project2_Mor_cong : forall (F1 F2 : obCoMod) (Z2 : obCoMod)
                        (z2 z2' : 'morCoMod(0 F2 ~> Z2 )0),
  z2' <~~1 z2 -> ( ~_2 @ F1 o>CoMod z2' ) <~~1 ( ~_2 @ F1 o>CoMod z2 )

| Pairing_Mor_cong : forall (L : obCoMod) (F1 F2 : obCoMod),
    forall (f1 f1' : 'morCoMod(0 L ~> F1 )0) (f2 f2' : 'morCoMod(0 L ~> F2 )0),
  f1' <~~1 f1 -> f2' <~~1 f2 -> ( << f1' ,CoMod f2' >> ) <~~1 ( << f1 ,CoMod f2 >> )

(** ----- the constant conversions which are used during the polyarrowing
elimination ----- **)

(** here there are none parameter/customized-arrow action ( non-structural
reindexing , customized boundary-or-degeneracy ) ; is it possible to make sense of
such ? *)

(** ----- the constant conversions which are used during the polymorphism
elimination ----- **)

| UnitMorCoMod_morphismMor_Pre :
    forall (F F'' : obCoMod), forall (f : 'morCoMod(0 F'' ~> F )0),
        ( f ) <~~1 ( f o>CoMod ( @'UnitMorCoMod F ) )

| UnitMorCoMod_morphismMor_Post :
    forall (F F' : obCoMod), forall (f' : 'morCoMod(0 F ~> F' )0),
        (  f' ) <~~1 ( ( @'UnitMorCoMod F ) o>CoMod f' )

| Project1_Mor_morphism : forall (F1 F2 : obCoMod) (Z1 : obCoMod),
  forall (z1 : 'morCoMod(0 F1 ~> Z1 )0), forall (Y1 : obCoMod) (y : 'morCoMod(0 Z1 ~> Y1 )0),
      ( ~_1 @ F2 o>CoMod (z1 o>CoMod y) )
        <~~1 ( ( ~_1 @ F2 o>CoMod z1 ) o>CoMod y )

| Project2_Mor_morphism : forall (F1 F2 : obCoMod) (Z2 : obCoMod),
  forall (z2 : 'morCoMod(0 F2 ~> Z2 )0), forall (Y2 : obCoMod) (y : 'morCoMod(0 Z2 ~> Y2 )0),
      ( ~_2 @ F1 o>CoMod (z2 o>CoMod y) )
        <~~1 ( ( ~_2 @ F1 o>CoMod z2 ) o>CoMod y )

(**memo: Pairing_Mor_morphism_derivable below *)
| Pairing_Mor_morphism : forall (L1 L2 : obCoMod) (F1 F2 : obCoMod)
    (f1 : 'morCoMod(0 Pair L1 L2 ~> F1 )0) (f2 : 'morCoMod(0 Pair L1 L2 ~> F2 )0),
    forall (M : obCoMod) (l1 : 'morCoMod(0 M ~> L1 )0) (l2 : 'morCoMod(0 M ~> L2 )0),
      ( << ( ( << l1 ,CoMod l2 >> ) o>CoMod f1 )
        ,CoMod ( ( << l1 ,CoMod l2 >> ) o>CoMod f2 ) >> )
        <~~1 ( ( << l1 ,CoMod l2 >> ) o>CoMod ( << f1 ,CoMod f2 >> ) )

| Pairing_Mor_Project1_Mor : forall (L : obCoMod) (F1 F2 : obCoMod)
    (f1 : 'morCoMod(0 L ~> F1 )0) (f2 : 'morCoMod(0 L ~> F2 )0),
```

```
    forall (Z1 : obCoMod) (z1 : 'morCoMod(0 F1 ~> Z1 )0 ),
      ( f1 o>CoMod z1 )
        <~~1 ( ( << f1 ,CoMod f2 >> ) o>CoMod ( ~_1 @ F2 o>CoMod z1 )
             : 'morCoMod(0 L ~> Z1 )0 )

| Pairing_Mor_Project2_Mor : forall (L : obCoMod) (F1 F2 : obCoMod)
    (f1 : 'morCoMod(0 L ~> F1 )0) (f2 : 'morCoMod(0 L ~> F2 )0),
    forall (Z2 : obCoMod) (z2 : 'morCoMod(0 F2 ~> Z2 )0 ),
      ( f2 o>CoMod z2 )
        <~~1 ( ( << f1 ,CoMod f2 >> ) o>CoMod ( ~_2 @ F1 o>CoMod z2 )
             : 'morCoMod(0 L ~> Z2 )0 )

(** ----- the constant conversions which are only for the wanted sense of
pairing-projections-grammar ----- **)

(** Attention : for non-contextual ( "1-weigthed" ) pairing-projections , none of
    such thing as [Project1_Mor_Project2_Mor_Pairing_Transf] for transformations
    instead of [Project1_Mor_Project2_Mor_Pairing_Mor] for morphisms **)

| Project1_Mor_Project2_Mor_Pairing_Mor : forall (F1 F2 : obCoMod),
    ( @'UnitMorCoMod (Pair F1 F2) )
      <~~1 ( << ( ~_1 @ F2 o>CoMod ( @'UnitMorCoMod F1 ) )
           ,CoMod ( ~_2 @ F1 o>CoMod ( @'UnitMorCoMod F2 ) ) >> )

(** ----- the constant conversions which are only for the confluence lemma ---- **)

| Pairing_Mor_morphism_Project1_Mor :
    forall (L : obCoMod) (F1 F2 : obCoMod)
      (f1 : 'morCoMod(0 L ~> F1 )0) (f2 : 'morCoMod(0 L ~> F2 )0) (H : obCoMod),
      ( ~_1 @ H o>CoMod ( << f1 ,CoMod f2 >> ) )
        <~~1 ( << ( ~_1 @ H o>CoMod f1 )
             ,CoMod ( ~_1 @ H o>CoMod f2 ) >> )

| Pairing_Mor_morphism_Project2_Mor :
    forall (L : obCoMod) (F1 F2 : obCoMod)
      (f1 : 'morCoMod(0 L ~> F1 )0) (f2 : 'morCoMod(0 L ~> F2 )0) (H : obCoMod),
      ( ~_2 @ H o>CoMod ( << f1 ,CoMod f2 >> ) )
        <~~1 ( << ( ~_2 @ H o>CoMod f1 )
             ,CoMod ( ~_2 @ H o>CoMod f2 ) >> )

(** ----- the constant conversions which are derivable by using the finished
cut-elimination lemma ----- **)

(**
(*TODO: COMMENT *)
| PolyMorCoMod_morphism_Pre : forall (F F' : obCoMod) (f' : 'morCoMod(0 F' ~> F )0),
    forall (F'' : obCoMod) (f_' : 'morCoMod(0 F'' ~> F' )0),
    forall (F''' : obCoMod) (f__ : 'morCoMod(0 F''' ~> F'' )0),
      ( ( f__ o>CoMod f_' ) o>CoMod f' )
        <~~1 ( f__ o>CoMod ( f_' o>CoMod f' ) )

(*TODO: COMMENT *)
| PolyMorCoMod_morphism_Post : forall (F F' : obCoMod) (f : 'morCoMod(0 F' ~> F )0),
    forall (F'' : obCoMod) (f' : 'morCoMod(0 F'' ~> F' )0),
    forall (F''' : obCoMod) (f'' : 'morCoMod(0 F''' ~> F'' )0),
      ( f'' o>CoMod ( f' o>CoMod f ) )
        <~~1 ( ( f'' o>CoMod f' ) o>CoMod f )
**)

(** ----- the constant conversions which are derivable immediately without the
finished cut-elimination lemma ----- **)

(**
(*TODO: COMMENT *)
| Pairing_Mor_morphism_derivable : forall (L : obCoMod) (F1 F2 : obCoMod)
    (f1 : 'morCoMod(0 L ~> F1 )0) (f2 : 'morCoMod(0 L ~> F2 )0),
    forall (L' : obCoMod) (l : 'morCoMod(0 L' ~> L )0),
```

```
      ( << ( l o>CoMod f1 ) ,CoMod ( l o>CoMod f2 ) >> )
        <~~1 ( l o>CoMod ( << f1 ,CoMod f2 >> ) )
**)

where "g' <~~1 g" := (@convMorCoMod _ _ g g') .

Hint Constructors convMorCoMod.
```

4.2 Linear total/asymptotic morphism-grade and the degradation lemma

```
Notation max m n := ((Nat.add m (Nat.sub n m))%coq_nat).

Definition gradeOb (F : obCoMod) : nat := 0 .

Fixpoint gradeMor (F G : obCoMod) (g : 'morCoMod(0 F ~> G )0 ) {struct g} : nat .
Proof.
  case : F G / g .
  - intros ? ? f' ? f_ .
    exact: (2 * (S (gradeMor _ _ f' + gradeMor _ _ f_)%coq_nat))%coq_nat .
  - (* memo that the unit-transformation-on-each-morphism [ UnitTransfCoMod ]
       type-constructor is some form of structural-multiplying-arrow (
       "degeneracy" ) action ...  now: the unit-morphism-on-each-object [
       UnitMorComod ] can also be seen as some form of action , whose argument is
       this object F *)
    intros F .
    exact: (S ( gradeOb F (* = 0 *) )).
  - intros ? ? ? z1 .
    exact: (S (S (gradeMor _ _ z1))).
  - intros ? ? ? z2 .
    exact: (S (S (gradeMor _ _ z2))).
  - intros ? ? ? f1 f2 .
    refine (S (S (max (gradeMor _ _ f1) (gradeMor _ _ f2)))).
Defined.

Lemma gradeMor_gt0 : forall (F G : obCoMod) (g : 'morCoMod(0 F ~> G )0 ),
     ((S O) <= (gradeMor g))%coq_nat.
Proof. intros; case : g; intros; apply/leP; intros; simpl; auto. Qed.

Ltac tac_gradeMor_gt0 :=
  match goal with
  | [ g1 : 'morCoMod(0 _ ~> _ )0 ,
          g2 : 'morCoMod(0 _ ~> _ )0 ,
                g3 : 'morCoMod(0 _ ~> _ )0 ,
                      g4 : 'morCoMod(0 _ ~> _ )0 |- _ ] =>
    move : (@gradeMor_gt0 _ _ g1) (@gradeMor_gt0 _ _ g2)
           (@gradeMor_gt0 _ _ g3) (@gradeMor_gt0 _ _ g4)

  | [ g1 : 'morCoMod(0 _ ~> _ )0 ,
          g2 : 'morCoMod(0 _ ~> _ )0 ,
                g3 : 'morCoMod(0 _ ~> _ )0 ,
                      g4 : 'morCoMod(0 _ ~> _ )0 |- _ ] =>
    move : (@gradeMor_gt0 _ _ g1) (@gradeMor_gt0 _ _ g2)
           (@gradeMor_gt0 _ _ g3) (@gradeMor_gt0 _ _ g4)

  | [ g1 : 'morCoMod(0 _ ~> _ )0 ,
          g2 : 'morCoMod(0 _ ~> _ )0 ,
                g3 : 'morCoMod(0 _ ~> _ )0 |- _ ] =>
    move : (@gradeMor_gt0 _ _ g1) (@gradeMor_gt0 _ _ g2) (@gradeMor_gt0 _ _ g3)

  | [ g1 : 'morCoMod(0 _ ~> _ )0 ,
          g2 : 'morCoMod(0 _ ~> _ )0  |- _ ] =>
    move : (@gradeMor_gt0 _ _ g1) (@gradeMor_gt0 _ _ g2)

  | [ g1 : 'morCoMod(0 _ ~> _ )0  |- _ ] =>
    move : (@gradeMor_gt0 _ _ g1)
```

```
  end.

Lemma degradeMor
  : forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 ),
    g' <~~1 g -> ( gradeMor g' <= gradeMor g )%coq_nat .
Proof.
  intros until g'. intros red_g.
  elim : F G g g' / red_g;
    try solve [ simpl; rewrite ?/gradeOb; intros;
                abstract Psatz.nia ].
  (*memo: Omega.omega too weak at  Pairing_Mor_morphism *)
  (*erase associativities conversions then Qed. *)
Qed.

Ltac tac_degradeMor H_gradeMor :=
  intuition idtac;
  repeat match goal with
         | [ Hred : ( _ <~~1 _ ) |- _ ] =>
           move : (degradeMor Hred) ; clear Hred
         end;
  move: H_gradeMor; clear; simpl; intros;
  try tac_gradeMor_gt0; intros; Omega.omega.
```

5 Polymorphism/cut-elimination by computational/total/asymptotic/reduction/(multi-step) resolution

For 2-folded polymorph mathematics , this resolution is made of some 1-resolution-for-morphisms [solveMorCoMod] and some 2-resolution-for-transformations [solveTransfCoMod] which depends/uses of this 1-resolution-for-morphisms .

The 1-resolution-for-morphisms [solveMorCoMod] is common , but has more attention into clearly separating the computational data-content function [solveMorCoMod] of the resolution from the derived logical properties [solveMorCoModP] which are satisfied by this function [solveMorCoMod] . In other words , because the 1-resolution-for-morphisms will be used during the 2-resolution-for-transformations [solveTransfCoMod] , then this 1-resolution-for-morphisms must compute somehow (without blockage from opaque logical interference). How compute ? by definitional-metaconversions or by propositional-equations ? Now , because this 1-resolution-for-morphisms function is not programmed by morphisms-structural recursion but instead is programmed by grade-structural recursion , then it is not easily-immediately usable by the 2-resolution-for-transformations , which indeed uses the instantiated 1-resolution-for-morphisms function [solveMorCoMod0] . Therefore oneself primo shall derive the propositional-equations [solveMorCoMod0_rewrite'] corresponding to the definitional-metaconversions of the 1-resolution-for-morphisms function [solveMorCoMod] .

As always , this COQ program and deduction is mostly-automated !

```
Module Resolve.
Export Sol.Ex_Notations.

Ltac tac_reduce := simpl in * ; intuition eauto.

Fixpoint solveMorCoMod_PolyMorCoMod len {struct len} :
  forall (F F' : obCoMod) (f'Sol : 'morCoMod(0 F' ~> F )0 %sol)
    (F'' : obCoMod) (f_Sol : 'morCoMod(0 F'' ~> F' )0 %sol),
  forall gradeMor_g : (gradeMor ((Sol.toPolyMor f_Sol)
                          o>CoMod (Sol.toPolyMor f'Sol)) <= len)%coq_nat,
    'morCoMod(0 F'' ~> F )0 %sol .
Proof.
  case : len => [ | len ].

  (* len is O *)
  - ( move => ? ? ? ? ? gradeMor_g ); exfalso; clear -gradeMor_g;
      by abstract tac_degradeMor gradeMor_g.
```

```
(* len is (S len) *)
- move => F F' f'Sol F'' f_Sol gradeMor_g; destruct f_Sol as
             [ _F (* @'UnitMorCoMod _F *)
             | F1 F2 Z1 z1Sol (* ~_1 @ F2 o>CoMod z1Sol *)
             | F1 F2 Z2 z2Sol (* ~_2 @ F1 o>CoMod z2Sol *)
             | L F1 F2 f1Sol f2Sol  (* << f1Sol ,CoMod f2Sol >> *) ] .

  (* g is f_ o>CoMod f' , to (f_Sol o>CoMod f'Sol) , is (@'UnitMorCoMod _F
  o>CoMod f'Sol) *)
  * refine (f'Sol)%sol .

  (* g is f_ o>CoMod f' , to (f_Sol o>CoMod f'Sol) , is ( ( ~_1 @ F2 o>CoMod
  z1Sol ) o>CoMod f'Sol) *)
  * have [:blurb] z1Sol_o_f'Sol :=
      (@solveMorCoMod_PolyMorCoMod len _ _ f'Sol _ z1Sol blurb);
        first by clear - gradeMor_g; abstract tac_degradeMor gradeMor_g .

    refine ( ~_1 @ F2 o>CoMod z1Sol_o_f'Sol )%sol .

  (* g is f_ o>CoMod f' , to (f_Sol o>CoMod f'Sol) , is ( ( ~_2 @ F1 o>CoMod
  z2Sol ) o>CoMod f'Sol) *)
  * have [:blurb] z2Sol_o_f'Sol :=
      (@solveMorCoMod_PolyMorCoMod len _ _ f'Sol _ z2Sol blurb);
        first by clear - gradeMor_g; abstract tac_degradeMor gradeMor_g .

    refine ( ~_2 @ F1 o>CoMod z2Sol_o_f'Sol )%sol .

  (* g is f_ o>CoMod f' , to (f_Sol o>CoMod f'Sol) , is ( << f1Sol ,CoMod f2Sol
  >> o>CoMod f'Sol ) *)
  * move: (Sol.Destruct_domPair.morCoMod_domPairP f'Sol) => f'Sol_domPairP.
    { destruct f'Sol_domPairP as
          [ F1 F2  (*  ( @'UnitMorCoMod (Pair F1 F2) )%sol  *)
          | F1 F2 Z1 z1  (*  ( ~_1 @ F2 o>CoMod z1 )%sol  *)
          | F1 F2 Z2 z2  (*   ( ~_2 @ F1 o>CoMod z2 )%sol  *)
          | M M' F1 F2 f1 f2 (*  ( << f1 ,CoMod f2 >> )%sol  *) ] .

      (* g is f_ o>CoMod f' , to (f_Sol o>CoMod f'Sol) , is ( << f1Sol ,CoMod
      f2Sol >> o>CoMod @'UnitMorCoMod (Pair F1 F2) ) *)
      - refine ( << f1Sol ,CoMod f2Sol >> )%sol .

      (* g is f_ o>CoMod f' , to (f_Sol o>CoMod f'Sol) , is ( << f1Sol ,CoMod
      f2Sol >> o>CoMod ~_1 @ F2 o>CoMod z1 *)
      - have [:blurb] f1Sol_o_z1 :=
          (@solveMorCoMod_PolyMorCoMod len _ _ z1 _ f1Sol blurb);
            first by clear - gradeMor_g; abstract tac_degradeMor gradeMor_g .

        refine ( f1Sol_o_z1 )%sol .

      (* g is f_ o>CoMod f' , to (f_Sol o>CoMod f'Sol) , is ( << f1Sol ,CoMod
      f2Sol >> o>CoMod ~_2 @ F1 o>CoMod z2 *)
      - have [:blurb] f2Sol_o_z2 :=
          (@solveMorCoMod_PolyMorCoMod len _ _ z2 _ f2Sol blurb);
            first by clear - gradeMor_g; abstract tac_degradeMor gradeMor_g .

        refine ( f2Sol_o_z2 )%sol .

      (* g is f_ o>CoMod f' , to (f_Sol o>CoMod f'Sol) , is ( << f1Sol ,CoMod
      f2Sol >> o>CoMod << f1 ,CoMod f2 >> *)
      - have [:blurb] f_Sol_o_f1 :=
          (@solveMorCoMod_PolyMorCoMod len _ _ f1 _
                                 ( << f1Sol ,CoMod f2Sol >> %sol) blurb);
            first by clear - gradeMor_g; abstract tac_degradeMor gradeMor_g .

        have [:blurb] f_Sol_o_f2 :=
          (@solveMorCoMod_PolyMorCoMod len _ _ f2 _
                                 ( << f1Sol ,CoMod f2Sol >> %sol) blurb);
            first by clear - gradeMor_g; abstract tac_degradeMor gradeMor_g .
```

```
        refine ( << f_Sol_o_f1 ,CoMod f_Sol_o_f2 >> )%sol .
    }
Defined.

Arguments solveMorCoMod_PolyMorCoMod !len _ _ _ _ !f_Sol _ : simpl nomatch .

Notation "ff_ o>CoMod ff' @ gradeMor_g" :=
  (@solveMorCoMod_PolyMorCoMod _ _ _ ff' _ ff_ gradeMor_g)
    (at level 40 , ff' at next level) : sol_scope.

Lemma solveMorCoMod_PolyMorCoMod_len :
  forall len, forall (F F' : obCoMod) (f'Sol : 'morCoMod(0 F' ~> F )0 %sol)
           (F'' : obCoMod) (f_Sol : 'morCoMod(0 F'' ~> F' )0 %sol),
      forall gradeMor_g_len : (gradeMor ((Sol.toPolyMor f_Sol)
                                        o>CoMod (Sol.toPolyMor f'Sol)) <= len)%coq_nat,
      forall len', forall gradeMor_g_len' : (gradeMor ((Sol.toPolyMor f_Sol)
                                       o>CoMod (Sol.toPolyMor f'Sol)) <= len')%coq_nat,
          (@solveMorCoMod_PolyMorCoMod len _ _ f'Sol _ f_Sol gradeMor_g_len)
          = (@solveMorCoMod_PolyMorCoMod len' _ _ f'Sol _ f_Sol gradeMor_g_len') .
Proof.
  induction len as [ | len ].
  - ( move => ? ? ? ? ? gradeMor_g_len ); exfalso; clear -gradeMor_g_len;
      by abstract tac_degradeMor gradeMor_g_len.
  - intros. destruct len'.
    + exfalso; clear -gradeMor_g_len'; by abstract tac_degradeMor gradeMor_g_len'.
    + destruct f_Sol; simpl.
      * reflexivity.
      * erewrite IHlen. reflexivity.
      * erewrite IHlen. reflexivity.
      * { destruct (Sol.Destruct_domPair.morCoMod_domPairP f'Sol); simpl.
          - reflexivity.
          - erewrite IHlen. reflexivity.
          - erewrite IHlen. reflexivity.
          - erewrite IHlen. rewrite {1}[solveMorCoMod_PolyMorCoMod]lock .
            erewrite IHlen. rewrite -lock. reflexivity.
        }
Qed.

Fixpoint solveMorCoMod_PolyMorCoModP len {struct len} :
  forall (F F' : obCoMod) (f'Sol : 'morCoMod(0 F' ~> F )0 %sol)
    (F'' : obCoMod) (f_Sol : 'morCoMod(0 F'' ~> F' )0 %sol),
  forall gradeMor_g : (gradeMor ((Sol.toPolyMor f_Sol)
                             o>CoMod (Sol.toPolyMor f'Sol)) <= len)%coq_nat,
    (Sol.toPolyMor (@solveMorCoMod_PolyMorCoMod len _ _ f'Sol _ f_Sol gradeMor_g))
      <~~1 (Sol.toPolyMor f_Sol o>CoMod Sol.toPolyMor f'Sol) .
Proof.
  case : len => [ | len ].

  (* len is O *)
  - ( move => ? ? ? ? ? gradeMor_g ); exfalso; clear -gradeMor_g;
      by abstract tac_degradeMor gradeMor_g.

  (* len is (S len) *)
  - move => F F' f'Sol F'' f_Sol gradeMor_g; destruct f_Sol as
              [ _F (* @'UnitMorCoMod _F *)
              | F1 F2 Z1 z1Sol (* ~_1 @ F2 o>CoMod z1Sol *)
              | F1 F2 Z2 z2Sol (* ~_2 @ F1 o>CoMod z2Sol *)
              | L F1 F2 f1Sol f2Sol  (* << f1Sol ,CoMod f2Sol >> *) ] .

    (* g is f_ o>CoMod f' , to (f_Sol o>CoMod f'Sol) , is (@'UnitMorCoMod _F
    o>CoMod f'Sol) *)
    * clear; abstract tac_reduce.

    (* g is f_ o>CoMod f' , to (f_Sol o>CoMod f'Sol) , is ( ( ~_1 @ F2 o>CoMod
    z1Sol ) o>CoMod f'Sol) *)
    * move: (@solveMorCoMod_PolyMorCoModP len _ _ f'Sol _ z1Sol);
```

```
        clear; abstract tac_reduce.

    (* g is f_ o>CoMod f' , to (f_Sol o>CoMod f'Sol) , is ( ( ~_2 @ F1 o>CoMod
    z2Sol ) o>CoMod f'Sol) *)
    * move: (@solveMorCoMod_PolyMorCoModP len _ _ f'Sol _ z2Sol);
        clear; abstract tac_reduce.

    (* g is f_ o>CoMod f' , to (f_Sol o>CoMod f'Sol) , is ( << f1Sol ,CoMod f2Sol
    >> o>CoMod f'Sol ) *)
    * move: (Sol.Destruct_domPair.morCoMod_domPairP f'Sol) => f'Sol_domPairP.
      { destruct f'Sol_domPairP as
            [ F1 F2  (*  ( @'UnitMorCoMod (Pair F1 F2) )%sol  *)
            | F1 F2 Z1 z1  (*  ( ~_1 @ F2 o>CoMod z1 )%sol  *)
            | F1 F2 Z2 z2  (*   ( ~_2 @ F1 o>CoMod z2 )%sol  *)
            | M M' F1 F2 f1 f2 (*  ( << f1 ,CoMod f2 >> )%sol  *) ] .

        (* g is f_ o>CoMod f' , to (f_Sol o>CoMod f'Sol) , is ( << f1Sol ,CoMod
        f2Sol >> o>CoMod @'UnitMorCoMod (Pair F1 F2) ) *)
        - clear; abstract tac_reduce.

        (* g is f_ o>CoMod f' , to (f_Sol o>CoMod f'Sol) , is ( << f1Sol ,CoMod
        f2Sol >> o>CoMod ~_1 @ F2 o>CoMod z1 *)
        - move: (@solveMorCoMod_PolyMorCoModP len _ _ z1 _ f1Sol);
            clear; abstract tac_reduce.

        (* g is f_ o>CoMod f' , to (f_Sol o>CoMod f'Sol) , is ( << f1Sol ,CoMod
        f2Sol >> o>CoMod ~_2 @ F1 o>CoMod z2 *)
        - move: (@solveMorCoMod_PolyMorCoModP len _ _ z2 _ f2Sol);
            clear; abstract tac_reduce.

        (* g is f_ o>CoMod f' , to (f_Sol o>CoMod f'Sol) , is ( << f1Sol ,CoMod
        f2Sol >> o>CoMod << f1 ,CoMod f2 >> *)
        - move: (@solveMorCoMod_PolyMorCoModP len _ _ f1 _
                                                    ( << f1Sol ,CoMod f2Sol >> %sol))
                  (@solveMorCoMod_PolyMorCoModP len _ _ f2 _
                                                      ( << f1Sol ,CoMod f2Sol >> %sol));
            clear; abstract tac_reduce.
      }
Defined.

Fixpoint solveMorCoMod len {struct len} :
  forall (F G : obCoMod) (g : 'morCoMod(0 F ~> G )0 ),
  forall gradeMor_g : (gradeMor g <= len)%coq_nat,
    'morCoMod(0 F ~> G )0 %sol .
Proof.
  case : len => [ | len ].

  (* len is O *)
  - ( move => F G g gradeMor_g ); exfalso; abstract tac_degradeMor gradeMor_g.

  (* len is (S len) *)
  - move => F G g; case : F G / g =>
    [ F F' f' F'' f_ (* f_ o>CoMod f' *)
    | F (* @'UnitMorCoMod F *)
    | F1 F2 Z1 z1 (* ~_1 @ F2 o>CoMod z1 *)
    | F1 F2 Z2 z2 (* ~_2 @ F1 o>CoMod z2 *)
    | L F1 F2 f1 f2 (* << f1 ,CoMod f2 >> *)
    ] gradeMor_g .

    (* g is f_ o>CoMod f' *)
    + all: cycle 1.

    (* g is @'UnitMorCoMod F *)
    + refine (@'UnitMorCoMod F)%sol.

    (* g is ~_1 @ F2 o>CoMod z1 *)
    + have [:blurb] z1Sol := (solveMorCoMod len _ _ z1 blurb);
```

```
      first by clear -gradeMor_g; abstract tac_degradeMor gradeMor_g.

    refine ( ~_1 @ F2 o>CoMod z1Sol )%sol.

  (* g is ~_2 @ F1 o>CoMod z2 *)
  + have [:blurb] z2Sol := (solveMorCoMod len _ _ z2 blurb);
       first by clear -gradeMor_g; abstract tac_degradeMor gradeMor_g.

    refine ( ~_2 @ F1 o>CoMod z2Sol )%sol.

  (* g is << f1 ,CoMod f2 >> *)
  + have [:blurb] f1Sol := (solveMorCoMod len _ _ f1 blurb);
      first by clear -gradeMor_g; abstract (tac_degradeMor gradeMor_g) .

    have [:blurb] f2Sol := (solveMorCoMod len _ _ f2 blurb);
      first by clear -gradeMor_g; abstract (tac_degradeMor gradeMor_g) .

    refine ( << f1Sol ,CoMod f2Sol >> )%sol.

  (* g is f_ o>CoMod f' *)
  + have [:blurb] f_Sol := (solveMorCoMod len _ _ f_ blurb);
      first by clear -gradeMor_g; abstract tac_degradeMor gradeMor_g.

    have [:blurb] f'Sol := (solveMorCoMod len _ _ f' blurb);
      first by clear -gradeMor_g; abstract tac_degradeMor gradeMor_g.

    have [:blurb] f_Sol_o_f'Sol :=
      (@solveMorCoMod_PolyMorCoMod (gradeMor ((Sol.toPolyMor f_Sol)
                       o>CoMod (Sol.toPolyMor f'Sol))) _ _ f'Sol _ f_Sol blurb);
        first by clear; abstract reflexivity.

    refine ( f_Sol_o_f'Sol )%sol.
Defined.

Arguments solveMorCoMod !len _ _ !g _ : clear implicits , simpl nomatch .

Lemma solveMorCoMod_len :
  forall len, forall (F G : obCoMod) (g : 'morCoMod(0 F ~> G )0 ),
    forall gradeMor_g_len : (gradeMor g <= len)%coq_nat,
    forall len', forall gradeMor_g_len' : (gradeMor g <= len')%coq_nat,

        (@solveMorCoMod len _ _ g gradeMor_g_len)
        = (@solveMorCoMod len' _ _ g gradeMor_g_len') .
Proof.
  induction len as [ | len ].
  - ( move => ? ? ? gradeMor_g_len ); exfalso; clear -gradeMor_g_len;
      by abstract tac_degradeMor gradeMor_g_len.
  - intros. destruct len'.
    + exfalso; clear -gradeMor_g_len'; by abstract tac_degradeMor gradeMor_g_len'.
    + destruct g; cbn -[solveMorCoMod_PolyMorCoMod]; cycle 1.
      * reflexivity.
      * erewrite IHlen. reflexivity.
      * erewrite IHlen. reflexivity.
      * erewrite IHlen. rewrite {1}[solveMorCoMod]lock . erewrite IHlen.
        rewrite -lock. reflexivity.
      * erewrite IHlen. rewrite {1}[solveMorCoMod]lock . erewrite IHlen.
        rewrite -lock. reflexivity.
Qed.

Fixpoint solveMorCoModP len {struct len} :
  forall (F G : obCoMod) (g : 'morCoMod(0 F ~> G )0 ),
  forall gradeMor_g : (gradeMor g <= len)%coq_nat,

    (Sol.toPolyMor (@solveMorCoMod len _ _ g gradeMor_g)) <~~1 g.
Proof.
  case : len => [ | len ].
```

```
  (* len is O *)
  - ( move => F G g gradeMor_g ); exfalso; abstract tac_degradeMor gradeMor_g.

  (* len is (S len) *)
  - move => F G g; case : F G / g =>
    [ F F' f' F'' f_ (* f_ o>CoMod f' *)
    | F (* @'UnitMorCoMod F *)
    | F1 F2 Z1 z1 (* ~_1 @ F2 o>CoMod z1 *)
    | F1 F2 Z2 z2 (* ~_2 @ F1 o>CoMod z2 *)
    | L F1 F2 f1 f2 (* << f1 ,CoMod f2 >> *)
    ] gradeMor_g .

    (* g is f_ o>CoMod f' *)
    + all: cycle 1.

    (* g is @'UnitMorCoMod F *)
    + clear; abstract tac_reduce.

    (* g is ~_1 @ F2 o>CoMod z1 *)
    + move: (@solveMorCoModP len _ _ z1); clear; abstract tac_reduce.

    (* g is ~_2 @ F1 o>CoMod z2 *)
    + move: (@solveMorCoModP len _ _ z2); clear; abstract tac_reduce.

    (* g is << f1 ,CoMod f2 >> *)
    + move: (@solveMorCoModP len _ _ f1) (@solveMorCoModP len _ _ f2);
        clear; abstract tac_reduce.

    (* g is f_ o>CoMod f' *)
    + move: (@solveMorCoModP len _ _ f_) (@solveMorCoModP len _ _ f') .
      move: solveMorCoMod_PolyMorCoModP.
      clear; abstract tac_reduce.
Qed.

Definition solveMorCoMod_PolyMorCoMod0 :
  forall (F F' : obCoMod) (f'Sol : 'morCoMod(0 F' ~> F )0 %sol)
    (F'' : obCoMod) (f_Sol : 'morCoMod(0 F'' ~> F' )0 %sol),
    'morCoMod(0 F'' ~> F )0 %sol .
Proof.
  intros; apply: (@solveMorCoMod_PolyMorCoMod (gradeMor ((Sol.toPolyMor f_Sol)
                                     o>CoMod (Sol.toPolyMor f'Sol)))); constructor.
Defined.

Notation "ff_ o>CoMod ff'" :=
  (@solveMorCoMod_PolyMorCoMod0 _ _ ff' _ ff_)
    (at level 40 , ff' at next level) : sol_scope.

Lemma solveMorCoMod_PolyMorCoMod0_len :
  forall (F F' : obCoMod) (f'Sol : 'morCoMod(0 F' ~> F )0 %sol)
    (F'' : obCoMod) (f_Sol : 'morCoMod(0 F'' ~> F' )0 %sol),
  forall len', forall gradeMor_g_len' : (gradeMor ((Sol.toPolyMor f_Sol)
                                   o>CoMod (Sol.toPolyMor f'Sol)) <= len')%coq_nat,

      (solveMorCoMod_PolyMorCoMod0 f'Sol f_Sol)
      = (@solveMorCoMod_PolyMorCoMod len' _ _ f'Sol _ f_Sol gradeMor_g_len') .
Proof. intros. apply:  solveMorCoMod_PolyMorCoMod_len . Qed.

Lemma solveMorCoMod_PolyMorCoMod0___UnitMorCoMod :
  ( forall (F F' : obCoMod) (f'Sol : 'morCoMod(0 F' ~> F )0),
      (@'UnitMorCoMod F') o>CoMod f'Sol = f'Sol )%sol.
Proof. intros. reflexivity. Qed.

Lemma solveMorCoMod_PolyMorCoMod0___Project1_Mor :
  ( forall (F Z1 : obCoMod) (f'Sol : 'morCoMod(0 Z1 ~> F )0),
      forall (F1 F2 : obCoMod) (z1Sol : 'morCoMod(0 F1 ~> Z1 )0),
        ( ~_1 @ F2 o>CoMod z1Sol) o>CoMod f'Sol
        = ~_1 @ F2 o>CoMod (z1Sol o>CoMod f'Sol) )%sol .
```

```
Proof.
  intros. rewrite [solveMorCoMod_PolyMorCoMod0 in LHS]lock.
  erewrite solveMorCoMod_PolyMorCoMod0_len.
  rewrite -lock /solveMorCoMod_PolyMorCoMod0. reflexivity.
Qed.

Lemma solveMorCoMod_PolyMorCoMod0___Project2_Mor :
  ( forall (F Z2 : obCoMod) (f'Sol : 'morCoMod(0 Z2 ~> F )0),
      forall (F1 F2 : obCoMod) (z2Sol : 'morCoMod(0 F2 ~> Z2 )0),
        ( ~_2 @ F1 o>CoMod z2Sol) o>CoMod f'Sol
        = ~_2 @ F1 o>CoMod (z2Sol o>CoMod f'Sol) )%sol .
Proof.
  intros. rewrite [solveMorCoMod_PolyMorCoMod0 in LHS]lock.
  erewrite solveMorCoMod_PolyMorCoMod0_len.
  rewrite -lock /solveMorCoMod_PolyMorCoMod0. reflexivity.
Qed.

Lemma solveMorCoMod_PolyMorCoMod0_UnitMorCoMod_Pairing_Mor :
  ( forall (L F1 : obCoMod) (f1Sol : 'morCoMod(0 L ~> F1 )0) (F2 : obCoMod)
      (f2Sol : 'morCoMod(0 L ~> F2 )0),
      << f1Sol ,CoMod f2Sol >> o>CoMod (@'UnitMorCoMod (Pair F1 F2))
                = << f1Sol ,CoMod f2Sol >> )%sol .
Proof. intros. reflexivity. Qed.

Lemma solveMorCoMod_PolyMorCoMod0_Project1_Mor_Pairing_Mor :
  ( forall (L F2 : obCoMod) (f2Sol : 'morCoMod(0 L ~> F2 )0) (F1 : obCoMod)
    (f1Sol : 'morCoMod(0 L ~> F1 )0) (Z1 : obCoMod) (z1 : 'morCoMod(0 F1 ~> Z1 )0),
      << f1Sol ,CoMod f2Sol >> o>CoMod (~_1 o>CoMod z1)
                = f1Sol o>CoMod z1 )%sol .
Proof.
  intros. rewrite [solveMorCoMod_PolyMorCoMod0 in LHS]lock.
  erewrite solveMorCoMod_PolyMorCoMod0_len.
  rewrite -lock /solveMorCoMod_PolyMorCoMod0. reflexivity.
Qed.

Lemma solveMorCoMod_PolyMorCoMod0_Project2_Mor_Pairing_Mor :
  ( forall (L F1 : obCoMod) (f1Sol : 'morCoMod(0 L ~> F1 )0) (F2 : obCoMod)
    (f2Sol : 'morCoMod(0 L ~> F2 )0) (Z2 : obCoMod) (z2 : 'morCoMod(0 F2 ~> Z2 )0),
      << f1Sol ,CoMod f2Sol >> o>CoMod (~_2 o>CoMod z2)
                = f2Sol o>CoMod z2 )%sol .
Proof.
  intros. rewrite [solveMorCoMod_PolyMorCoMod0 in LHS]lock.
  erewrite solveMorCoMod_PolyMorCoMod0_len.
  rewrite -lock /solveMorCoMod_PolyMorCoMod0. reflexivity.
Qed.

Lemma solveMorCoMod_PolyMorCoMod0_Pairing_Mor_Pairing_Mor :
  ( forall (L M : obCoMod) (f1Sol : 'morCoMod(0 L ~> M )0) (M' : obCoMod)
      (f2Sol : 'morCoMod(0 L ~> M' )0) (F1 F2 : obCoMod)
      (f1 : 'morCoMod(0 Pair M M' ~> F1 )0) (f2 : 'morCoMod(0 Pair M M' ~> F2 )0),
      << f1Sol ,CoMod f2Sol >> o>CoMod ( << f1 ,CoMod f2 >> )
                = ( << ( << f1Sol ,CoMod f2Sol >> o>CoMod f1 )
                    ,CoMod ( << f1Sol ,CoMod f2Sol >> o>CoMod f2 ) >> ) )%sol .
Proof.
  intros. rewrite [solveMorCoMod_PolyMorCoMod0 in LHS]lock.
  do 2 erewrite solveMorCoMod_PolyMorCoMod0_len.
  rewrite -lock /solveMorCoMod_PolyMorCoMod0. reflexivity.
Qed.

Lemma solveMorCoMod_PolyMorCoMod0P :
  forall (F F' : obCoMod) (f'Sol : 'morCoMod(0 F' ~> F )0 %sol)
    (F'' : obCoMod) (f_Sol : 'morCoMod(0 F'' ~> F' )0 %sol),
    (Sol.toPolyMor (@solveMorCoMod_PolyMorCoMod0 _ _ f'Sol _ f_Sol))
      <~~1 (Sol.toPolyMor f_Sol o>CoMod Sol.toPolyMor f'Sol) .
Proof. intros; apply:  solveMorCoMod_PolyMorCoModP. Qed.

Definition solveMorCoMod0 :
```

```
  forall (F G : obCoMod) (g : 'morCoMod(0 F ~> G )0 ),
    'morCoMod(0 F ~> G )0 %sol .
Proof.
  intros; apply: (@solveMorCoMod (gradeMor g)); constructor.
Defined.

Lemma solveMorCoMod0_len :
  forall (F G : obCoMod) (g : 'morCoMod(0 F ~> G )0 ),
  forall len, forall gradeMor_g : (gradeMor g <= len)%coq_nat,

      (solveMorCoMod0 g) = (@solveMorCoMod len _ _ g gradeMor_g) .
Proof. intros. apply:  solveMorCoMod_len . Qed.

Lemma solveMorCoMod0_UnitMorCoMod :
  forall (F : obCoMod) ,
    solveMorCoMod0 (@'UnitMorCoMod F) = (@'UnitMorCoMod F)%sol.
Proof. intros. reflexivity. Qed.

Lemma solveMorCoMod0_Project1_Mor :
  forall (F1 F2 Z1 : obCoMod) (z1 : 'morCoMod(0 F1 ~> Z1 )0),
    solveMorCoMod0 ( ~_1 @ F2 o>CoMod z1)
    = ( ~_1 @ F2 o>CoMod (solveMorCoMod0 z1) )%sol.
Proof.
  intros. rewrite [solveMorCoMod0 in LHS]lock. erewrite solveMorCoMod0_len.
  rewrite -lock /solveMorCoMod0. reflexivity.
Qed.

Lemma solveMorCoMod0_Project2_Mor :
  forall (F1 F2 Z2 : obCoMod) (z2 : 'morCoMod(0 F2 ~> Z2 )0),
    solveMorCoMod0 ( ~_2 @ F1 o>CoMod z2)
    = ( ~_2 @ F1 o>CoMod (solveMorCoMod0 z2) )%sol.
Proof.
  intros. rewrite [solveMorCoMod0 in LHS]lock. erewrite solveMorCoMod0_len.
  rewrite -lock /solveMorCoMod0. reflexivity.
Qed.

Lemma solveMorCoMod0_Pairing_Mor :
 forall (L F1 F2 : obCoMod) (f1 : 'morCoMod(0 L ~> F1 )0) (f2 : 'morCoMod(0 L ~> F2 )0),
    solveMorCoMod0 ( << f1 ,CoMod f2 >> )
    = ( << solveMorCoMod0 f1 ,CoMod solveMorCoMod0 f2 >> ) %sol.
Proof.
  intros. rewrite [solveMorCoMod0 in LHS]lock.
  do 2 erewrite solveMorCoMod0_len.
  rewrite -lock /solveMorCoMod0. reflexivity.
Qed.

Lemma solveMorCoMod0_PolyMorCoMod :
  forall (F F' : obCoMod) (f' : 'morCoMod(0 F' ~> F )0)
    (F'' : obCoMod) (f_ : 'morCoMod(0 F'' ~> F' )0),
    solveMorCoMod0 ( f_ o>CoMod f' )%poly
    = ( (solveMorCoMod0 f_) o>CoMod (solveMorCoMod0 f') )%sol.
Proof.
  intros. rewrite [solveMorCoMod0 in LHS]lock.
  do 2 erewrite solveMorCoMod0_len. erewrite solveMorCoMod_PolyMorCoMod0_len.
  rewrite -lock /solveMorCoMod0. reflexivity.
Qed.

Lemma solveMorCoMod0P :
  forall (F G : obCoMod) (g : 'morCoMod(0 F ~> G )0 ),

    (Sol.toPolyMor (@solveMorCoMod0 _ _ g)) <~~1 g.
Proof. intros. apply: solveMorCoModP . Qed.

Definition solveMorCoMod_PolyMorCoMod0_rewrite :=
  (solveMorCoMod_PolyMorCoMod0___UnitMorCoMod,
   solveMorCoMod_PolyMorCoMod0___Project1_Mor,
   solveMorCoMod_PolyMorCoMod0___Project2_Mor,
```

```
    solveMorCoMod_PolyMorCoMod0_UnitMorCoMod_Pairing_Mor,
    solveMorCoMod_PolyMorCoMod0_Project1_Mor_Pairing_Mor,
    solveMorCoMod_PolyMorCoMod0_Project2_Mor_Pairing_Mor,
    solveMorCoMod_PolyMorCoMod0_Pairing_Mor_Pairing_Mor).

Definition solveMorCoMod0_rewrite :=
  ( ( solveMorCoMod0_PolyMorCoMod, solveMorCoMod_PolyMorCoMod0_rewrite ) ,
    solveMorCoMod0_UnitMorCoMod, solveMorCoMod0_Project1_Mor,
    solveMorCoMod0_Project2_Mor, solveMorCoMod0_Pairing_Mor ).

(**
(*TODO: NOT USED , COMMENT , derivable immediately without the finished
cut-elimination lemma *)
Lemma solveMorCoMod_PolyMorCoMod0___Pairing_Mor :
  ( forall (L : obCoMod) (F1 F2 : obCoMod)
      (f1 : 'morCoMod(0 L ~> F1 )0) (f2 : 'morCoMod(0 L ~> F2 )0),
      forall (L' : obCoMod) (l : 'morCoMod(0 L' ~> L )0),
        l o>CoMod ( << f1 ,CoMod f2 >> )
        = << ( l o>CoMod f1 ) ,CoMod ( l o>CoMod f2 ) >>  )%sol .
Abort.
**)

Definition solveMorCoMod0_toPolyMor :
  forall (F G : obCoMod) (g : 'morCoMod(0 F ~> G )0 %sol),
    solveMorCoMod0 (Sol.toPolyMor g) = g .
Proof.
  induction g; simpl in *; rewrite ?solveMorCoMod0_rewrite; intros;
    repeat match goal with
           | [ Hred : ( solveMorCoMod0 _ = _ ) |- _ ] =>
             rewrite Hred; clear Hred
           end; reflexivity.
Qed.

Definition solveMorCoMod0_rewrite' :=
  ( ( solveMorCoMod0_PolyMorCoMod, solveMorCoMod_PolyMorCoMod0_rewrite ) ,
    solveMorCoMod0_UnitMorCoMod, solveMorCoMod0_Project1_Mor,
    solveMorCoMod0_Project2_Mor, solveMorCoMod0_Pairing_Mor,
    (** solveMorCoMod_PolyMorCoMod0___Pairing_Mor, **) solveMorCoMod0_toPolyMor ) .

Ltac tac_reduce_solveMorCoMod0 :=
  rewrite /= ?solveMorCoMod0_rewrite' /= ;
  intuition (subst; repeat match goal with
                           | [ Hred : ( solveMorCoMod0 _ = _ ) |- _ ] =>
                             rewrite Hred
                           end;
             rewrite /= ?solveMorCoMod0_rewrite' /= ;
             try congruence;
             eauto).
End Resolve.
```

6 Grammatical presentation of transformations

Now the inductive-family-presentation [transfCoMod] has some additional/embedded type-indexes and type-constructor : type-indexes to represent the domain-codomain-morphisms and type-constructor [UnitTransfCoMod] to represent the unit-transformation .

Each decoding ("Yoneda") of the whatever-is-interesting arrows between the indexes-for-touched-morphisms are metatransformations which are programmed as some grammatical-constructors of the inductive-family-presentations [morCoMod] and [transfCoMod] .

Memo that the functoriality ("arrows-action") of each metafunctor (decoded index-for-touched-morphisms) and the naturality ("arrows-action") of each metatransformation (decoded arrow-between-indexes) is signified via the additional/embedded type-indexes of [transfCoMod] and type-constructor [UnitTransfCoMod] of [transfCoMod] . All this is effected via the two conversion relations [convMorCoMod] [convTransfCoMod] which relate those grammatical-touched-morphisms : [convMorCoMod] is for morphisms and [convTransfCoMod] is for transformations .

Memo that here the (multiplicative) outer/material ("horizontal") composition [PolyMorCoMod] [TransfCoMod_PolyMorCoMod_Pre] [TransfCoMod_PolyMorCoMod_Post] is some common operation , but there is also some uncommon operation [PolyTransfCoMod] which is the (coordinatewise/dimensional/pointwise) inner/structural ("vertical") composition of transformation-later-transformation (along some tight/strict or lax « cut-adherence ») inside each enrichment/indexer-graph ; and both compositions cut-constructors shall be eliminated/erased .

Attention : the formulation of the inner/structural composition [PolyTransfCoMod] cut-constructor must relate the codomain-morphism of the prefix-input transformation in-relation-with the domain-morphism of the postfix-input transformation , by propositional-equality . In other words, this [PolyTransfCoMod] constructor have some extra argument/parameter , named « cut-adherence/adhesive » , which behold this propositional-equality . Such formulation is because of this fact : the 2-conversion-for-transformations do convert across two transformations whose domain-codomain-morphisms-computation arguments are not syntactically/grammatically-the-same , and therefore , during the recursion step which is the 2-resolution of the inner/structural composition [PolyTransfCoMod] cut-constructor , oneself lacks to know that the codomain-morphism of the recursively 2-resolved prefix-output transformation is grammatically-same (or propositionally-equal …) as the domain-morphism of the recursively 2-resolved postfix-output transformation ; the presence of the cut-adherence in the input will infer some adherence in the output .

During the programming of the (inner/structural) « resolved cut » [solveTransfCoMod_PolyTransfCoMod], which ingets two morphisms together with some adhesive, the precedence is such that primo each of the two input morphisms is destructed , then secondo the input adhesive is inverted such to exfalso/exclude/impossible the preceding/outer case or to obtain some subadhesive for the recursive call .

6.1 Inversion of the cut-adherence (here propositional-equality)

For the material mathematics , the decidable equality [obCoMod_eq] on the objects enables to do any logical-inversion of the cut-adherence , which here is some very-dependently-typed propositional-equality-across-any-two-morphisms . In the alternative formulation where the cut-adherence is the more-lax (instead of tight/strict) polymorphism-conversion , the lemma [convMorCoMod_toPolyMorP'] would be some confluence lemma-corollary , instead of simply being the inversion lemma below .

```
Module Transf.

Module EqMorCoMod.

Definition convMorCoMod (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 %poly)
: Prop := @eq ( 'morCoMod(0 F ~> G )0 ) g' g .

Module Export Ex_Notations.
  Notation "g' <~>1 g" := (@convMorCoMod _ _ g g') (at level 70): poly_scope .
End Ex_Notations.

Lemma convMorCoMod_eq : forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 %poly),
    g' <~>1 g -> g' = g .
Proof. trivial. Qed.

Lemma eq_convMorCoMod : forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 %poly),
    g' = g -> g' <~>1 g .
Proof. trivial. Qed.

Lemma convMorCoMod_sym : forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 %poly),
    g' <~>1 g -> g <~>1 g' .
Proof. symmetry. trivial. Qed.

Module Inversion_Project1.

Lemma convMorCoMod_Project1P'
: forall (F1 F2 : obCoMod) (Z1 : obCoMod) (z1 z1' : 'morCoMod(0 F1 ~> Z1 )0),
    ( ( ~_1 @ F2 o>CoMod  z1 ) <~>1 ( ~_1 @ F2 o>CoMod  z1' ) ) -> (z1 <~>1 z1') .
Proof.
  intros until z1'. intros H. inversion H.
```

```
  match goal with
  | [ H1 : existT _ _ _ = existT _ _ _  |- _ ] =>
    do 2 apply (ObCoMod_eq.Eqdep_dec_inj_pair2_eq_dec) in H1
  end; subst; constructor.
Qed.

End Inversion_Project1.

Module Inversion_Project2.

Lemma convMorCoMod_Project2P'
: forall (F1 F2 : obCoMod) (Z2 : obCoMod) (z2 z2' : 'morCoMod(0 F2 ~> Z2 )0),
    ( ( ~_2 @ F1 o>CoMod  z2 ) <~>1 ( ~_2 @ F1 o>CoMod  z2' ) ) -> (z2 <~>1 z2') .
Proof.
  intros until z2'. intros H. inversion H.
  match goal with
  | [ H1 : existT _ _ _ = existT _ _ _  |- _ ] =>
    do 2 apply (ObCoMod_eq.Eqdep_dec_inj_pair2_eq_dec) in H1
  end; subst; constructor.
Qed.

End Inversion_Project2.

Module Inversion_Pairing.

Lemma convMorCoMod_PairingP'
: forall (L : obCoMod) (F1 F2 : obCoMod) (f1 : 'morCoMod(0 L ~> F1 )0)
    (f2 : 'morCoMod(0 L ~> F2 )0),
    forall (g1 : 'morCoMod(0 L ~> F1 )0) (g2 : 'morCoMod(0 L ~> F2 )0),
      ( << f1 ,CoMod f2 >> <~>1 << g1 ,CoMod g2 >> )
      -> (f1 <~>1 g1) /\ (f2 <~>1 g2).
Proof.
  intros until g2. intros H. inversion H.
  do 2 match goal with
       | [ H1 : existT _ _ _ = existT _ _ _  |- _ ] =>
         do 2 apply (ObCoMod_eq.Eqdep_dec_inj_pair2_eq_dec) in H1
       end; subst; split; constructor.
Qed.

End Inversion_Pairing.

Module Inversion_Exfalso.

Lemma convMorCoMod_ExfalsoP_Project1_Project2
: forall (F1 F2 : obCoMod) (Z1 : obCoMod) (z1 : 'morCoMod(0 F1 ~> Z1 )0)
    (z2 : 'morCoMod(0 F2 ~> Z1 )0),
    ( ( ~_1 @ F2 o>CoMod  z1 ) <~>1 ( ~_2 @ F1 o>CoMod  z2 ) ) -> False .
Proof. intros until z2. intros H. inversion H. Qed.

Lemma convMorCoMod_ExfalsoP_UnitMorCoMod_Project1
  : forall (F1 F2 : obCoMod) (z1 : 'morCoMod(0 F1 ~> Pair F1 F2 )0),
    ( @'UnitMorCoMod (Pair F1 F2 ) ) <~>1 ( ~_1 @ F2 o>CoMod z1 ) -> False.
Proof. intros until z1. intros H. inversion H. Qed.

Lemma convMorCoMod_ExfalsoP_Project1_Pairing
  : forall (F1 F2 : obCoMod) (Z1 Z1' : obCoMod) (z1 : 'morCoMod(0 F1 ~> Pair Z1 Z1' )0)
    (f1 : 'morCoMod(0 Pair F1 F2 ~> Z1 )0) (f2 : 'morCoMod(0 Pair F1 F2 ~> Z1' )0),
    ( ( ~_1 @ F2 o>CoMod  z1 ) <~>1 ( << f1 ,CoMod f2 >> ) ) -> False .
Proof. intros until f2. intros H. inversion H. Qed.

Lemma convMorCoMod_ExfalsoP_Project2_Pairing
  : forall (F1 F2 : obCoMod) (Z2 Z2' : obCoMod) (z2 : 'morCoMod(0 F2 ~> Pair Z2 Z2' )0)
    (f1 : 'morCoMod(0 Pair F1 F2 ~> Z2 )0) (f2 : 'morCoMod(0 Pair F1 F2 ~> Z2' )0),
    ( ( ~_2 @ F1 o>CoMod  z2 ) <~>1 ( << f1 ,CoMod f2 >> ) ) -> False .
Proof. intros until f2. intros H. inversion H. Qed.

End Inversion_Exfalso.
```

```
Module Inversion_toPolyMor.
Import Sol.Ex_Notations.

Local Definition Sol_convMorCoMod (F G : obCoMod)
      (g g' : 'morCoMod(0 F ~> G )0 %sol) : Prop :=
  @eq ( 'morCoMod(0 F ~> G )0 %sol ) g' g .

Local Notation "g' ~~~1 g" := (@Sol_convMorCoMod _ _ g g')
                                (at level 70): sol_scope.

Lemma convMorCoMod_toPolyMorP' :
  forall (F G : obCoMod) (g' g : 'morCoMod(0 F ~> G )0 %sol),
    ( Sol.toPolyMor g' <~>1 Sol.toPolyMor g )%poly -> ( g' ~~~1 g )%sol .
Proof.
  induction g' as [ | ? ? ? ? IHg' | ? ? ? ? IHg' | ? ? ? ? IHg'1 ? IHg'2 ];
    simpl; intros g H.
  - { move: (Sol.Inversion_domEqcod.morCoMod_domEqcodP g).
      unfold Sol.Inversion_domEqcod.morCoMod_domEqcodP_Type.
      rewrite (ObCoMod_eq.obCoMod_eqP). intros g_morCoMod_domEqcodP.
      destruct g_morCoMod_domEqcodP.
      - reflexivity.
      - exfalso; inversion H.
     (* apply (Inversion_Exfalso.convMorCoMod_ExfalsoP_UnitMorCoMod_Project1 H). *)
      - exfalso; inversion H.
      - exfalso; inversion H.
    }
  - { destruct (Sol.Destruct_domPair.morCoMod_domPairP g); simpl in * .
      - exfalso; inversion H.
      - move: (Inversion_Project1.convMorCoMod_Project1P' H);
          move /IHg' -> ; reflexivity.
      - exfalso; inversion H.
      - exfalso; inversion H.
    }
  - { destruct (Sol.Destruct_domPair.morCoMod_domPairP g); simpl in * .
      - exfalso; inversion H.
      - exfalso; inversion H.
      - move: (Inversion_Project2.convMorCoMod_Project2P' H);
          move /IHg' -> ; reflexivity.
      - exfalso; inversion H.
    }
  - { destruct (Sol.Destruct_codPair.morCoMod_codPairP g); simpl in * .
      - exfalso; inversion H.
      - exfalso; inversion H.
      - exfalso; inversion H.
      - move: (Inversion_Pairing.convMorCoMod_PairingP' H)
        => [ ] /IHg'1 -> /IHg'2 ->  ; reflexivity.
    }
Qed.

End Inversion_toPolyMor.

End EqMorCoMod.
```

6.2 Outer ("horizontal") left-whisk cut , outer ("horizontal") right-whisk cut , and inner ("vertical") composition cut with cut-adhesive

The common operations on the touched-morphisms are multifold/multiplicative ; but because of the generating-views subindexer , then oneself can avoid the (multiplicative) outer/material ("horizontal") composition of transformation-next-transformation (whose output multiplicity is outside the subindexer) and instead it is sufficient to describe the (multiplicative) outer/material composition of transformation-next-morphism ("right-whisk") and the (multiplicative) outer/material composition of morphism-next-transformation ("left-whisk") (whose output multiplicity is the shape {0 ~> 1} inside the subindexer) .

Moreover, there is also inner/(structure-logical) ("vertical") composition of transformation-later-transformation inside each enrichment/indexer-graph , whose formulation has some cut-adhesive parameter between the codomain of the prefix-transformation and the domain of the infix-transformation .

```
Import EqMorCoMod.Ex_Notations.

Reserved Notation "''transfCoMod' (0 g ~> g' )0"
         (at level 0, format "''transfCoMod' (0  g  ~>  g'  )0").

(** here there are none parameter/customized-arrow action ( non-structural
reindexing , customized boundary-or-degeneracy ) ; is it possible to make sense of
such ? *)

Inductive transfCoMod : forall (F G : obCoMod), morCoMod F G -> morCoMod F G -> Type :=

| PolyTransfCoMod : forall (F G : obCoMod), forall (g g' : 'morCoMod(0 F ~> G )0),
      forall (g'g : 'transfCoMod(0 g' ~> g )0), forall (g'0 g'' : 'morCoMod(0 F ~> G )0 ),
          forall (g''g' : 'transfCoMod(0 g'' ~> g'0 )0),
          forall (eqMor : g'0 <~>1 g' ) (** cut-adherence **) ,
            'transfCoMod(0 g'' ~> g )0

| TransfCoMod_PolyMorCoMod_Pre :
 forall (F G : obCoMod), forall (g g' : 'morCoMod(0 F ~> G )0),
 forall (g'g : 'transfCoMod(0 g' ~> g )0), forall (E : obCoMod) (f : 'morCoMod(0 E ~> F )0),
     'transfCoMod(0 f o>CoMod g' ~> f o>CoMod g )0

| TransfCoMod_PolyMorCoMod_Post :
 forall (G H : obCoMod) (h : 'morCoMod(0 G ~> H )0),
 forall (F : obCoMod) (g g' : 'morCoMod(0 F ~> G )0) (g'g : 'transfCoMod(0 g' ~> g )0),
   'transfCoMod(0 g' o>CoMod h ~> g o>CoMod h )0

(** unit-transformation-on-each-morphism **)
| UnitTransfCoMod : forall (F G : obCoMod), forall (g : 'morCoMod(0 F ~> G )0),
      'transfCoMod(0 g (*memo: same g *) ~> g (*memo: same g *) )0
(**
| TransfCoMod_Gen : forall (F G : obCoMod_Gen),
    forall (g g' : morCoMod_Gen F G), transfCoMod_Gen g g' ->
      'transfCoMod(0 (MorCoMod_Gen g) ~> (MorCoMod_Gen g') )0
**)

| Project1_Transf : forall (F1 F2 : obCoMod) (Z1 : obCoMod),
    forall (z1 z1' : 'morCoMod(0 F1 ~> Z1 )0) (z1z1' : 'transfCoMod(0 z1 ~> z1' )0),
      'transfCoMod(0 ~_1 @ F2 o>CoMod z1 ~> ~_1 @ F2 o>CoMod z1' )0

| Project2_Transf : forall (F1 F2 : obCoMod) (Z2 : obCoMod),
    forall (z2 z2' : 'morCoMod(0 F2 ~> Z2 )0) (z2z2' : 'transfCoMod(0 z2 ~> z2' )0),
      'transfCoMod(0 ~_2 @ F1 o>CoMod z2 ~> ~_2 @ F1 o>CoMod z2' )0

| Pairing_Transf : forall (L : obCoMod) (F1 F2 : obCoMod),
    forall (f1 f1' : 'morCoMod(0 L ~> F1 )0) (f1f1' : 'transfCoMod(0 f1 ~> f1' )0),
    forall (f2 f2' : 'morCoMod(0 L ~> F2 )0) (f2f2' : 'transfCoMod(0 f2 ~> f2' )0),
      'transfCoMod(0 << f1 ,CoMod f2 >> ~> << f1' ,CoMod f2' >> )0

where "''transfCoMod' (0 g ~> g' )0" := (@transfCoMod _ _ g g') : poly_scope.

Notation "g''g' o^CoMod g'g # eqMor" :=
  (@PolyTransfCoMod _ _ _ _ g'g _ _ g''g' eqMor)
    (at level 40 , g'g at next level) : poly_scope.

Notation "f _o>CoMod^ g'g" :=
  (@TransfCoMod_PolyMorCoMod_Pre _ _ _ _ g'g _ f)
    (at level 40 , g'g at next level) : poly_scope.

Notation "g'g ^o>CoMod_ h" :=
  (@TransfCoMod_PolyMorCoMod_Post _ _ h _ _ _ g'g)
```

```
    (at level 40 , h at next level) : poly_scope.

Notation "@ ''UnitTransfCoMod' g" :=
  (@UnitTransfCoMod _ _ g) (at level 10, only parsing) : poly_scope.

Notation "''UnitTransfCoMod'" :=
  (@UnitTransfCoMod _ _ _) (at level 0) : poly_scope.

(** Notation "''TransfCoMod_Gen' ff" :=
      (@TransfCoMod_Gen _ _ _ ff) (at level 3) : poly_scope. **)

(*  @  in  ~_1 @   says argument *)
Notation "~_1 @ F2 _o>CoMod^ z1z1'" :=
  (@Project1_Transf _ F2 _ _ _ z1z1') (at level 4, F2 at next level) : poly_scope.

Notation "~_1 _o>CoMod^ z1z1'" :=
  (@Project1_Transf _ _ _ _ _ z1z1') (at level 4) : poly_scope.

(*  @  in  ~_2 @   says argument *)
Notation "~_2 @ F1 _o>CoMod^ z2z2'" :=
  (@Project2_Transf F1 _ _ _ _ z2z2') (at level 4, F1 at next level) : poly_scope.

Notation "~_2 _o>CoMod^ z2z2'" :=
  (@Project2_Transf _ _ _ _ _ z2z2') (at level 4) : poly_scope.

Notation "<< f1f1' ,^CoMod f2f2' >>" :=
  (@Pairing_Transf _ _ _ _ _ f1f1' _ _ f2f2')
    (at level 4, f1f1' at next level, f2f2' at next level,
     format "<<  f1f1'  ,^CoMod  f2f2'  >>") : poly_scope.
```

7 Solution transformations

As common, the purely-grammatical polymorphism cut-constructors , for (multiplicative) outer/material composition [PolyMorCoMod] [TransfCoMod_PolyMorCoMod_Pre] [TransfCoMod_PolyMorCoMod_Post] and (coordinatewise) inner/structural composition [PolyTransfCoMod] , are not part of the solution terminology .

7.1 Solution transformations without polymorphism

```
Module Sol.
Export TWOFOLD.Sol.

Section Section1.
Delimit Scope sol_scope with sol.
Open Scope sol_scope.

Inductive transfCoMod : forall (F G : obCoMod), morCoMod F G -> morCoMod F G -> Type :=

| UnitTransfCoMod : forall (F G : obCoMod), forall (g : 'morCoMod(0 F ~> G )0),
      'transfCoMod(0 g ~> g )0

(**
| TransfCoMod_Gen : forall (F G : obCoMod_Gen), forall (g g' : morCoMod_Gen F G),
    transfCoMod_Gen g g' -> 'transfCoMod(0 (MorCoMod_Gen g) ~> (MorCoMod_Gen g') )0
**)

| Project1_Transf : forall (F1 F2 : obCoMod) (Z1 : obCoMod),
    forall (z1 z1' : 'morCoMod(0 F1 ~> Z1 )0) (z1z1' : 'transfCoMod(0 z1 ~> z1' )0),
      'transfCoMod(0 ~_1 @ F2 o>CoMod z1 ~> ~_1 @ F2 o>CoMod z1' )0

| Project2_Transf : forall (F1 F2 : obCoMod) (Z2 : obCoMod),
    forall (z2 z2' : 'morCoMod(0 F2 ~> Z2 )0) (z2z2' : 'transfCoMod(0 z2 ~> z2' )0),
      'transfCoMod(0 ~_2 @ F1 o>CoMod z2 ~> ~_2 @ F1 o>CoMod z2' )0
```

```
| Pairing_Transf : forall (L : obCoMod) (F1 F2 : obCoMod),
    forall (f1 f1' : 'morCoMod(0 L ~> F1 )0) (f1f1' : 'transfCoMod(0 f1 ~> f1' )0),
    forall (f2 f2' : 'morCoMod(0 L ~> F2 )0) (f2f2' : 'transfCoMod(0 f2 ~> f2' )0),
      'transfCoMod(0 << f1 ,CoMod f2 >> ~> << f1' ,CoMod f2' >> )0

where "''transfCoMod' (0 g ~> g' )0" := (@transfCoMod _ _ g g') : sol_scope.

End Section1.

Module Export Ex_Notations.
Export TWOFOLD.Sol.Ex_Notations.
Delimit Scope sol_scope with sol.

Notation "''transfCoMod' (0 g ~> g' )0" := (@transfCoMod _ _ g g') : sol_scope.

Notation "@ ''UnitTransfCoMod' g" := (@UnitTransfCoMod _ _ g)
                                       (at level 10, only parsing) : sol_scope.

Notation "''UnitTransfCoMod'" := (@UnitTransfCoMod _ _ _) (at level 0) : sol_scope.

(*  ^CoMod  in  _o>CoMod^ says vertical transformation *)
Notation "~_1 @ F2 _o>CoMod^ z1z1'" :=
  (@Project1_Transf _ F2 _ _ _ z1z1') (at level 4, F2 at next level) : sol_scope.

Notation "~_1 _o>CoMod^ z1z1'" :=
  (@Project1_Transf _ _ _ _ _ z1z1') (at level 4) : sol_scope.

(*  ^CoMod  in  _o>CoMod^ says vertical transformation *)
Notation "~_2 @ F1 _o>CoMod^ z2z2'" :=
  (@Project2_Transf F1 _ _ _ _ z2z2') (at level 4, F1 at next level) : sol_scope.

Notation "~_2 _o>CoMod^ z2z2'" :=
  (@Project2_Transf _ _ _ _ _ z2z2') (at level 4) : sol_scope.

(*  ^CoMod  in  ,^CoMod   says vertical transformation *)
Notation "<< f1f1' ,^CoMod f2f2' >>" :=
  (@Pairing_Transf _ _ _ _ _ f1f1' _ _ f2f2')
    (at level 4, f1f1' at next level, f2f2' at next level,
     format "<<  f1f1'  ,^CoMod  f2f2'  >>") : sol_scope.

End Ex_Notations.

Fixpoint toPolyTransf (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 %sol)
         (gg' : 'transfCoMod(0 g ~> g' )0 %sol) {struct gg'} :
  'transfCoMod(0 toPolyMor g ~> toPolyMor g' )0 %poly .
Proof.
  refine
    match gg' with
    | ( @'UnitTransfCoMod f )%sol => ( @'UnitTransfCoMod (toPolyMor f) )%poly
    | ( ~_1 @ F2 _o>CoMod^ z1z1' )%sol =>
      ( ~_1 @ F2 _o>CoMod^ (toPolyTransf _ _ _ _ z1z1') )%poly
    | ( ~_2 @ F1 _o>CoMod^ z2z2' )%sol =>
      ( ~_2 @ F1 _o>CoMod^ (toPolyTransf _ _ _ _ z2z2') )%poly
    | ( << f1f1' ,^CoMod f2f2' >> )%sol =>
   ( << (toPolyTransf _ _ _ _ f1f1') ,^CoMod (toPolyTransf _ _ _ _ f2f2') >> )%poly
    end.
Defined.
```

7.2 Destruction of transformations with inner-instantiation of morphism-indexes or object-indexes

Regardless the domain-codomain-morphisms extra-argument/parameter in the inductive-family-presentation [transfCoMod] of transformations , oneself easily still-gets the common dependent-destruction of transformations with inner-instantiation of object-indexes or inner-instantiation of morphism-indexes or inner-instantiation of both morphism-indexes and object-indexes . Memo that because the cut-adhesive will be inverted such to exfalso/exclude/impossible some couplings of prefix-transformation and postfix-transformation or to obtain some subadhesive for some recursive call to resolution , therefore this below [Destruct_domProject1_Mor] and similar destruction lemmas will not be used …

```
Module Destruct_domPair.

Inductive transfCoMod_domPair
: forall (F1 F2 G : obCoMod) (g g' : 'morCoMod(0 Pair F1 F2 ~> G )0 %sol),
    'transfCoMod(0 g ~> g' )0 %sol -> Type :=

| UnitTransfCoMod : ( forall (F1 F2 G : obCoMod),
                        forall (g : 'morCoMod(0 Pair F1 F2 ~> G )0),
                          transfCoMod_domPair ( @'UnitTransfCoMod g ) )%sol

| Project1_Transf :
    ( forall (F1 F2 : obCoMod) (Z1 : obCoMod),
        forall (z1 z1' : 'morCoMod(0 F1 ~> Z1 )0) (z1z1' : 'transfCoMod(0 z1 ~> z1' )0),
          transfCoMod_domPair ( ~_1 @ F2 _o>CoMod^ z1z1' ) )%sol

| Project2_Transf :
    ( forall (F1 F2 : obCoMod) (Z2 : obCoMod),
        forall (z2 z2' : 'morCoMod(0 F2 ~> Z2 )0) (z2z2' : 'transfCoMod(0 z2 ~> z2' )0),
          transfCoMod_domPair ( ~_2 @ F1 _o>CoMod^ z2z2' ) )%sol

| Pairing_Transf :
    ( forall (L L' : obCoMod) (F1 F2 : obCoMod),
 forall (f1 f1' : 'morCoMod(0 Pair L L' ~> F1 )0) (f1f1' : 'transfCoMod(0 f1 ~> f1' )0),
 forall (f2 f2' : 'morCoMod(0 Pair L L' ~> F2 )0) (f2f2' : 'transfCoMod(0 f2 ~> f2' )0),
   transfCoMod_domPair ( << f1f1' ,^CoMod f2f2' >> ) )%sol .

Definition transfCoMod_domPairP_Type
           (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 %sol)
           (gg' : 'transfCoMod(0 g ~> g' )0 %sol) :=
  ltac:( destruct F;  refine (transfCoMod_domPair gg')  ).

Lemma transfCoMod_domPairP
  : forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 %sol)
      (gg' : 'transfCoMod(0 g ~> g' )0 %sol),
    transfCoMod_domPairP_Type gg' .
Proof.
  intros. case: F G g g' / gg' .
  - destruct F;
      constructor 1.
  - constructor 2.
  - constructor 3.
  - destruct L; constructor 4.
Defined.

End Destruct_domPair.

Module Destruct_codPair.

Inductive transfCoMod_codPair
: forall (F G1 G2 : obCoMod) (g g' : 'morCoMod(0 F ~> Pair G1 G2 )0 %sol),
    'transfCoMod(0 g ~> g' )0 %sol -> Type :=

| UnitTransfCoMod :
    ( forall (F G1 G2 : obCoMod),
        forall (g : 'morCoMod(0 F ~> Pair G1 G2 )0),
          transfCoMod_codPair ( @'UnitTransfCoMod g ) )%sol

| Project1_Transf :
    ( forall (F1 F2 : obCoMod) (Z1 Z1' : obCoMod),
        forall (z1 z1' : 'morCoMod(0 F1 ~> Pair Z1 Z1' )0)
          (z1z1' : 'transfCoMod(0 z1 ~> z1' )0),
          transfCoMod_codPair ( ~_1 @ F2 _o>CoMod^ z1z1' ) )%sol
```

```
| Project2_Transf :
    ( forall (F1 F2 : obCoMod) (Z2 Z2' : obCoMod),
        forall (z2 z2' : 'morCoMod(0 F2 ~> Pair Z2 Z2' )0)
          (z2z2' : 'transfCoMod(0 z2 ~> z2' )0),
          transfCoMod_codPair ( ~_2 @ F1 _o>CoMod^ z2z2' ) )%sol

| Pairing_Transf :
    ( forall (L : obCoMod) (F1 F2 : obCoMod),
        forall (f1 f1' : 'morCoMod(0 L ~> F1 )0) (f1f1' : 'transfCoMod(0 f1 ~> f1' )0),
        forall (f2 f2' : 'morCoMod(0 L ~> F2 )0) (f2f2' : 'transfCoMod(0 f2 ~> f2' )0),
          transfCoMod_codPair ( << f1f1' ,^CoMod f2f2' >> ) )%sol .

Definition transfCoMod_codPairP_Type
           (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 %sol)
           (gg' : 'transfCoMod(0 g ~> g' )0 %sol) :=
  ltac:( destruct G;  refine (transfCoMod_codPair gg')  ).

Lemma transfCoMod_codPairP
  : forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 %sol)
      (gg' : 'transfCoMod(0 g ~> g' )0 %sol),
    transfCoMod_codPairP_Type gg' .
Proof.
  intros. case: F G g g' / gg' .
  - intros ? G. destruct G; constructor 1.
  - intros ? ? Z1. destruct Z1. constructor 2.
  - intros ? ? Z2. destruct Z2. constructor 3.
  - constructor 4.
Defined.

End Destruct_codPair.

(** because of the cut-adhesive , therefore this below [Destruct_domProject1_Mor]
and similar destruction lemmas will not be used ... *)
Module Destruct_domProject1_Mor.

Inductive transfCoMod_domProject1_Mor
: forall (F1 F2 : obCoMod) (Z1 : obCoMod) (z1 : 'morCoMod(0 F1 ~> Z1 )0 %sol),
    forall (g : 'morCoMod(0 Pair F1 F2 ~> Z1 )0 %sol),
      'transfCoMod(0 (Project1_Mor F2 z1) ~> g )0 %sol -> Type :=

| UnitTransfCoMod :
    forall (F1 F2 : obCoMod) (Z1 : obCoMod) (z1 : 'morCoMod(0 F1 ~> Z1 )0 %sol),
     transfCoMod_domProject1_Mor ( @'UnitTransfCoMod ( ~_1 @ F2 o>CoMod z1 ) )%sol

| Project1_Transf : forall (F1 F2 : obCoMod) (Z1 : obCoMod),
    forall (z1 z1' : 'morCoMod(0 F1 ~> Z1 )0 %sol)
      (z1z1' : 'transfCoMod(0 z1 ~> z1' )0 %sol),
      transfCoMod_domProject1_Mor ( ~_1 @ F2 _o>CoMod^ z1z1' )%sol .

Definition transfCoMod_domProject1_MorP_Type
           (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 %sol)
           (gg' : 'transfCoMod(0 g ~> g' )0 %sol) :=
  ltac:( destruct g; [ intros; refine (unit : Type)
                     | refine (transfCoMod_domProject1_Mor gg')
                     | intros; refine (unit : Type)
                     | intros; refine (unit : Type) ] ).

Lemma transfCoMod_domProject1_MorP
  : forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 %sol)
      (gg' : 'transfCoMod(0 g ~> g' )0 %sol),
    transfCoMod_domProject1_MorP_Type gg' .
Proof.
  intros. case: F G g g' / gg' .
  - destruct g; [ intros; exact: tt | | intros; exact: tt ..] ;
    constructor 1.
  - constructor 2.
```

```
  - intros; exact: tt.
  - intros; exact: tt.
Defined.

End Destruct_domProject1_Mor.

End Sol.
```

8 Grammatical 2-conversion of transformations , which infer the 1-conversions of their domain-codomain morphisms

As common , the grammatical 1-conversions-for-morphisms [convMorCoMod] ans 2-conversions-for-transformations [convTransfCoMod] are classified into : the total/(multi-step) conversions , and the congruences conversions , and the constant conversions which are used in the polymorphism/cut-elimination lemma , and the constant conversions which are only for the wanted sense of pairing-projections-grammar , and the constant conversions which are only for the confluence lemma , and the constant conversions which are derivable by using the finished cut-elimination lemma .

In contrast , because of the structural-multiplying-arrow ("degeneracy") action which is the unit-transformation-on-each-morphism [UnitTransfCoMod] type-constructor (which is elsewhere also hidden/blended in the outer left/right-whisk cut constructors [TransfCoMod_PolyMorCoMod_Pre] [TransfCoMod_PolyMorCoMod_Post]) , then the 2-conversions-for-transformations [convTransfCoMod] depends/uses of the 1-conversions-for-morphisms [convMorCoMod] , via the conversion-constructors [UnitTransfCoMod_cong] [TransfCoMod_PolyMorCoMod_Pre_cong] [TransfCoMod_PolyMorCoMod_Post_cong] .

8.1 Grammatical 2-conversions of transformations

```
Reserved Notation "g'_g_ <~~2 g'g" (at level 70).

Inductive convTransfCoMod :
  forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 %poly)
    (gg' : 'transfCoMod(0 g ~> g' )0 %poly) (g_ g'_ : 'morCoMod(0 F ~> G )0 %poly)
    (g_g'_ : 'transfCoMod(0 g_ ~> g'_ )0 %poly) , Prop :=

(**  ----- the total/(multi-step) conversions -----  **)

| convTransfCoMod_Refl : forall (F G : obCoMod) (g : 'morCoMod(0 F ~> G )0 )
                            (gg : 'transfCoMod(0 g ~> g )0) ,
    gg <~~2 gg

| convTransfCoMod_Trans :
    forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 %poly)
      (gg' : 'transfCoMod(0 g ~> g' )0 %poly)
      (g0 g'0 : 'morCoMod(0 F ~> G )0 ) (uTrans : 'transfCoMod(0 g0 ~> g'0 )0 ),
      uTrans <~~2 gg' ->
  forall (g00 g'00 : 'morCoMod(0 F ~> G )0 ) (g00g'00 : 'transfCoMod(0 g00 ~> g'00 )0 ),
    g00g'00 <~~2 uTrans -> g00g'00 <~~2 gg'

(**  ----- the congruences conversions -----  **)

| PolyTransfCoMod_cong : forall (F G : obCoMod),
    forall (g g' : 'morCoMod(0 F ~> G )0) (g'g : 'transfCoMod(0 g' ~> g )0),
    forall (g'0 g'' : 'morCoMod(0 F ~> G )0)
      (g''g' : 'transfCoMod(0 g'' ~> g'0 )0) eqMor,
    forall (g_ g'_ : 'morCoMod(0 F ~> G )0) (g'_g_ : 'transfCoMod(0 g'_ ~> g_ )0),
    forall (g'_0 g''_ : 'morCoMod(0 F ~> G )0)
      (g''_g'_ : 'transfCoMod(0 g''_ ~> g'_0 )0) eqMor0,
      g''_g'_ <~~2 g''g' -> g'_g_ <~~2 g'g ->
      ( g''_g'_ o^CoMod g'_g_ # eqMor0 ) <~~2 ( g''g' o^CoMod g'g # eqMor)

| TransfCoMod_PolyMorCoMod_Pre_cong : forall (F G : obCoMod),
    forall (g g' : 'morCoMod(0 F ~> G )0) (g'g : 'transfCoMod(0 g' ~> g )0),
    forall (g_ g'_ : 'morCoMod(0 F ~> G )0) (g'_g_ : 'transfCoMod(0 g'_ ~> g_ )0),
    forall (E : obCoMod) (f f_ : 'morCoMod(0 E ~> F )0),
```

```
     f_ <~~1 f -> g'_g_ <~~2 g'g -> ( f_ _o>CoMod^ g'_g_ ) <~~2 ( f _o>CoMod^ g'g )

| TransfCoMod_PolyMorCoMod_Post_cong :
    forall (G H : obCoMod) (h h_ : 'morCoMod(0 G ~> H )0), forall (F : obCoMod),
        forall (g g' : 'morCoMod(0 F ~> G )0) (g'g : 'transfCoMod(0 g' ~> g )0),
        forall (g_ g'_ : 'morCoMod(0 F ~> G )0) (g'_g_ : 'transfCoMod(0 g'_ ~> g_ )0),
     h_ <~~1 h -> g'_g_ <~~2 g'g -> ( g'_g_ ^o>CoMod_ h_ ) <~~2 ( g'g ^o>CoMod_ h )

| UnitTransfCoMod_cong : forall (F G : obCoMod), forall (g g' : 'morCoMod(0 F ~> G )0),
      g' <~~1 g -> (@'UnitTransfCoMod g') <~~2 (@'UnitTransfCoMod g)

| Project1_Transf_cong : forall (F1 F2 : obCoMod) (Z1 : obCoMod),
    forall (z1 z1' : 'morCoMod(0 F1 ~> Z1 )0) (z1z1' : 'transfCoMod(0 z1 ~> z1' )0),
 forall (z1_ z1'_ : 'morCoMod(0 F1 ~> Z1 )0) (z1_z1'_ : 'transfCoMod(0 z1_ ~> z1'_ )0),
   z1_z1'_ <~~2 z1z1' ->
   ( ~_1 @ F2 _o>CoMod^ z1_z1'_ ) <~~2 ( ~_1 @ F2 _o>CoMod^ z1z1' )

| Project2_Transf_cong : forall (F1 F2 : obCoMod) (Z2 : obCoMod),
    forall (z2 z2' : 'morCoMod(0 F2 ~> Z2 )0) (z2z2' : 'transfCoMod(0 z2 ~> z2' )0),
 forall (z2_ z2'_ : 'morCoMod(0 F2 ~> Z2 )0) (z2_z2'_ : 'transfCoMod(0 z2_ ~> z2'_ )0),
   z2_z2'_ <~~2 z2z2' ->
   ( ~_2 @ F1 _o>CoMod^ z2_z2'_ ) <~~2 ( ~_2 @ F1 _o>CoMod^ z2z2' )

| Pairing_Transf_cong : forall (L : obCoMod) (F1 F2 : obCoMod),
    forall (f1 f1' : 'morCoMod(0 L ~> F1 )0) (f1f1' : 'transfCoMod(0 f1 ~> f1' )0),
  forall (f1_ f1'_ : 'morCoMod(0 L ~> F1 )0) (f1_f1'_ : 'transfCoMod(0 f1_ ~> f1'_ )0),
  forall (f2 f2' : 'morCoMod(0 L ~> F2 )0) (f2f2' : 'transfCoMod(0 f2 ~> f2' )0),
  forall (f2_ f2'_ : 'morCoMod(0 L ~> F2 )0) (f2_f2'_ : 'transfCoMod(0 f2_ ~> f2'_ )0),
    f1_f1'_ <~~2 f1f1' -> f2_f2'_ <~~2 f2f2' ->
    ( << f1_f1'_ ,^CoMod f2_f2'_ >> ) <~~2 ( << f1f1' ,^CoMod f2f2' >> )

(** ----- the constant conversions which are used during the polyarrowing
elimination ----- **)

(** here there are none parameter/customized-arrow action ( non-structural
reindexing , customized boundary-or-degeneracy ) ; is it possible to make sense of
such ? *)

(** ----- the constant conversions which are used during the polymorphism
elimination ----- **)

(* in other words : this polymorphisms also as structural-polyarrowing along the (
"degeneracy" ) structural-multiplying-arrow {0 -> 1} |- {0} *)
| UnitTransfCoMod_morphismMor_Pre :
    forall (F G : obCoMod), forall (g : 'morCoMod(0 F ~> G )0),
        forall (E : obCoMod) (f : 'morCoMod(0 E ~> F )0),
          ( @'UnitTransfCoMod (f o>CoMod g)
            : 'transfCoMod(0 f o>CoMod g ~> f o>CoMod g )0 )
            <~~2 ( f _o>CoMod^ ( @'UnitTransfCoMod g )
                  : 'transfCoMod(0 f o>CoMod g ~> f o>CoMod g )0 )

(* in other words : this polymorphisms also as structural-polyarrowing along the (
"degeneracy" ) structural-multiplying-arrow {0 -> 1} |- {0} *)
| UnitTransfCoMod_morphismMor_Post :
    forall (G H : obCoMod) (h : 'morCoMod(0 G ~> H )0),
    forall (F : obCoMod), forall (g : 'morCoMod(0 F ~> G )0),
        ( @'UnitTransfCoMod (g o>CoMod h)
          : 'transfCoMod(0 g o>CoMod h ~> g o>CoMod h )0 )
          <~~2 ( ( @'UnitTransfCoMod g ) ^o>CoMod_ h
                : 'transfCoMod(0 g o>CoMod h ~> g o>CoMod h )0 )

| UnitTransfCoMod_morphismTransf_Pre :
    forall (F G : obCoMod), forall (g' : 'morCoMod(0 F ~> G )0),
 forall (g'0 g'' : 'morCoMod(0 F ~> G )0) (g''g'0 : 'transfCoMod(0 g'' ~> g'0 )0) eqMor,
   ( g''g'0 ) <~~2 ( (  g''g'0 o^CoMod ( @'UnitTransfCoMod  g' ) # eqMor )
                  : 'transfCoMod(0  g'' ~>  g' )0 )
```

```
| UnitTransfCoMod_morphismTransf_Post : forall (F G : obCoMod),
    forall (g g' : 'morCoMod(0 F ~> G )0) (g'g : 'transfCoMod(0 g' ~> g )0)
      (g'0 : 'morCoMod(0 F ~> G )0) eqMor,
      ( g'g ) <~~2 ( ( @'UnitTransfCoMod g'0 ) o^CoMod g'g # eqMor
                   : 'transfCoMod(0 g'0 ~> g )0 )

| UnitMorCoMod_morphismTransf_Pre :
forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0) (g'g : 'transfCoMod(0 g' ~> g )0),
  ( g'g
    : 'transfCoMod(0 g' ~> g )0 )
    <~~2 ( g'g ^o>CoMod_ ( @'UnitMorCoMod G )
         : 'transfCoMod(0 g' o>CoMod 'UnitMorCoMod ~> g o>CoMod 'UnitMorCoMod )0 )

| UnitMorCoMod_morphismTransf_Post :
forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0) (g'g : 'transfCoMod(0 g' ~> g )0),
  ( g'g
    : 'transfCoMod(0 g' ~> g )0 )
    <~~2 ( ( @'UnitMorCoMod F ) _o>CoMod^ g'g
         : 'transfCoMod(0 'UnitMorCoMod o>CoMod g' ~> 'UnitMorCoMod o>CoMod g )0 )

| Project1_Mor_morphismTransf :
    forall (F1 F2 : obCoMod) (Z1 : obCoMod) (z1 : 'morCoMod(0 F1 ~> Z1 )0),
    forall (Y1 : obCoMod) (y' y : 'morCoMod(0 Z1 ~> Y1 )0)
      (y'y : 'transfCoMod(0 y' ~> y )0),
 ( ~_1 @ F2 _o>CoMod^ (z1 _o>CoMod^ y'y)
   : 'transfCoMod(0 ~_1 o>CoMod (z1 o>CoMod y') ~> ~_1 o>CoMod (z1 o>CoMod y) )0 )
   <~~2 ( ( ~_1 @ F2 o>CoMod z1 ) _o>CoMod^ y'y
: 'transfCoMod(0 ( ~_1 o>CoMod z1 ) o>CoMod y' ~> ( ~_1 o>CoMod z1 ) o>CoMod y )0 )

| Project2_Mor_morphismTransf :
    forall (F1 F2 : obCoMod) (Z2 : obCoMod) (z2 : 'morCoMod(0 F2 ~> Z2 )0),
forall (Y2 : obCoMod) (y' y : 'morCoMod(0 Z2 ~> Y2 )0) (y'y : 'transfCoMod(0 y' ~> y )0),
  ( ~_2 @ F1 _o>CoMod^ (z2 _o>CoMod^ y'y)
   : 'transfCoMod(0 ~_2 o>CoMod (z2 o>CoMod y') ~> ~_2 o>CoMod (z2 o>CoMod y) )0 )
    <~~2 ( ( ~_2 @ F1 o>CoMod z2 ) _o>CoMod^ y'y
: 'transfCoMod(0 ( ~_2 o>CoMod z2 ) o>CoMod y' ~> ( ~_2 o>CoMod z2 ) o>CoMod y )0 )

| Project1_Transf_morphismMor : forall (F1 F2 : obCoMod) (Z1 : obCoMod),
    forall (z1 z1' : 'morCoMod(0 F1 ~> Z1 )0) (z1z1' : 'transfCoMod(0 z1 ~> z1' )0),
    forall (Y1 : obCoMod) (y : 'morCoMod(0 Z1 ~> Y1 )0),
( ~_1 @ F2 _o>CoMod^ (z1z1' ^o>CoMod_ y)
  : 'transfCoMod(0 ~_1 o>CoMod (z1 o>CoMod y) ~> ~_1 o>CoMod (z1' o>CoMod y) )0 )
  <~~2 ( ( ~_1 @ F2 _o>CoMod^ z1z1' ) ^o>CoMod_ y
: 'transfCoMod(0 ( ~_1 o>CoMod z1 ) o>CoMod y ~> ( ~_1 o>CoMod z1' ) o>CoMod y )0 )

| Project2_Transf_morphismMor : forall (F1 F2 : obCoMod) (Z2 : obCoMod),
    forall (z2 z2' : 'morCoMod(0 F2 ~> Z2 )0) (z2z2' : 'transfCoMod(0 z2 ~> z2' )0),
    forall (Y2 : obCoMod) (y : 'morCoMod(0 Z2 ~> Y2 )0),
( ~_2 @ F1 _o>CoMod^ (z2z2' ^o>CoMod_ y)
  : 'transfCoMod(0 ~_2 o>CoMod (z2 o>CoMod y) ~> ~_2 o>CoMod (z2' o>CoMod y) )0 )
  <~~2 ( ( ~_2 @ F1 _o>CoMod^ z2z2' ) ^o>CoMod_ y
: 'transfCoMod(0 ( ~_2 o>CoMod z2 ) o>CoMod y ~> ( ~_2 o>CoMod z2' ) o>CoMod y )0 )

| Project1_Transf_morphismTransf : forall (F1 F2 : obCoMod) (Z1 : obCoMod),
    forall (z1 z1' : 'morCoMod(0 F1 ~> Z1 )0) (z1z1' : 'transfCoMod(0 z1 ~> z1' )0),
forall (z1'0 z1'' : 'morCoMod(0 F1 ~> Z1 )0) (z1'z1'' : 'transfCoMod(0 z1'0 ~> z1'' )0),
forall (eqMor_param : ( (~_1 o>CoMod z1') <~>1 (~_1 o>CoMod z1'0) ))
  (eqMor_ex : ( z1' <~>1  z1'0 )),
  ( ~_1 @ F2 _o>CoMod^ (z1z1' o^CoMod z1'z1'' # eqMor_ex)
    : 'transfCoMod(0 ~_1 o>CoMod z1 ~> ~_1 o>CoMod z1'' )0 )
    <~~2 ( ( ~_1 @ F2 _o>CoMod^ z1z1' )
           o^CoMod ( ~_1 @ F2 _o>CoMod^ z1'z1'' ) # eqMor_param
         : 'transfCoMod(0 ~_1 o>CoMod z1 ~> ~_1 o>CoMod z1'' )0 )

| Project2_Transf_morphismTransf : forall (F1 F2 : obCoMod) (Z2 : obCoMod),
    forall (z2 z2' : 'morCoMod(0 F2 ~> Z2 )0) (z2z2' : 'transfCoMod(0 z2 ~> z2' )0),
forall (z2'0 z2'' : 'morCoMod(0 F2 ~> Z2 )0) (z2'z2'' : 'transfCoMod(0 z2'0 ~> z2'' )0),
```

```
forall eqMor_param eqMor_ex,
  ( ~_2 @ F1 _o>CoMod^ (z2z2' o^CoMod z2'z2'' # eqMor_ex)
    : 'transfCoMod(0 ~_2 o>CoMod z2 ~> ~_2 o>CoMod z2'' )0 )
    <~~2 ( ( ~_2 @ F1 _o>CoMod^ z2z2' )
           o^CoMod ( ~_2 @ F1 _o>CoMod^ z2'z2'' ) # eqMor_param
         : 'transfCoMod(0 ~_2 o>CoMod z2 ~> ~_2 o>CoMod z2'' )0 )

(**memo: Pairing_Mor_morphismTransf_derivable below *)
| Pairing_Mor_morphismTransf :
    forall (L1 L2 : obCoMod) (F1 F2 : obCoMod) (f1 : 'morCoMod(0 Pair L1 L2 ~> F1 )0)
      (f2 :'morCoMod(0 Pair L1 L2 ~> F2 )0),
    forall (M : obCoMod) (l1 l1' : 'morCoMod(0 M ~> L1 )0)
      (l1l1' : 'transfCoMod(0 l1 ~> l1' )0)
      (l2 l2' : 'morCoMod(0 M ~> L2 )0) (l2l2' : 'transfCoMod(0 l2 ~> l2' )0),
      ( << ( ( << l1l1' ,^CoMod l2l2' >> ) ^o>CoMod_ f1 )
           ,^CoMod ( ( << l1l1' ,^CoMod l2l2' >> ) ^o>CoMod_ f2 ) >>
        : 'transfCoMod(0 << ( ( << l1 ,CoMod l2 >> ) o>CoMod f1)
                       ,CoMod ( ( << l1 ,CoMod l2 >> ) o>CoMod f2) >> ~>
                              << ( ( << l1' ,CoMod l2' >> ) o>CoMod f1)
                       ,CoMod ( ( << l1' ,CoMod l2' >> ) o>CoMod f2) >> )0 )
        <~~2 ( ( << l1l1' ,^CoMod l2l2' >> ) ^o>CoMod_ ( << f1 ,CoMod f2 >> )
             : 'transfCoMod(0 ( << l1 ,CoMod l2 >> ) o>CoMod << f1 ,CoMod f2 >> ~>
                          ( << l1' ,CoMod l2' >> ) o>CoMod << f1 ,CoMod f2 >> )0 )

(**memo: Pairing_Transf_morphismMor_derivable below *)
| Pairing_Transf_morphismMor : forall (L1 L2 : obCoMod) (F1 F2 : obCoMod),
forall (f1 f1' : 'morCoMod(0 Pair L1 L2 ~> F1 )0) (f1f1' : 'transfCoMod(0 f1 ~> f1' )0),
forall (f2 f2' : 'morCoMod(0 Pair L1 L2 ~> F2 )0) (f2f2' : 'transfCoMod(0 f2 ~> f2' )0),
forall (M : obCoMod) (l1 : 'morCoMod(0 M ~> L1 )0) (l2 : 'morCoMod(0 M ~> L2 )0),
  ( << ( ( << l1 ,CoMod l2 >> ) _o>CoMod^ f1f1' )
       ,^CoMod ( ( << l1 ,CoMod l2 >> ) _o>CoMod^ f2f2' ) >>
    : 'transfCoMod(0 << ( ( << l1 ,CoMod l2 >> ) o>CoMod f1)
                   ,CoMod ( ( << l1 ,CoMod l2 >> ) o>CoMod f2) >> ~>
                          << ( ( << l1 ,CoMod l2 >> ) o>CoMod f1')
                   ,CoMod ( ( << l1 ,CoMod l2 >> ) o>CoMod f2') >> )0 )
    <~~2 ( ( << l1 ,CoMod l2 >> ) _o>CoMod^ ( << f1f1' ,^CoMod f2f2' >> )
         : 'transfCoMod(0 ( << l1 ,CoMod l2 >> ) o>CoMod << f1 ,CoMod f2 >> ~>
                          ( << l1 ,CoMod l2 >> ) o>CoMod << f1' ,CoMod f2' >> )0 )

| Pairing_Transf_morphismTransf : forall (L : obCoMod) (F1 F2 : obCoMod),
    forall (f1 f1' : 'morCoMod(0 L ~> F1 )0) (f1f1' : 'transfCoMod(0 f1 ~> f1' )0),
    forall (f2 f2' : 'morCoMod(0 L ~> F2 )0) (f2f2' : 'transfCoMod(0 f2 ~> f2' )0),
forall (f1'0 f1'' : 'morCoMod(0 L ~> F1 )0) (f1'f1'' : 'transfCoMod(0 f1'0 ~> f1'' )0),
forall (f2'0 f2'' : 'morCoMod(0 L ~> F2 )0) (f2'f2'' : 'transfCoMod(0 f2'0 ~> f2'' )0),
forall eqMor1_ex eqMor2_ex eqMor_param,
( << f1f1' o^CoMod f1'f1'' # eqMor1_ex ,^CoMod f2f2' o^CoMod f2'f2'' # eqMor2_ex >>
  : 'transfCoMod(0 << f1 ,CoMod f2 >> ~> << f1'' ,CoMod f2'' >> )0 )
  <~~2 ( ( << f1f1' ,^CoMod f2f2' >> )
         o^CoMod ( << f1'f1'' ,^CoMod f2'f2'' >> ) # eqMor_param
       : 'transfCoMod(0 << f1 ,CoMod f2 >> ~> << f1'' ,CoMod f2'' >> )0 )

| Pairing_Transf_Project1_Mor : forall (L : obCoMod) (F1 F2 : obCoMod),
    forall (f1 f1' : 'morCoMod(0 L ~> F1 )0) (f1f1' : 'transfCoMod(0 f1 ~> f1' )0),
    forall (f2 f2' : 'morCoMod(0 L ~> F2 )0) (f2f2' : 'transfCoMod(0 f2 ~> f2' )0),
    forall (Z1 : obCoMod) (z1 : 'morCoMod(0 F1 ~> Z1 )0),
      ( f1f1' ^o>CoMod_ z1
        : 'transfCoMod(0 f1 o>CoMod z1 ~> f1' o>CoMod z1 )0 )
        <~~2 ( ( << f1f1' ,^CoMod f2f2' >> ) ^o>CoMod_ ( ~_1 @ F2 o>CoMod z1 )
             : 'transfCoMod(0 << f1 ,CoMod f2 >> o>CoMod ~_1 o>CoMod z1 ~>
                              << f1' ,CoMod f2' >> o>CoMod ~_1 o>CoMod z1 )0 )

| Pairing_Transf_Project2_Mor : forall (L : obCoMod) (F1 F2 : obCoMod),
    forall (f1 f1' : 'morCoMod(0 L ~> F1 )0) (f1f1' : 'transfCoMod(0 f1 ~> f1' )0),
    forall (f2 f2' : 'morCoMod(0 L ~> F2 )0) (f2f2' : 'transfCoMod(0 f2 ~> f2' )0),
    forall (Z2 : obCoMod) (z2 : 'morCoMod(0 F2 ~> Z2 )0),
      ( f2f2' ^o>CoMod_ z2
        : 'transfCoMod(0 f2 o>CoMod z2 ~> f2' o>CoMod z2 )0 )
```

```
        <~~2 ( ( << f1f1' ,^CoMod f2f2' >> ) ^o>CoMod_ ( ~_2 @ F1 o>CoMod z2 )
             : 'transfCoMod(0 << f1 ,CoMod f2 >> o>CoMod ~_2 o>CoMod z2 ~>
                              << f1' ,CoMod f2' >> o>CoMod ~_2 o>CoMod z2 )0 )

| Pairing_Mor_Project1_Transf :
    forall (L : obCoMod) (F1 F2 : obCoMod) (f1 : 'morCoMod(0 L ~> F1 )0)
      (f2 : 'morCoMod(0 L ~> F2 )0),
    forall (Z1 : obCoMod) (z1 z1' : 'morCoMod(0 F1 ~> Z1 )0)
      (z1z1' : 'transfCoMod(0 z1 ~> z1' )0),
      ( f1 _o>CoMod^ z1z1'
        : 'transfCoMod(0 f1 o>CoMod z1 ~> f1 o>CoMod z1' )0 )
        <~~2 ( ( << f1 ,CoMod f2 >> ) _o>CoMod^ ( ~_1 @ F2 _o>CoMod^ z1z1' )
             : 'transfCoMod(0 << f1 ,CoMod f2 >> o>CoMod ~_1 o>CoMod z1 ~>
                              << f1 ,CoMod f2 >> o>CoMod ~_1 o>CoMod z1' )0 )

| Pairing_Mor_Project2_Transf :
    forall (L : obCoMod) (F1 F2 : obCoMod) (f1 : 'morCoMod(0 L ~> F1 )0)
      (f2 : 'morCoMod(0 L ~> F2 )0),
    forall (Z2 : obCoMod) (z2 z2' : 'morCoMod(0 F2 ~> Z2 )0)
      (z2z2' : 'transfCoMod(0 z2 ~> z2' )0),
      ( f2 _o>CoMod^ z2z2'
        : 'transfCoMod(0 f2 o>CoMod z2 ~> f2 o>CoMod z2' )0 )
        <~~2 ( ( << f1 ,CoMod f2 >> ) _o>CoMod^ ( ~_2 @ F1 _o>CoMod^ z2z2' )
             : 'transfCoMod(0 << f1 ,CoMod f2 >> o>CoMod ~_2 o>CoMod z2 ~>
                              << f1 ,CoMod f2 >> o>CoMod ~_2 o>CoMod z2' )0 )

(** ----- the constant conversions which are only for the wanted sense of
pairing-projections-grammar ----- **)

(** Attention : for non-contextual ( "1-weigthed" ) pairing-projections , none of
    such thing as [Project1_Mor_Project2_Mor_Pairing_Transf] for transformations
    instead of [Project1_Mor_Project2_Mor_Pairing_Mor] for morphisms **)

(* structural-polyarrowing along the ( "degeneracy" ) structural-multiplying-arrow
{0 -> 1} |- {0} *)
(*TODO: shall reverse this conversion ? *)
| Project1_UnitTransfCoMod :
    forall (F1 F2 Z1 : obCoMod) (z1 : 'morCoMod(0 F1 ~> Z1 )0),
      ( ~_1 @ F2 _o>CoMod^ ( @'UnitTransfCoMod z1 : 'transfCoMod(0 z1 ~> z1 )0 )
        : 'transfCoMod(0 ~_1 @ F2 o>CoMod z1 ~>
                ( ~_1 @ F2 o>CoMod z1 : 'morCoMod(0 Pair F1 F2 ~> Z1 )0 ) )0 )
        <~~2 ( @'UnitTransfCoMod ( ~_1 @ F2 o>CoMod z1 ) )

(* structural-polyarrowing along the ( "degeneracy" ) structural-multiplying-arrow
{0 -> 1} |- {0} *)
(*TODO: shall reverse this conversion ? *)
| Project2_UnitTransfCoMod : forall (F1 F2 : obCoMod) (Z2 : obCoMod)
    (z2 : 'morCoMod(0 F2 ~> Z2 )0),
    ( ~_2 @ F1 _o>CoMod^ ( @'UnitTransfCoMod z2 )
      : 'transfCoMod(0 ~_2 o>CoMod z2 ~> ~_2 o>CoMod z2 )0 )
      <~~2 ( @'UnitTransfCoMod ( ~_2 @ F1 o>CoMod z2 )
           : 'transfCoMod(0 ~_2 o>CoMod z2 ~> ~_2 o>CoMod z2 )0 )

(* structural-polyarrowing along the ( "degeneracy" ) structural-multiplying-arrow
{0 -> 1} |- {0} *)
(*TODO: shall reverse this conversion ?*)
| Pairing_UnitTransfCoMod : forall (L : obCoMod) (F1 F2 : obCoMod),
    forall (f1 : 'morCoMod(0 L ~> F1 )0), forall (f2 : 'morCoMod(0 L ~> F2 )0),
        ( << ( @'UnitTransfCoMod f1 ) ,^CoMod ( @'UnitTransfCoMod f2 ) >>
           : 'transfCoMod(0 << f1 ,CoMod f2 >> ~> << f1 ,CoMod f2 >> )0 )
          <~~2 ( @'UnitTransfCoMod << f1 ,CoMod f2 >>
               : 'transfCoMod(0 << f1 ,CoMod f2 >> ~> << f1 ,CoMod f2 >> )0 )

(** ----- the constant conversions which are only for the confluence lemma --- **)

| Pairing_Transf_morphism_Project1_Transf : forall (L : obCoMod) (F1 F2 : obCoMod),
    forall (f1 f1' : 'morCoMod(0 L ~> F1 )0) (f1f1' : 'transfCoMod(0 f1 ~> f1' )0),
```

```
    forall (f2 f2' : 'morCoMod(0 L ~> F2 )0) (f2f2' : 'transfCoMod(0 f2 ~> f2' )0),
    forall (H : obCoMod),
      ( ~_1 @ H _o>CoMod^ ( << f1f1' ,^CoMod f2f2' >> )
        : 'transfCoMod(0 ~_1 o>CoMod ( << f1 ,CoMod f2 >> ) ~>
                         ~_1 o>CoMod ( << f1' ,CoMod f2' >> ) )0 )
      <~~2 ( << ( ~_1 @ H _o>CoMod^ f1f1' )
               ,^CoMod ( ~_1 @ H _o>CoMod^ f2f2' ) >>
           : 'transfCoMod(0 << ( ~_1 o>CoMod f1 ) ,CoMod ( ~_1 o>CoMod f2 ) >> ~>
                          << ( ~_1 o>CoMod f1' ) ,CoMod ( ~_1 o>CoMod f2' ) >> )0 )

| Pairing_Transf_morphism_Project2_Transf : forall (L : obCoMod) (F1 F2 : obCoMod),
    forall (f1 f1' : 'morCoMod(0 L ~> F1 )0) (f1f1' : 'transfCoMod(0 f1 ~> f1' )0),
    forall (f2 f2' : 'morCoMod(0 L ~> F2 )0) (f2f2' : 'transfCoMod(0 f2 ~> f2' )0),
    forall (H : obCoMod),
      ( ~_2 @ H _o>CoMod^ ( << f1f1' ,^CoMod f2f2' >> )
        : 'transfCoMod(0 ~_2 o>CoMod ( << f1 ,CoMod f2 >> ) ~>
                         ~_2 o>CoMod ( << f1' ,CoMod f2' >> ) )0 )
      <~~2 ( << ( ~_2 @ H _o>CoMod^ f1f1' )
               ,^CoMod ( ~_2 @ H _o>CoMod^ f2f2' ) >>
           : 'transfCoMod(0 << ( ~_2 o>CoMod f1 ) ,CoMod ( ~_2 o>CoMod f2 ) >> ~>
                          << ( ~_2 o>CoMod f1' ) ,CoMod ( ~_2 o>CoMod f2' ) >> )0 )

(** ----- the constant conversions which are derivable by using the finished
cut-elimination lemma ----- **)

(**
(*TODO: COMMENT *)
| PolyTransCoMod_morphism : forall (F G : obCoMod),
    forall (g g' : 'morCoMod(0 F ~> G )0) (g'g : 'transfCoMod(0 g' ~> g )0),
    forall (g'0 g'' : 'morCoMod(0 F ~> G )0)
      (g''g' : 'transfCoMod(0 g'' ~> g'0 )0) eqMor_param_in eqMor_ex_out,
  forall (g''0 g''' : 'morCoMod(0 F ~> G )0) (g'''g'' : 'transfCoMod(0 g''' ~> g''0 )0)
    eqMor_param_out eqMor_ex_in,
    ( ( g'''g'' o^CoMod g''g' # eqMor_ex_in ) o^CoMod g'g # eqMor_ex_out
      : 'transfCoMod(0 g''' ~> g )0 )
   <~~2 ( g'''g'' o^CoMod ( g''g' o^CoMod g'g # eqMor_param_in ) # eqMor_param_out
        : 'transfCoMod(0 g''' ~> g )0 )

(*TODO: COMMENT *)
| TransfCoMod_PolyMorCoMod_Pre_morphism_Pre : forall (F G : obCoMod),
    forall (g g' : 'morCoMod(0 F ~> G )0) (g'g : 'transfCoMod(0 g' ~> g )0),
    forall (E : obCoMod) (f : 'morCoMod(0 E ~> F )0),
    forall (D : obCoMod) (e : 'morCoMod(0 D ~> E )0),
      ( ( e o>CoMod f ) _o>CoMod^ g'g
        : 'transfCoMod(0 (e o>CoMod f) o>CoMod g' ~> (e o>CoMod f) o>CoMod g )0 )
        <~~2 ( e _o>CoMod^ ( f _o>CoMod^ g'g )
          : 'transfCoMod(0 e o>CoMod (f o>CoMod g') ~> e o>CoMod (f o>CoMod g) )0 )

(*TODO: COMMENT *)
| TransfCoMod_PolyMorCoMod_Pre_morphism_Post : forall (F G : obCoMod),
    forall (g g' : 'morCoMod(0 F ~> G )0) (g'g : 'transfCoMod(0 g' ~> g )0),
    forall (E : obCoMod) (f : 'morCoMod(0 E ~> F )0),
    forall (H : obCoMod) (h : 'morCoMod(0 G ~> H )0),
      ( f _o>CoMod^ ( g'g ^o>CoMod_ h )
        : 'transfCoMod(0 f o>CoMod (g' o>CoMod h) ~> f o>CoMod (g o>CoMod h) )0 )
        <~~2 ( ( f _o>CoMod^ g'g ) ^o>CoMod_ h
          : 'transfCoMod(0 (f o>CoMod g') o>CoMod h ~> (f o>CoMod g) o>CoMod h )0 )

(*TODO: COMMENT *)
| TransfCoMod_PolyMorCoMod_Post_morphism_Pre :
    forall (G H : obCoMod) (h : 'morCoMod(0 G ~> H )0), forall (F : obCoMod),
        forall (g g' : 'morCoMod(0 F ~> G )0) (g'g : 'transfCoMod(0 g' ~> g )0),
        forall (E : obCoMod) (f : 'morCoMod(0 E ~> F )0),
          ( ( f _o>CoMod^ g'g ) ^o>CoMod_ h
          : 'transfCoMod(0 (f o>CoMod g') o>CoMod h ~> (f o>CoMod g) o>CoMod h )0 )
            <~~2 ( f _o>CoMod^ ( g'g ^o>CoMod_ h )
          : 'transfCoMod(0 f o>CoMod (g' o>CoMod h) ~> f o>CoMod (g o>CoMod h) )0 )
```

```
(*TODO: COMMENT *)
| TransfCoMod_PolyMorCoMod_Post_morphism_Post :
    forall (G H : obCoMod) (h : 'morCoMod(0 G ~> H )0), forall (F : obCoMod),
        forall (g g' : 'morCoMod(0 F ~> G )0) (g'g : 'transfCoMod(0 g' ~> g )0),
        forall (I : obCoMod) (i : 'morCoMod(0 H ~> I )0),
          ( g'g ^o>CoMod_ ( h o>CoMod i )
          : 'transfCoMod(0 g' o>CoMod (h o>CoMod i) ~> g o>CoMod (h o>CoMod i) )0 )
            <~~2 ( ( g'g ^o>CoMod_ h ) ^o>CoMod_ i
          : 'transfCoMod(0 (g' o>CoMod h) o>CoMod i ~> (g o>CoMod h) o>CoMod i )0 )
**)

(*TODO: COMMENT *)
| TransfCoMod_PolyMorCoMod_Pre_morphismTransf : forall (F G : obCoMod),
    forall (g g' : 'morCoMod(0 F ~> G )0) (g'g : 'transfCoMod(0 g' ~> g )0),
    forall (E : obCoMod) (f : 'morCoMod(0 E ~> F )0),
    forall (g'0 g'' : 'morCoMod(0 F ~> G )0) (g''g' : 'transfCoMod(0 g'' ~> g'0 )0),
    forall eqMor_param eqMor_ex,
      ( f _o>CoMod^ ( g''g' o^CoMod g'g # eqMor_ex )
        : 'transfCoMod(0 f o>CoMod g'' ~> f o>CoMod g )0 )
        <~~2 ( ( f _o>CoMod^ g''g' ) o^CoMod ( f _o>CoMod^ g'g ) # eqMor_param
             : 'transfCoMod(0 f o>CoMod g'' ~> f o>CoMod g )0 )

(*TODO: COMMENT *)
| TransfCoMod_PolyMorCoMod_Post_morphismTransf :
    forall (G H : obCoMod) (h : 'morCoMod(0 G ~> H )0), forall (F : obCoMod),
        forall (g g' : 'morCoMod(0 F ~> G )0) (g'g : 'transfCoMod(0 g' ~> g )0),
        forall (g'0 g'' : 'morCoMod(0 F ~> G )0) (g''g' : 'transfCoMod(0 g'' ~> g'0 )0),
        forall eqMor_param eqMor_ex,
          ( ( g''g' o^CoMod g'g # eqMor_ex ) ^o>CoMod_ h
            : 'transfCoMod(0 g'' o>CoMod h ~> g o>CoMod h )0 )
            <~~2 ( ( g''g' ^o>CoMod_ h ) o^CoMod ( g'g ^o>CoMod_ h ) # eqMor_param
                 : 'transfCoMod(0 g'' o>CoMod h ~> g o>CoMod h )0 )

(** ----- the constant conversions which are derivable immediately without the
finished cut-elimination lemma ----- **)

(**
(*TODO: COMMENT *)
| Pairing_Mor_morphismTransf_derivable :
    forall (L : obCoMod) (F1 F2 : obCoMod) (f1 : 'morCoMod(0 L ~> F1 )0)
      (f2 :'morCoMod(0 L ~> F2 )0),
    forall (L' : obCoMod) (l l0 : 'morCoMod(0 L' ~> L )0)
      (ll0 : 'transfCoMod(0 l ~> l0 )0),
      ( << ( ll0 ^o>CoMod_ f1 ) ,^CoMod ( ll0 ^o>CoMod_ f2 ) >>
        : 'transfCoMod(0 << (l o>CoMod f1) ,CoMod (l o>CoMod f2) >> ~>
                         << (l0 o>CoMod f1) ,CoMod (l0 o>CoMod f2) >> )0 )
        <~~2 ( ll0 ^o>CoMod_ ( << f1 ,CoMod f2 >> )
             : 'transfCoMod(0 l o>CoMod << f1 ,CoMod f2 >> ~>
                              l0 o>CoMod << f1 ,CoMod f2 >> )0 )

(*TODO: COMMENT *)
| Pairing_Transf_morphismMor_derivable : forall (L : obCoMod) (F1 F2 : obCoMod),
    forall (f1 f1' : 'morCoMod(0 L ~> F1 )0) (f1f1' : 'transfCoMod(0 f1 ~> f1' )0),
    forall (f2 f2' : 'morCoMod(0 L ~> F2 )0) (f2f2' : 'transfCoMod(0 f2 ~> f2' )0),
    forall (L' : obCoMod) (l : 'morCoMod(0 L' ~> L )0),
      ( << ( l _o>CoMod^ f1f1' ) ,^CoMod ( l _o>CoMod^ f2f2' ) >>
        : 'transfCoMod(0 << (l o>CoMod f1) ,CoMod (l o>CoMod f2) >> ~>
                         << (l o>CoMod f1') ,CoMod (l o>CoMod f2') >> )0 )
        <~~2 ( l _o>CoMod^ ( << f1f1' ,^CoMod f2f2' >> )
             : 'transfCoMod(0 l o>CoMod << f1 ,CoMod f2 >> ~>
                              l o>CoMod << f1' ,CoMod f2' >> )0 )
**)

where "g_g'_ <~~2 gg'" := (@convTransfCoMod _ _ _ _ gg' _ _ g_g'_) .

Hint Constructors convTransfCoMod.
```

8.2 1-convertibility of the domain/codomain morphisms for 2-convertible transformations

Because of the embedded/computed domain-codomain morphisms extra-argument/parameter in the inductive-family-presentation of the transformations , the 2-conversion-for-transformations relation shall convert across two transformations whose domain-codomain-morphisms-computation arguments are not syntactically/grammatically-the-same . But oneself does show that , by logical-deduction [convTransfCoMod_convMorCoMod_dom] [convTransfCoMod_convMorCoMod_cod] , these two domain-codomain-morphisms are indeed 1-convertible ("soundness lemma") .

```
Section NotUsed.

Lemma convTransfCoMod_convMorCoMod_dom :
  forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 %poly)
    (gg' : 'transfCoMod(0 g ~> g' )0 %poly) (g_ g'_ : 'morCoMod(0 F ~> G )0 %poly)
    (g_g'_ : 'transfCoMod(0 g_ ~> g'_ )0 %poly),
    g_g'_ <~~2 gg' -> g_ <~~1 g .
Proof.
  induction 1 ; try solve [eauto];
    match goal with
    | [eqMor : ( _ <~>1 _ )%poly |- _] =>
      move: (EqMorCoMod.convMorCoMod_eq eqMor)
    end;
    intros; subst; eauto.
Qed.

Lemma convTransfCoMod_convMorCoMod_cod :
  forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 %poly)
    (gg' : 'transfCoMod(0 g ~> g' )0 %poly) (g_ g'_ : 'morCoMod(0 F ~> G )0 %poly)
    (g_g'_ : 'transfCoMod(0 g_ ~> g'_ )0 %poly),
    g_g'_ <~~2 gg' -> g'_ <~~1 g' .
Proof.
  induction 1 ; try solve [eauto];
    match goal with
    | [eqMor : ( _ <~>1 _ )%poly |- _] =>
      move: (EqMorCoMod.convMorCoMod_eq eqMor)
    end;
    intros; subst; eauto.
Qed.

End NotUsed.
```

8.3 Linear total/asymptotic transformation-grade and the degradation lemma

```
Fixpoint gradeTransf (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 )
         (gg' : 'transfCoMod(0 g ~> g' )0 ) {struct gg'} : nat .
Proof.
  case : F G g g' / gg' .
  - intros ? ? ? ? g'g ? ? g''g' _ .
    exact: (2 * (S (gradeTransf _ _ _ _ g'g +
                    gradeTransf _ _ _ _ g''g')%coq_nat))%coq_nat .
  - intros ? ? ? ? g'g ? f .
    exact: (2 * (S (gradeTransf _ _ _ _ g'g + @gradeMor _ _ f)%coq_nat))%coq_nat .
  - intros ? ? h ? ? ? g'g .
    exact: (2 * (S (@gradeMor _ _ h + gradeTransf _ _ _ _ g'g)%coq_nat))%coq_nat .
  - (* UnitTransfComod corresponds to structure-arrow action for
    multifold-enriched , on the argument morphism g *)
    intros ? ? g .
    exact: (S (@gradeMor _ _ g)  )%coq_nat .
  - intros ? ? ? ? ? z1z1' .
    exact: (S (S (gradeTransf _ _ _ _ z1z1'))).
  - intros ? ? ? ? ? z2z2' .
```

```
    exact: (S (S (gradeTransf _ _ _ _ z2z2'))).
  - intros ? ? ? ? ? f1f1' ? ? f2f2' .
    refine (S (S (max (gradeTransf _ _ _ _ f1f1') (gradeTransf _ _ _ _ f2f2')))).
Defined.

Lemma gradeTransf_gt0 : forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 ),
    forall (gg' : 'transfCoMod(0 g ~> g' )0 ),
     ((S O) <= (gradeTransf gg'))%coq_nat.
Proof. intros; case : gg'; intros; apply/leP; intros; simpl; auto. Qed.

Ltac tac_gradeTransf_gt0 :=
  match goal with
  | [ gg1 : 'transfCoMod(0 _ ~> _ )0 ,
            gg2 : 'transfCoMod(0 _ ~> _ )0 ,
                  gg3 : 'transfCoMod(0 _ ~> _ )0 ,
                        gg4 : 'transfCoMod(0 _ ~> _ )0 |- _ ] =>
    move : (@gradeTransf_gt0 _ _ _ _ gg1) (@gradeTransf_gt0 _ _ _ _ gg2)
                                          (@gradeTransf_gt0 _ _ _ _ gg3)
                                          (@gradeTransf_gt0 _ _ _ _ gg4)
  | [ gg1 : 'transfCoMod(0 _ ~> _ )0 ,
            gg2 : 'transfCoMod(0 _ ~> _ )0 ,
                  gg3 : 'transfCoMod(0 _ ~> _ )0 ,
                        gg4 : 'transfCoMod(0 _ ~> _ )0 |- _ ] =>
    move : (@gradeTransf_gt0 _ _ _ _ gg1) (@gradeTransf_gt0 _ _ _ _ gg2)
                                          (@gradeTransf_gt0 _ _ _ _ gg3)
                                          (@gradeTransf_gt0 _ _ _ _ gg4)
  | [ gg1 : 'transfCoMod(0 _ ~> _ )0 ,
            gg2 : 'transfCoMod(0 _ ~> _ )0 ,
                  gg3 : 'transfCoMod(0 _ ~> _ )0 |- _ ] =>
    move : (@gradeTransf_gt0 _ _ _ _ gg1) (@gradeTransf_gt0 _ _ _ _ gg2)
                                          (@gradeTransf_gt0 _ _ _ _ gg3)
  | [ gg1 : 'transfCoMod(0 _ ~> _ )0 ,
            gg2 : 'transfCoMod(0 _ ~> _ )0  |- _ ] =>
    move : (@gradeTransf_gt0 _ _ _ _ gg1) (@gradeTransf_gt0 _ _ _ _ gg2)

  | [ gg1 : 'transfCoMod(0 _ ~> _ )0  |- _ ] =>
    move : (@gradeTransf_gt0 _ _ _ _ gg1)
  end.

Lemma degradeTransf :
  forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 %poly)
    (gg' : 'transfCoMod(0 g ~> g' )0 %poly)  (g_ g'_ : 'morCoMod(0 F ~> G )0 %poly)
    (g_g'_ : 'transfCoMod(0 g_ ~> g'_ )0 %poly),
    g_g'_ <~~2 gg' -> ( gradeTransf g_g'_ <= gradeTransf gg' )%coq_nat .
Proof.
  intros until g_g'_ . intros red_gg'.
  elim : F G g g' gg' g_ g'_ g_g'_ / red_gg' ;
    try solve [ simpl; intros;
                try match goal with
                    | [ Hred : ( _ <~~1 _ ) |- _ ] =>
                      move : (degradeMor Hred) ; clear Hred
                    end;
                intros; abstract Psatz.nia ].
(*memo: Omega.omega too weak at Pairing_Mor_morphismTransf
  Pairing_Transf_morphismMor *)
  (* erase associativities conversions then Qed. *)
Qed.

Ltac tac_degradeTransf H_gradeTransf :=
  intuition idtac;
  repeat match goal with
         | [ Hred : ( _ <~~1 _ ) |- _ ] =>
           move : (degradeMor Hred) ; clear Hred
         | [ Hred : ( _ <~~2 _ ) |- _ ] =>
           move : (degradeTransf Hred) ; clear Hred
         end;
```

```
move: H_gradeTransf; clear; simpl; intros;
try tac_gradeMor_gt0; try tac_gradeTransf_gt0; intros; Omega.omega.
```

9 Polymorphism/cut-elimination by computational/total/asymptotic/reduction/(multi-step) resolution

For 2-folded polymorph mathematics , this resolution is made of some 1-resolution-for-morphisms [solveMorCoMod] and some 2-resolution-for-transformations [solveTransfCoMod] which depends/uses of this 1-resolution-for-morphisms .

In contrast from 1-resolution-for-morphisms , the 2-resolution-for-transformations has some almost-computational data-content function [solveTransfCoMod] of the resolution and some derived logical properties [solveTransfCoModP] which are satisfied by this function [solveTransfCoMod] . In other words : the programmation of the function [solveTransfCoMod] cannot be purely-computational because of this fact : the 2-conversion-for-transformations do convert across two transformations whose domain-codomain-morphisms-computation arguments are not syntactically/grammatically-the-same , and therefore , during the recursion step which is the 2-resolution of the inner/structural ("vertical") composition [PolyTransfCoMod] cut-constructor , oneself lacks to know that the codomain-morphism of the recursively 2-resolved prefix-output transformation is grammatically-same (or propositionally-equal …) as the domain-morphism of the recursively 2-resolved postfix-output transformation . In short : the programmation of the 2-resolution function [solveTransfCoMod] shall memorize the logical property that the domain/codomain of the 2-resolved output transformation is propositionally-equal to the 1-resolution of the domain/codomain of the input transformation .

Also in contrast , regardless that oneself may extract/derive the propositional-equations [solveTransfCoMod0_Project1_Transf] [solveTransfCoMod0_PolyTransfCoMod] corresponding to the purely-data part/component of the definitional-metaconversions of the 2-resolution-for-transformations function [solveTransfCoMod] , the fact that these propositional-equations are across ([existsT]) dependent-pairs (in other words , are dependent-equalities) , will make them not very-practical/usable . This suggests that the expected presentation of the 2-resolution (and 1-resolution) is not by this ongoing computational-resolution but instead shall be by some logical-resolution (logical congruent-rewrite-style cut-elimination resolution) . Ultimately , the future confluence lemmas will require this logical-resolution .

Memo that the technical progress of this 2-resolution does NOT immediately-require the earlier lemma [convTransfCoMod_convMorCoMod_dom] [convTransfCoMod_convMorCoMod_cod] ("soundness lemma") that 2-convertible transformations do have 1-convertible domain/codomain-morphisms .

Memo that some more-lax (instead of tight/strict) alternative formulation of the inner/structural ("vertical") composition [PolyTransfCoMod] cut-constructor is possible . Such new formulation of the [PolyTransfCoMod] cut-constructor again will have some cut-adherence parameter which will relate the codomain-morphism of the prefix-input transformation in-relation-with the domain-morphism of the postfix-input transformation , by the (symmetrized , with associativity-conversion) 1-polymorphism-conversion-for-morphisms , instead of by propositional-equality . Consequently the new 2-resolution [solveTransfCoMod_requireConfluenceOf_solveMorCoMod] shall be formulated , by using 1-conversions instead of propositional-equality , from the domain/codomain of the input transformation to the domain/codomain of the 2-resolved output transformation . Now memo that the 1-confluence lemma for morphisms says that two morphisms are 1-polymorphism-convertible if and only if their 1-resolution is 1-solution-convertible . Finally such new formulations of [PolyTransfCoMod] and [solveTransfCoMod_requireConfluenceOf_solveMorCoMod] , together with the finished 1-confluence lemma for morphisms , will enable to sucessfully do the recursion step which is the 2-resolution of the [PolyTransfCoMod] cut-constructor . BUT , even in such new formulation , for simplicity (to avoid any recursively-schematic 2-conversions-for-transformations …) , the full 1-resolution of morphisms to their final solution will be required , such that the 1-solution-conversion-for-morphism is easier and coincides with tight/strict propositional-equality (only congruences …) . Elsewhere , memo that , even in this alternative formulation , the "soundness lemma" [convTransfCoMod_convMorCoMod_dom] will not be immediately-used during this alternative 2-resolution , but will only confirm after-the-fact the properties of this alternative 2-resolution .

As always , this COQ program and deduction is mostly-automated !

```
Module Resolve.
Import TWOFOLD.Resolve.
```

```
Export Sol.Ex_Notations.

Fixpoint solveTransfCoMod_requireConfluenceOf_solveMorCoMod len {struct len} :
  forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 )
    (gg' : 'transfCoMod(0 g ~> g' )0 ),
  forall gradeTransf_gg' : (gradeTransf gg' <= len)%coq_nat,
    { g_ : 'morCoMod(0 F ~> G )0 %sol & { g'_ : 'morCoMod(0 F ~> G )0 %sol &
       { gg'Sol : 'transfCoMod(0 g_ ~> g'_ )0 %sol |
       ( ( Sol.toPolyMor g_ <~~1 g ) * ( Sol.toPolyMor g'_ <~~1 g' ) )%type } } } .
Abort.

Fixpoint solveTransfCoMod_PolyTransfCoMod len {struct len} :
  forall (F G : obCoMod) (gSol g'Sol : 'morCoMod(0 F ~> G )0%sol)
    (g'Sol_gSol : 'transfCoMod(0 g'Sol ~> gSol )0%sol)
    (g'Sol0  g''Sol : 'morCoMod(0 F ~> G )0%sol)
    (g''Sol_g'Sol : 'transfCoMod(0 g''Sol ~> g'Sol0 )0%sol)
    (eqMor : Sol.toPolyMor g'Sol0 <~>1 Sol.toPolyMor g'Sol)
    (gradeTransf_gg' : (gradeTransf ((Sol.toPolyTransf g''Sol_g'Sol)
                 o^CoMod (Sol.toPolyTransf g'Sol_gSol) # eqMor) <= len)%coq_nat),
    {g_ : 'morCoMod(0 F ~> G )0%sol & {g'_ : 'morCoMod(0 F ~> G )0%sol &
                                       'transfCoMod(0 g_ ~> g'_ )0%sol } } .
Proof.
  case : len => [ | len ].

  (* len is O *)
  - ( move => F G gSol g'Sol g'Sol_gSol g'Sol0 g''Sol g''Sol_g'Sol
              eqMor gradeTransf_gg' );
      exfalso; abstract tac_degradeTransf gradeTransf_gg'.

  (* len is (S len) *)
  (* gg' is g''g' o^CoMod g'g , to  (g''Sol_g'Sol o^CoMod g'Sol_gSol) *)
  - move =>  F G gSol g'Sol g'Sol_gSol g'Sol0 g''Sol g''Sol_g'Sol
             eqMor gradeTransf_gg' .
    { destruct g''Sol_g'Sol as
          [ F G g'Sol0 (* @'UnitTransfCoMod g'Sol *)
          | F1 F2 Z1 z1Sol z1'Sol z1Solz1'Sol (* ~_1 @ F2 _o>CoMod^ z1Solz1'Sol *)
          | F1 F2 Z2 z2Sol z2'Sol z2Solz2'Sol (* ~_2 @ F1 _o>CoMod^ z2Solz2'Sol *)
          | L F1 F2 f1Sol f1'Sol f1Solf1'Sol f2Sol f2'Sol f2Solf2'Sol
(* << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> *) ] .

      (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
      (@'UnitTransfCoMod g'Sol o^CoMod g'Sol_gSol) *)
      * eexists. eexists. refine (g'Sol_gSol)%sol .

      (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is ( (
      ~_1 @ F2 _o>CoMod^ z1Solz1'Sol ) o^CoMod g'Sol_gSol) *)
      * move: (Sol.Destruct_domPair.transfCoMod_domPairP g'Sol_gSol)
        => g'Sol_gSol_domPairP.
        { destruct g'Sol_gSol_domPairP as
              [ F1 F2 G g  (*  (  @'UnitTransfCoMod g %sol  *)
              | F1 F2 Z1 _z1Sol _z1'Sol _z1Sol_z1'Sol
              (* ~_1 @ F2 _o>CoMod^ _z1Sol_z1'Sol *)
              | F1 F2 Z2 z2Sol z2'Sol z2Solz2'Sol
              (* ~_2 @ F1 _o>CoMod^ z2Solz2'Sol *)
              | L L' F1 F2 f1Sol f1'Sol f1Solf1'Sol f2Sol f2'Sol f2Solf2'Sol
(* << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> *) ] .

          (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
          ( ( ~_1 @ F2 _o>CoMod^ z1Solz1'Sol ) o^CoMod ( @'UnitTransfCoMod ( g ) )
          ) *)
          - eexists. eexists. refine ( ~_1 @ F2 _o>CoMod^ z1Solz1'Sol )%sol .

          (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
          ( ( ~_1 @ F2 _o>CoMod^ z1Solz1'Sol ) o^CoMod ( ~_1 @ F2 _o>CoMod^
          _z1Sol_z1'Sol ) ) *)
          - have [:blurb] z1Sol_z1' :=
              (projT2 (projT2 (solveTransfCoMod_PolyTransfCoMod len _ _ _ _
```

```
                _z1Sol_z1'Sol _ _ z1Solz1'Sol
          (EqMorCoMod.Inversion_Project1.convMorCoMod_Project1P' eqMor) blurb )));
               first by clear -gradeTransf_gg';
               abstract tac_degradeTransf gradeTransf_gg' .
           eexists. eexists. refine ( ~_1 @ F2 _o>CoMod^ z1Sol_z1' )%sol .

         (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
         ( ( ~_1 @ F2 _o>CoMod^ z1Solz1'Sol ) o^CoMod ( ~_2 @ F1 _o>CoMod^
         z2Solz2'Sol ) ) *)
         - exfalso. clear -eqMor. apply:
     ( EqMorCoMod.Inversion_Exfalso.convMorCoMod_ExfalsoP_Project1_Project2 eqMor ).

         (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
         ( ( ~_1 @ F2 _o>CoMod^ z1Solz1'Sol ) o^CoMod ( << f1Solf1'Sol ,^CoMod
         f2Solf2'Sol >> ) ) *)
         - exfalso. clear -eqMor. apply:
      ( EqMorCoMod.Inversion_Exfalso.convMorCoMod_ExfalsoP_Project1_Pairing eqMor ).
        }

       (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is ( (
       ~_2 @ F1 _o>CoMod^ z2Solz2'Sol ) o^CoMod g'Sol_gSol) *)
       * move: (Sol.Destruct_domPair.transfCoMod_domPairP g'Sol_gSol)
         => g'Sol_gSol_domPairP.
         { destruct g'Sol_gSol_domPairP as
              [ F1 F2 G g  (*  (  @'UnitTransfCoMod g %sol  *)
              | F1 F2 Z1 z1Sol z1'Sol z1Solz1'Sol
              (* ~_1 @ F2 _o>CoMod^ z1Solz1'Sol *)
              | F1 F2 Z2 _z2Sol _z2'Sol _z2Sol_z2'Sol
              (* ~_2 @ F1 _o>CoMod^ _z2Sol_z2'Sol *)
              | L L' F1 F2 f1Sol f1'Sol f1Solf1'Sol f2Sol f2'Sol f2Solf2'Sol
(* << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> *) ] .

         (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
         ( ( ~_2 @ F1 _o>CoMod^ z2Solz2'Sol ) o^CoMod ( @'UnitTransfCoMod ( g ) )
         ) *)
         - eexists. eexists. refine ( ~_2 @ F1 _o>CoMod^ z2Solz2'Sol )%sol .

         (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
         ( ( ~_2 @ F1 _o>CoMod^ z2Solz2'Sol ) o^CoMod ( ~_1 @ F2 _o>CoMod^
         z1Solz1'Sol ) ) *)
         - exfalso. clear -eqMor. apply:
           ( EqMorCoMod.Inversion_Exfalso.convMorCoMod_ExfalsoP_Project1_Project2
               (  EqMorCoMod.convMorCoMod_sym eqMor)  ).

         (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
         ( ( ~_2 @ F1 _o>CoMod^ z2Solz2'Sol ) o^CoMod ( ~_2 @ F1 _o>CoMod^
         _z2Sol_z2'Sol ) ) *)
         - have [:blurb] z2Sol_z2' :=
             (projT2 (projT2 (solveTransfCoMod_PolyTransfCoMod len _ _ _ _
               _z2Sol_z2'Sol _ _ z2Solz2'Sol
          (EqMorCoMod.Inversion_Project2.convMorCoMod_Project2P' eqMor) blurb )));
               first by clear -gradeTransf_gg';
               abstract tac_degradeTransf gradeTransf_gg' .
           eexists. eexists. refine ( ~_2 @ F1 _o>CoMod^ z2Sol_z2' )%sol .

         (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
         ( ( ~_2 @ F1 _o>CoMod^ z2Solz2'Sol ) o^CoMod ( << f1Solf1'Sol ,^CoMod
         f2Solf2'Sol >> ) ) *)
         - exfalso. clear -eqMor. apply:
      ( EqMorCoMod.Inversion_Exfalso.convMorCoMod_ExfalsoP_Project2_Pairing eqMor ).
        }

       (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is ( (
       << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ) o^CoMod g'Sol_gSol) *)
       * move: (Sol.Destruct_codPair.transfCoMod_codPairP g'Sol_gSol)
         => g'Sol_gSol_codPairP.
         { destruct g'Sol_gSol_codPairP as
```

```
            [ F G1 G2 g  (*  (  @'UnitTransfCoMod (g) %sol  *)
            | F1 F2 Z1 Z1' z1Sol z1'Sol z1Solz1'Sol
            (* ~_1 @ F2 _o>CoMod^ z1Solz1'Sol *)
            | F1 F2 Z2 Z2' z2Sol z2'Sol z2Solz2'Sol
            (* ~_2 @ F1 _o>CoMod^ z2Solz2'Sol *)
            | L F1 F2 _f1Sol _f1'Sol _f1Sol_f1'Sol _f2Sol _f2'Sol _f2Sol_f2'Sol
(* << _f1Sol_f1'Sol ,^CoMod _f2Sol_f2'Sol >> *) ] .

          (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
          ( ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ) o^CoMod ( @'UnitTransfCoMod
          (g) ) ) *)
          - eexists. eexists. refine ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> )%sol .

          (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
          ( ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ) o^CoMod ( ~_1 @ F2 _o>CoMod^
          z1Solz1'Sol ) ) *)
          - exfalso. clear -eqMor. apply:
            ( EqMorCoMod.Inversion_Exfalso.convMorCoMod_ExfalsoP_Project1_Pairing
                (EqMorCoMod.convMorCoMod_sym eqMor) ).

          (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
          ( ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ) o^CoMod ( _2 @ F1 _o>CoMod^
          z2Solz2'Sol ) ) *)
          - exfalso. clear -eqMor. apply:
             ( EqMorCoMod.Inversion_Exfalso.convMorCoMod_ExfalsoP_Project2_Pairing
                 (EqMorCoMod.convMorCoMod_sym eqMor)  ).

          (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
          ( ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ) o^CoMod ( << _f1Sol_f1'Sol
          ,^CoMod _f2Sol_f2'Sol >> ) ) *)
          - simpl in eqMor , gradeTransf_gg' .
            have [:blurb] f1Sol_f1' :=
              (projT2 (projT2 (solveTransfCoMod_PolyTransfCoMod len _ _ _ _
                       _f1Sol_f1'Sol _ _ f1Solf1'Sol
      (proj1 (EqMorCoMod.Inversion_Pairing.convMorCoMod_PairingP' eqMor)) blurb )));
                first by abstract tac_degradeTransf gradeTransf_gg' .
            have [:blurb] f2Sol_f2' :=
              (projT2 (projT2 (solveTransfCoMod_PolyTransfCoMod len _ _ _ _
                               _f2Sol_f2'Sol _ _ f2Solf2'Sol
      (proj2 (EqMorCoMod.Inversion_Pairing.convMorCoMod_PairingP' eqMor)) blurb )));
                first by abstract tac_degradeTransf gradeTransf_gg' .
            eexists. eexists. refine ( <<  f1Sol_f1' ,^CoMod f2Sol_f2' >>  )%sol .
        }
    }
Defined.

Arguments solveTransfCoMod_PolyTransfCoMod !len _ _ _ _ _ _ _ _ !g''Sol_g'Sol _ _
  : simpl nomatch .

Notation "g''g' o^CoMod g'g # eqMor @ gradeTransf_gg'" :=
  (@solveTransfCoMod_PolyTransfCoMod _ _ _ _ _ g'g _ _ g''g' eqMor gradeTransf_gg')
    (at level 40 , g'g at next level, eqMor at next level) : sol_scope.

Lemma solveTransfCoMod_PolyTransfCoMod_len :
  forall len (F G : obCoMod) (gSol g'Sol : 'morCoMod(0 F ~> G )0%sol)
    (g'Sol_gSol : 'transfCoMod(0 g'Sol ~> gSol )0%sol)
    (g'Sol0  g''Sol : 'morCoMod(0 F ~> G )0%sol)
    (g''Sol_g'Sol : 'transfCoMod(0 g''Sol ~> g'Sol0 )0%sol)
    (eqMor: Sol.toPolyMor g'Sol0 <~>1 Sol.toPolyMor g'Sol)
    (gradeTransf_gg'_len : (gradeTransf ((Sol.toPolyTransf g''Sol_g'Sol)
                    o^CoMod (Sol.toPolyTransf g'Sol_gSol) # eqMor) <= len)%coq_nat),
  forall len' eqMor' (gradeTransf_gg'_len' :
                  (gradeTransf ((Sol.toPolyTransf g''Sol_g'Sol)
               o^CoMod (Sol.toPolyTransf g'Sol_gSol) # eqMor') <= len')%coq_nat),
    ( g''Sol_g'Sol o^CoMod g'Sol_gSol # eqMor @ gradeTransf_gg'_len
      = g''Sol_g'Sol o^CoMod g'Sol_gSol # eqMor' @ gradeTransf_gg'_len' )%sol .
Proof.
```

```
  induction len as [ | len ].
  - ( move => ? ? ? ? ? ? ? ? ?  gradeTransf_gg'_len ); exfalso;
      clear -gradeTransf_gg'_len;
        by abstract tac_degradeTransf gradeTransf_gg'_len.
  - intros. destruct len'.
    + exfalso; clear -gradeTransf_gg'_len';
        by abstract tac_degradeTransf gradeTransf_gg'_len'.
    + destruct g''Sol_g'Sol .
      * reflexivity.
      * { destruct (Sol.Destruct_domPair.transfCoMod_domPairP g'Sol_gSol); simpl.
          - reflexivity.
          - erewrite IHlen. reflexivity.
          - exfalso. apply
      (EqMorCoMod.Inversion_Exfalso.convMorCoMod_ExfalsoP_Project1_Project2 eqMor).
          - exfalso. apply
      (EqMorCoMod.Inversion_Exfalso.convMorCoMod_ExfalsoP_Project1_Pairing eqMor).
        }
      * { destruct (Sol.Destruct_domPair.transfCoMod_domPairP g'Sol_gSol); simpl.
          - reflexivity.
          - exfalso. apply
        (EqMorCoMod.Inversion_Exfalso.convMorCoMod_ExfalsoP_Project1_Project2
                              (EqMorCoMod.convMorCoMod_sym eqMor)).
          - erewrite IHlen. reflexivity.
          - exfalso. apply
       (EqMorCoMod.Inversion_Exfalso.convMorCoMod_ExfalsoP_Project2_Pairing eqMor).
        }
      * { destruct (Sol.Destruct_codPair.transfCoMod_codPairP g'Sol_gSol); simpl.
          - reflexivity.
          - exfalso. apply
              (EqMorCoMod.Inversion_Exfalso.convMorCoMod_ExfalsoP_Project1_Pairing
                 (EqMorCoMod.convMorCoMod_sym eqMor)).
          - exfalso. apply
              (EqMorCoMod.Inversion_Exfalso.convMorCoMod_ExfalsoP_Project2_Pairing
                 (EqMorCoMod.convMorCoMod_sym eqMor)).
          - unfold ssr_have. simpl.
            erewrite (IHlen _ _ _ _ f1f1' _ _ g''Sol_g'Sol1).
            erewrite (IHlen _ _ _ _ f2f2' _ _ g''Sol_g'Sol2). reflexivity.
        }
Qed.

Definition solveTransfCoMod_PolyTransfCoMod0 :
  forall (F G : obCoMod) (gSol g'Sol : 'morCoMod(0 F ~> G )0%sol)
    (g'Sol_gSol : 'transfCoMod(0 g'Sol ~> gSol )0%sol)
    (g'Sol0  g''Sol : 'morCoMod(0 F ~> G )0%sol)
    (g''Sol_g'Sol : 'transfCoMod(0 g''Sol ~> g'Sol0 )0%sol)
    (eqMor: Sol.toPolyMor g'Sol0 <~>1 Sol.toPolyMor g'Sol),
    {g_ : 'morCoMod(0 F ~> G )0%sol & {g'_ : 'morCoMod(0 F ~> G )0%sol &
                                            'transfCoMod(0 g_ ~> g'_ )0%sol } } .
Proof.
  intros; apply: (@solveTransfCoMod_PolyTransfCoMod
                  (gradeTransf ((Sol.toPolyTransf g''Sol_g'Sol)
                         o^CoMod (Sol.toPolyTransf g'Sol_gSol) # eqMor))%coq_nat
         F G gSol g'Sol g'Sol_gSol g'Sol0 g''Sol g''Sol_g'Sol eqMor); constructor.
Defined.

Notation "g''g' o^CoMod g'g # eqMor" :=
  (@solveTransfCoMod_PolyTransfCoMod0 _ _ _ _ g'g _ _ g''g' eqMor)
    (at level 40 , g'g at next level) : sol_scope.

Lemma solveTransfCoMod_PolyTransfCoMod0_len :
  forall len (F G : obCoMod) (gSol g'Sol : 'morCoMod(0 F ~> G )0%sol)
    (g'Sol_gSol : 'transfCoMod(0 g'Sol ~> gSol )0%sol)
    (g'Sol0  g''Sol : 'morCoMod(0 F ~> G )0%sol)
    (g''Sol_g'Sol : 'transfCoMod(0 g''Sol ~> g'Sol0 )0%sol)
    (eqMor: Sol.toPolyMor g'Sol0 <~>1 Sol.toPolyMor g'Sol)
    (gradeTransf_gg'_len : (gradeTransf ((Sol.toPolyTransf g''Sol_g'Sol)
                  o^CoMod (Sol.toPolyTransf g'Sol_gSol) # eqMor) <= len)%coq_nat),
```

```
  forall (eqMor': Sol.toPolyMor g'Sol0 <~>1 Sol.toPolyMor g'Sol),
    ( g''Sol_g'Sol o^CoMod g'Sol_gSol # eqMor'
      = g''Sol_g'Sol o^CoMod g'Sol_gSol # eqMor @ gradeTransf_gg'_len )%sol .
Proof. intros. erewrite solveTransfCoMod_PolyTransfCoMod_len. reflexivity. Qed.

Lemma solveTransfCoMod_PolyTransfCoMod0___UnitTransfCoMod :
  forall (F G : obCoMod) (gSol g'Sol : 'morCoMod(0 F ~> G )0%sol)
    (g'Sol_gSol : 'transfCoMod(0 g'Sol ~> gSol )0%sol)
    (g'Sol0 : 'morCoMod(0 F ~> G )0%sol)
    (eqMor : Sol.toPolyMor g'Sol0 <~>1 Sol.toPolyMor g'Sol),
    projT2 (projT2 ('UnitTransfCoMod o^CoMod g'Sol_gSol # eqMor)%sol)
    = (g'Sol_gSol)%sol.
Proof. reflexivity. Qed.

Lemma solveTransfCoMod_PolyTransfCoMod0_UnitTransfCoMod_Project1_Transf :
  forall (F1 G : obCoMod) (z1Sol z1'Sol : 'morCoMod(0 F1 ~> G )0%sol)
    (z1Solz1'Sol : 'transfCoMod(0 z1Sol ~> z1'Sol )0%sol)
    (F2 : obCoMod) (g : 'morCoMod(0 Pair F1 F2 ~> G )0%sol)
    (eqMor : ~_1 o>CoMod (Sol.toPolyMor z1'Sol) <~>1 Sol.toPolyMor g),
   projT2 (projT2 (~_1 _o>CoMod^ z1Solz1'Sol o^CoMod 'UnitTransfCoMod # eqMor)%sol)
   = ( ~_1 @ F2 _o>CoMod^ z1Solz1'Sol )%sol.
Proof. reflexivity. Qed.

Lemma solveTransfCoMod_PolyTransfCoMod0_Project1_Transf_Project1_Transf_dom :
  forall (F1 Z1 : obCoMod) (z1Sol z1'Sol : 'morCoMod(0 F1 ~> Z1 )0%sol)
    (z1Solz1'Sol : 'transfCoMod(0 z1Sol ~> z1'Sol )0%sol)
    (F2 : obCoMod) (_z1Sol : 'morCoMod(0 F1 ~> Z1 )0%sol)
    (eqMor : ~_1 o>CoMod (Sol.toPolyMor z1'Sol)
                         <~>1 Sol.toPolyMor (~_1 o>CoMod _z1Sol)%sol)
    (_z1'Sol : 'morCoMod(0 F1 ~> Z1 )0%sol)
    (_z1Sol_z1'Sol : 'transfCoMod(0 _z1Sol ~> _z1'Sol )0%sol),
 projT1 (~_1 _o>CoMod^ z1Solz1'Sol o^CoMod ~_1 _o>CoMod^ _z1Sol_z1'Sol # eqMor)%sol
    = (~_1 @ F2 o>CoMod (projT1 (z1Solz1'Sol o^CoMod _z1Sol_z1'Sol
             # EqMorCoMod.Inversion_Project1.convMorCoMod_Project1P' eqMor)))%sol.
Proof.
  intros. simpl. erewrite solveTransfCoMod_PolyTransfCoMod0_len. reflexivity.
Qed. (*TIME: 44 sec *)

Lemma solveTransfCoMod_PolyTransfCoMod0_Project1_Transf_Project1_Transf :
  forall (F1 Z1 : obCoMod) (z1Sol z1'Sol : 'morCoMod(0 F1 ~> Z1 )0%sol)
    (z1Solz1'Sol : 'transfCoMod(0 z1Sol ~> z1'Sol )0%sol) (F2 : obCoMod)
    (_z1Sol : 'morCoMod(0 F1 ~> Z1 )0%sol)
    (eqMor : ~_1 o>CoMod (Sol.toPolyMor z1'Sol)
                         <~>1 Sol.toPolyMor (~_1 o>CoMod _z1Sol)%sol)
    (_z1'Sol : 'morCoMod(0 F1 ~> Z1 )0%sol)
    (_z1Sol_z1'Sol : 'transfCoMod(0 _z1Sol ~> _z1'Sol )0%sol)
    (z1Sol_z1' := (z1Solz1'Sol o^CoMod _z1Sol_z1'Sol
             # (EqMorCoMod.Inversion_Project1.convMorCoMod_Project1P' eqMor))%sol),
    ( ( ( ~_1 _o>CoMod^ z1Solz1'Sol )
          o^CoMod ( ~_1 _o>CoMod^ _z1Sol_z1'Sol ) # eqMor )%sol)
    = existT _ (~_1  o>CoMod (projT1  z1Sol_z1'))%sol
             (existT _ (~_1  o>CoMod (projT1 (projT2 z1Sol_z1' )))%sol
                     ( ~_1 @ F2 _o>CoMod^ (projT2 (projT2 z1Sol_z1')) )%sol).
Proof.
  intros. subst z1Sol_z1'. rewrite [solveTransfCoMod_PolyTransfCoMod0 in LHS]lock.
  erewrite solveTransfCoMod_PolyTransfCoMod0_len. unlock. reflexivity.
Qed. (*TIME: LONG 412 sec /!\ *)

(**ETC : ... *)

Fixpoint solveTransfCoMod_PolyTransfCoModP len {struct len} :
  forall (F G : obCoMod) (gSol g'Sol : 'morCoMod(0 F ~> G )0%sol)
    (g'Sol_gSol : 'transfCoMod(0 g'Sol ~> gSol )0%sol)
    (g'Sol0  g''Sol : 'morCoMod(0 F ~> G )0%sol)
    (g''Sol_g'Sol : 'transfCoMod(0 g''Sol ~> g'Sol0 )0%sol)
    (eqMor : Sol.toPolyMor g'Sol0 <~>1 Sol.toPolyMor g'Sol)
    (gradeTransf_gg' : (gradeTransf ((Sol.toPolyTransf g''Sol_g'Sol)
```

```
                    o^CoMod (Sol.toPolyTransf g'Sol_gSol) # eqMor) <= len)%coq_nat)
    (eqMor_param : Sol.toPolyMor g'Sol0 <~>1 Sol.toPolyMor g'Sol),

    ( ( (g''Sol = (projT1 (g''Sol_g'Sol o^CoMod g'Sol_gSol
                                        # eqMor @ gradeTransf_gg' )%sol)) *
        (gSol = (projT1 (projT2 (g''Sol_g'Sol o^CoMod g'Sol_gSol
                                        # eqMor @ gradeTransf_gg' )%sol))) ) *
      ( Sol.toPolyTransf (projT2 (projT2 (g''Sol_g'Sol o^CoMod g'Sol_gSol
                                        # eqMor @ gradeTransf_gg' )%sol))
       <~~2 (Sol.toPolyTransf g''Sol_g'Sol o^CoMod Sol.toPolyTransf g'Sol_gSol
                                                # eqMor_param )%poly ) )%type .
Proof.
  case : len => [ | len ].

  (* len is O *)
  - ( move => F G gSol g'Sol g'Sol_gSol g'Sol0 g''Sol g''Sol_g'Sol
             eqMor gradeTransf_gg' eqMor_param );
      exfalso; abstract tac_degradeTransf gradeTransf_gg'.

  (* len is (S len) *)
  (* gg' is g''g' o^CoMod g'g , to  (g''Sol_g'Sol o^CoMod g'Sol_gSol) *)
  - move => F G gSol g'Sol g'Sol_gSol g'Sol0 g''Sol g''Sol_g'Sol
           eqMor gradeTransf_gg' eqMor_param.
    { destruct g''Sol_g'Sol as
          [ F G g'Sol0 (* @'UnitTransfCoMod g'Sol0 *)
          | F1 F2 Z1 z1Sol z1'Sol z1Solz1'Sol (* ~_1 @ F2 _o>CoMod^ z1Solz1'Sol *)
          | F1 F2 Z2 z2Sol z2'Sol z2Solz2'Sol (* ~_2 @ F1 _o>CoMod^ z2Solz2'Sol *)
          | L F1 F2 f1Sol f1'Sol f1Solf1'Sol f2Sol f2'Sol f2Solf2'Sol
(* << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> *) ] .

      (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
      (@'UnitTransfCoMod g'Sol0 o^CoMod g'Sol_gSol) *)
      * clear; move:
              (EqMorCoMod.Inversion_toPolyMor.convMorCoMod_toPolyMorP' eqMor);
        abstract tac_reduce_solveMorCoMod0.

      (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is ( (
      ~_1 @ F2 _o>CoMod^ z1Solz1'Sol ) o^CoMod g'Sol_gSol) *)
      * move: (Sol.Destruct_domPair.transfCoMod_domPairP g'Sol_gSol)
        => g'Sol_gSol_domPairP.
        { destruct g'Sol_gSol_domPairP as
              [ F1 F2 G g  (*  (  @'UnitTransfCoMod g %sol  *)
              | F1 F2 Z1 _z1Sol _z1'Sol _z1Sol_z1'Sol
              (* ~_1 @ F2 _o>CoMod^ _z1Sol_z1'Sol *)
              | F1 F2 Z2 z2Sol z2'Sol z2Solz2'Sol
              (* ~_2 @ F1 _o>CoMod^ z2Solz2'Sol *)
              | L L' F1 F2 f1Sol f1'Sol f1Solf1'Sol f2Sol f2'Sol f2Solf2'Sol
(* << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> *) ] .

          (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
      ( ( ~_1 @ F2 _o>CoMod^ z1Solz1'Sol ) o^CoMod ( @'UnitTransfCoMod ( g ) ) ) *)
          - clear; move:
                  (EqMorCoMod.Inversion_toPolyMor.convMorCoMod_toPolyMorP' eqMor);
            abstract tac_reduce_solveMorCoMod0.

          (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
          ( ( ~_1 @ F2 _o>CoMod^ z1Solz1'Sol ) o^CoMod ( ~_1 @ F2 _o>CoMod^
          _z1Sol_z1'Sol ) ) *)
          - simpl; set same_blurb := (_ gradeTransf_gg' : ( _ <= len )%coq_nat) .
            move: (solveTransfCoMod_PolyTransfCoModP len _ _ _ _
                         _z1Sol_z1'Sol _ _ z1Solz1'Sol _ same_blurb
              (EqMorCoMod.Inversion_Project1.convMorCoMod_Project1P' eqMor_param)).
            clear; abstract tac_reduce_solveMorCoMod0.

          (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
          ( ( ~_1 @ F2 _o>CoMod^ z1Solz1'Sol ) o^CoMod ( ~_2 @ F1 _o>CoMod^
          z2Solz2'Sol ) ) *)
```

```
          - exfalso. clear -eqMor. apply:
    ( EqMorCoMod.Inversion_Exfalso.convMorCoMod_ExfalsoP_Project1_Project2 eqMor ).

          (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
          ( ( ~_1 @ F2 _o>CoMod^ z1Solz1'Sol ) o^CoMod ( << f1Solf1'Sol ,^CoMod
          f2Solf2'Sol >> ) ) *)
          - exfalso. clear -eqMor. apply:
     ( EqMorCoMod.Inversion_Exfalso.convMorCoMod_ExfalsoP_Project1_Pairing eqMor ).
        }

      (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is ( (
      ~_2 @ F1 _o>CoMod^ z2Solz2'Sol ) o^CoMod g'Sol_gSol) *)
      * move: (Sol.Destruct_domPair.transfCoMod_domPairP g'Sol_gSol)
        => g'Sol_gSol_domPairP.
        { destruct g'Sol_gSol_domPairP as
              [ F1 F2 G g  (*  (  @'UnitTransfCoMod g %sol  *)
              | F1 F2 Z1 z1Sol z1'Sol z1Solz1'Sol
              (* ~_1 @ F2 _o>CoMod^ z1Solz1'Sol *)
              | F1 F2 Z2 _z2Sol _z2'Sol _z2Sol_z2'Sol
              (* ~_2 @ F1 _o>CoMod^ _z2Sol_z2'Sol *)
              | L L' F1 F2 f1Sol f1'Sol f1Solf1'Sol f2Sol f2'Sol f2Solf2'Sol
(* << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> *) ] .

          (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
      ( ( ~_2 @ F1 _o>CoMod^ z2Solz2'Sol ) o^CoMod ( @'UnitTransfCoMod ( g ) ) ) *)
          - clear;
            move: (EqMorCoMod.Inversion_toPolyMor.convMorCoMod_toPolyMorP' eqMor);
            abstract tac_reduce_solveMorCoMod0.

          (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
          ( ( ~_2 @ F1 _o>CoMod^ z2Solz2'Sol ) o^CoMod ( ~_1 @ F2 _o>CoMod^
          z1Solz1'Sol ) ) *)
          - exfalso. clear -eqMor. apply:
            ( EqMorCoMod.Inversion_Exfalso.convMorCoMod_ExfalsoP_Project1_Project2
                                         (EqMorCoMod.convMorCoMod_sym eqMor) ).

          (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
          ( ( ~_2 @ F1 _o>CoMod^ z2Solz2'Sol ) o^CoMod ( ~_2 @ F1 _o>CoMod^
          _z2Sol_z2'Sol ) ) *)
          - simpl; set same_blurb := (_ gradeTransf_gg' : ( _ <= len )%coq_nat) .
            move: (solveTransfCoMod_PolyTransfCoModP len _ _ _ _
                                 _z2Sol_z2'Sol _ _ z2Solz2'Sol _ same_blurb
             (EqMorCoMod.Inversion_Project2.convMorCoMod_Project2P' eqMor_param) );
              clear; abstract tac_reduce_solveMorCoMod0.

          (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
          ( ( ~_2 @ F1 _o>CoMod^ z2Solz2'Sol ) o^CoMod ( << f1Solf1'Sol ,^CoMod
          f2Solf2'Sol >> ) ) *)
          - exfalso. clear -eqMor. apply:
     ( EqMorCoMod.Inversion_Exfalso.convMorCoMod_ExfalsoP_Project2_Pairing eqMor ).
        }

      (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is ( (
      << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ) o^CoMod g'Sol_gSol) *)
      * move: (Sol.Destruct_codPair.transfCoMod_codPairP g'Sol_gSol)
        => g'Sol_gSol_codPairP.
        { destruct g'Sol_gSol_codPairP as
              [ F G1 G2 g  (*  (  @'UnitTransfCoMod (g) %sol  *)
              | F1 F2 Z1 Z1' z1Sol z1'Sol z1Solz1'Sol
              (* ~_1 @ F2 _o>CoMod^ z1Solz1'Sol *)
              | F1 F2 Z2 Z2' z2Sol z2'Sol z2Solz2'Sol
              (* ~_2 @ F1 _o>CoMod^ z2Solz2'Sol *)
              | L F1 F2 _f1Sol _f1'Sol _f1Sol_f1'Sol _f2Sol _f2'Sol _f2Sol_f2'Sol
(* << _f1Sol_f1'Sol ,^CoMod _f2Sol_f2'Sol >> *) ] .

          (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
          ( ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ) o^CoMod ( @'UnitTransfCoMod
```

```
        (g) ) ) *)
        - clear;
          move: (EqMorCoMod.Inversion_toPolyMor.convMorCoMod_toPolyMorP' eqMor);
          abstract tac_reduce_solveMorCoMod0.

        (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
        ( ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ) o^CoMod ( ~_1 @ F2 _o>CoMod^
        z1Solz1'Sol ) ) *)
        - exfalso. clear -eqMor. apply:
          (EqMorCoMod.Inversion_Exfalso.convMorCoMod_ExfalsoP_Project1_Pairing
             (EqMorCoMod.convMorCoMod_sym eqMor)).

        (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
        ( ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ) o^CoMod ( _2 @ F1 _o>CoMod^
        z2Solz2'Sol ) ) *)
        - exfalso. clear -eqMor. apply:
          (EqMorCoMod.Inversion_Exfalso.convMorCoMod_ExfalsoP_Project2_Pairing
             (EqMorCoMod.convMorCoMod_sym eqMor)).

        (* gg' is g''g' o^CoMod g'g , to (g''Sol_g'Sol o^CoMod g'Sol_gSol) , is
        ( ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ) o^CoMod ( << _f1Sol_f1'Sol
        ,^CoMod _f2Sol_f2'Sol >> ) ) *)
        - simpl; set same_blurb1 := (_ gradeTransf_gg' : ( _ <= len )%coq_nat) .
          move: (solveTransfCoMod_PolyTransfCoModP len _ _ _ _
                                 _f1Sol_f1'Sol _ _ f1Solf1'Sol _ same_blurb1
       (proj1 (EqMorCoMod.Inversion_Pairing.convMorCoMod_PairingP' eqMor_param))).
          set same_blurb2 := (_ gradeTransf_gg' : ( _ <= len )%coq_nat) .
          move: (solveTransfCoMod_PolyTransfCoModP len _ _ _ _
                                 _f2Sol_f2'Sol _ _ f2Solf2'Sol _ same_blurb2
       (proj2 (EqMorCoMod.Inversion_Pairing.convMorCoMod_PairingP' eqMor_param))).
          clear; abstract tac_reduce_solveMorCoMod0.
       }
    }
Qed.

Fixpoint solveTransfCoMod_TransfCoMod_PolyMorCoMod_Pre len {struct len} :
  forall (F G : obCoMod) (g'Sol gSol : 'morCoMod(0 F ~> G )0%sol)
    (g'Sol_gSol : 'transfCoMod(0 g'Sol ~> gSol )0%sol)
    (E : obCoMod) (fSol : 'morCoMod(0 E ~> F )0%sol)
    (gradeTransf_gg' : (gradeTransf (Sol.toPolyMor fSol
                       _o>CoMod^ Sol.toPolyTransf g'Sol_gSol) <= len)%coq_nat),
    {g_ : 'morCoMod(0 E ~> G )0%sol &
        {g'_ : 'morCoMod(0 E ~> G )0%sol &
              'transfCoMod(0 g_ ~> g'_ )0%sol } }.
Proof.
  case : len => [ | len ].

  (* len is O *)
  - ( move => F G g'Sol gSol g'Sol_gSol E fSol gradeTransf_gg' );
      exfalso; abstract tac_degradeTransf gradeTransf_gg'.

  (* len is (S len) *)
  (* gg' is f _o>CoMod^ g'g , to  (fSol _o>CoMod^ g'Sol_gSol) *)
  - move => F G g'Sol gSol g'Sol_gSol E fSol gradeTransf_gg'.
    { destruct g'Sol_gSol as
        [ F G gSol (* @'UnitTransfCoMod gSol *)
        | F1 F2 Z1 z1Sol z1'Sol z1Solz1'Sol (* ~_1 @ F2 _o>CoMod^ z1Solz1'Sol *)
        | F1 F2 Z2 z2Sol z2'Sol z2Solz2'Sol (* ~_2 @ F1 _o>CoMod^ z2Solz2'Sol *)
        | L F1 F2 f1Sol f1'Sol f1Solf1'Sol f2Sol f2'Sol f2Solf2'Sol
(* << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> *) ] .

      (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is (fSol
      _o>CoMod^ @'UnitTransfCoMod gSol) *)
      * have fSol_o_gSol_prop :=
      (@solveMorCoMod0P _ _ ( (Sol.toPolyMor fSol) o>CoMod (Sol.toPolyMor gSol) )).
        set fSol_o_gSol := (@solveMorCoMod0 _ _ ( (Sol.toPolyMor fSol)
                               o>CoMod (Sol.toPolyMor gSol) )) in fSol_o_gSol_prop.
```

```
    eexists. eexists. refine ( @'UnitTransfCoMod fSol_o_gSol )%sol .

  (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is (fSol
  _o>CoMod^ ~_1 @ F2 _o>CoMod^ z1Solz1'Sol) *)
  * move: (Sol.Destruct_codPair.morCoMod_codPairP fSol) => fSol_codPairP.
    { destruct fSol_codPairP as
          [ F1 F2  (*  ( @'UnitMorCoMod (Pair F1 F2) )%sol  *)
          | _F1 _F2 _Z1 _Z1' _z1  (*  ( ~_1 @ _F2 o>CoMod _z1 )%sol  *)
          | _F1 _F2 _Z2 _Z2' _z2  (*   ( ~_2 @ _F1 o>CoMod _z2 )%sol  *)
          | L F1 F2 f1 f2 (*  ( << f1 ,CoMod f2 >> )%sol  *) ] .

      (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is
     ( @'UnitMorCoMod (Pair F1 F2) _o>CoMod^ ~_1 @ _ _o>CoMod^ z1Solz1'Sol ) *)
      - eexists. eexists. refine ( ~_1 @ F2 _o>CoMod^ z1Solz1'Sol )%sol .

      (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is ( ( ~_1
      @ _F2 o>CoMod _z1 ) _o>CoMod^ ( ~_1 @ _ _o>CoMod^ z1Solz1'Sol ) ) *)
      - have [:blurb]  _z1_o_g'Sol__z1_o_gSol :=
          (projT2 (projT2 (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_Pre
             len _ _ _ _ ( ~_1 @ _ _o>CoMod^ z1Solz1'Sol )%sol _ _z1 blurb)));
            first by abstract tac_degradeTransf gradeTransf_gg' .
        eexists. eexists.
        refine ( ~_1 @ _F2 _o>CoMod^ _z1_o_g'Sol__z1_o_gSol )%sol .

      (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is ( ( ~_2
      @ _F1 o>CoMod _z2 ) _o>CoMod^ ( ~_1 @ _ _o>CoMod^ z1Solz1'Sol ) ) *)
      - have [:blurb]  _z2_o_g'Sol__z2_o_gSol :=
          (projT2 (projT2 (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_Pre
              len _ _ _ _ ( ~_1 @ _ _o>CoMod^ z1Solz1'Sol )%sol _ _z2 blurb)));
            first by abstract tac_degradeTransf gradeTransf_gg' .
        eexists. eexists.
        refine ( ~_2 @ _F1 _o>CoMod^ _z2_o_g'Sol__z2_o_gSol )%sol .

      (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is ( << f1
      ,CoMod f2 >> _o>CoMod^ ( ~_1 @ _ _o>CoMod^ z1Solz1'Sol ) ) *)
      - have [:blurb] f1_o_z1Sol_f1_o_z1'Sol :=
          (projT2 (projT2 (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_Pre
                              len _ _ _ _ z1Solz1'Sol _ f1 blurb)));
            first by abstract tac_degradeTransf gradeTransf_gg' .
        eexists. eexists. refine ( f1_o_z1Sol_f1_o_z1'Sol )%sol .
    }

  (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is (fSol
  _o>CoMod^ ~_2 @ F1 _o>CoMod^ z2Solz2'Sol) *)
  * move: (Sol.Destruct_codPair.morCoMod_codPairP fSol) => fSol_codPairP.
    { destruct fSol_codPairP as
          [ F1 F2  (*  ( @'UnitMorCoMod (Pair F1 F2) )%sol  *)
          | _F1 _F2 _Z1 _Z1' _z1  (*  ( ~_1 @ _F2 o>CoMod _z1 )%sol  *)
          | _F1 _F2 _Z2 _Z2' _z2  (*   ( ~_2 @ _F1 o>CoMod _z2 )%sol  *)
          | L F1 F2 f1 f2 (*  ( << f1 ,CoMod f2 >> )%sol  *) ] .

      (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is
    ( @'UnitMorCoMod (Pair F1 F2) _o>CoMod^ ~_2 @ _ _o>CoMod^ z2Solz2'Sol  ) *)
      - eexists. eexists. refine ( ~_2 @ F1 _o>CoMod^ z2Solz2'Sol )%sol .

      (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is ( ( ~_1
      @ _F2 o>CoMod _z1 ) _o>CoMod^ ( ~_2 @ _ _o>CoMod^ z2Solz2'Sol ) ) *)
      - have [:blurb] _z1_o_g'Sol__z1_o_gSol :=
          (projT2 (projT2 (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_Pre
              len _ _ _ _ ( ~_2 @ _ _o>CoMod^ z2Solz2'Sol )%sol _ _z1 blurb)));
            first by abstract tac_degradeTransf gradeTransf_gg' .
        eexists. eexists.
        refine ( ~_1 @ _F2 _o>CoMod^ _z1_o_g'Sol__z1_o_gSol )%sol .

      (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is ( ( ~_2
      @ _F1 o>CoMod _z2 ) _o>CoMod^ ( ~_2 @ _ _o>CoMod^ z2Solz2'Sol ) ) *)
      - have [:blurb] _z2_o_g'Sol__z2_o_gSol :=
```

```
            (projT2 (projT2 (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_Pre
               len _ _ _ _ ( ~_2 @ _ _o>CoMod^ z2Solz2'Sol )%sol _ _z2 blurb )));
              first by abstract tac_degradeTransf gradeTransf_gg' .
          eexists. eexists.
          refine ( ~_2 @ _F1 _o>CoMod^ _z2_o_g'Sol__z2_o_gSol )%sol .

        (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is ( << f1
        ,CoMod f2 >> _o>CoMod^ ( ~_2 @ _ _o>CoMod^ z2Solz2'Sol ) ) *)
        - have [:blurb] f2_o_z2Sol_f2_o_z2'Sol :=
            (projT2 (projT2 (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_Pre
                              len _ _ _ _ z2Solz2'Sol _ f2 blurb)));
              first by abstract tac_degradeTransf gradeTransf_gg' .
          eexists. eexists. refine ( f2_o_z2Sol_f2_o_z2'Sol )%sol .
      }

    (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is (fSol
    _o>CoMod^ << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ) *)
    * { destruct fSol as
            [ F (* @'UnitMorCoMod F *)
            | _F1 _F2 Z1 z1Sol (* ~_1 @ F2 o>CoMod z1Sol *)
            | _F1 _F2 Z2 z2Sol (* ~_2 @ F1 o>CoMod z2Sol *)
            | L _F1 _F2 _f1Sol _f2Sol  (* << _f1Sol ,CoMod _f2Sol >> *) ] .

        (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is
        ( @'UnitMorCoMod F _o>CoMod^ << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ) *)
        - eexists. eexists. refine ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> )%sol.

        (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is ( ( ~_1
      @ _F2 o>CoMod z1Sol ) _o>CoMod^ << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ) *)
        - have [:blurb] z1Sol_o_g'Sol_z1Sol_o_gSol :=
            (projT2 (projT2 (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_Pre
      len _ _ _ _ ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> )%sol _ z1Sol blurb)));
              first by abstract tac_degradeTransf gradeTransf_gg' .
          eexists. eexists.
          refine ( ~_1 @ _F2 _o>CoMod^ z1Sol_o_g'Sol_z1Sol_o_gSol )%sol .

        (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is ( ( ~_2
     @ _F1 o>CoMod z2Sol ) _o>CoMod^ ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ) ) *)
        - have [:blurb] z2Sol_o_g'Sol_z2Sol_o_gSol :=
            (projT2 (projT2 (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_Pre
      len _ _ _ _ ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> )%sol _ z2Sol blurb)));
              first by abstract tac_degradeTransf gradeTransf_gg' .
          eexists. eexists.
          refine ( ~_2 @ _F1 _o>CoMod^ z2Sol_o_g'Sol_z2Sol_o_gSol )%sol .

          (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is ( <<
   _f1Sol ,CoMod _f2Sol >> _o>CoMod^ ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ) ) *)
          * have [:blurb] fSol_o_f1Sol_fSol_o_f1'Sol :=
              (projT2 (projT2 (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_Pre
          len _ _ _ _ f1Solf1'Sol _ ( << _f1Sol ,CoMod _f2Sol >> %sol) blurb)));
                first by abstract tac_degradeTransf gradeTransf_gg' .
            have [:blurb] fSol_o_f2Sol_fSol_o_f2'Sol :=
              (projT2 (projT2 (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_Pre
           len _ _ _ _ f2Solf2'Sol _ ( << _f1Sol ,CoMod _f2Sol >> %sol) blurb)));
                first by abstract tac_degradeTransf gradeTransf_gg' .
            eexists. eexists. refine ( << fSol_o_f1Sol_fSol_o_f1'Sol
                                    ,^CoMod fSol_o_f2Sol_fSol_o_f2'Sol >> )%sol .
      }
  }
Defined.

Arguments solveTransfCoMod_TransfCoMod_PolyMorCoMod_Pre
          !len _ _ _ _ !g'Sol_gSol _ _ _ : simpl nomatch .

Notation "f _o>CoMod^ g'g @ gradeTransf_gg'" :=
 (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_Pre _ _ _ _ _ g'g _ f gradeTransf_gg')
   (at level 40 , g'g at next level) : sol_scope.
```

```
Fixpoint solveTransfCoMod_TransfCoMod_PolyMorCoMod_PreP len {struct len} :
  forall (F G : obCoMod) (g'Sol gSol : 'morCoMod(0 F ~> G )0%sol)
    (g'Sol_gSol : 'transfCoMod(0 g'Sol ~> gSol )0%sol)
    (E : obCoMod) (fSol : 'morCoMod(0 E ~> F )0%sol)
    (gradeTransf_gg' : (gradeTransf (Sol.toPolyMor fSol
                              _o>CoMod^ Sol.toPolyTransf g'Sol_gSol) <= len)%coq_nat),
    ( ( (solveMorCoMod0 (Sol.toPolyMor fSol o>CoMod Sol.toPolyMor g'Sol)
         = projT1(fSol _o>CoMod^ g'Sol_gSol @ gradeTransf_gg')%sol) *
        (solveMorCoMod0 (Sol.toPolyMor fSol o>CoMod Sol.toPolyMor gSol)
         = projT1(projT2(fSol _o>CoMod^ g'Sol_gSol @ gradeTransf_gg')%sol)) ) *
(Sol.toPolyTransf (projT2(projT2(fSol _o>CoMod^ g'Sol_gSol @ gradeTransf_gg')%sol))
     <~~2 (Sol.toPolyMor fSol _o>CoMod^ Sol.toPolyTransf g'Sol_gSol)%poly ) )%type.
Proof.
  case : len => [ | len ].

  (* len is O *)
  - ( move => F G g'Sol gSol g'Sol_gSol E fSol gradeTransf_gg' );
      exfalso; abstract tac_degradeTransf gradeTransf_gg'.

  (* len is (S len) *)
  (* gg' is f _o>CoMod^ g'g , to  (fSol _o>CoMod^ g'Sol_gSol) *)
  - move => F G g'Sol gSol g'Sol_gSol E fSol gradeTransf_gg' .
    { destruct g'Sol_gSol as
          [ F G gSol (* @'UnitTransfCoMod gSol *)
          | F1 F2 Z1 z1Sol z1'Sol z1Solz1'Sol (* ~_1 @ F2 _o>CoMod^ z1Solz1'Sol *)
          | F1 F2 Z2 z2Sol z2'Sol z2Solz2'Sol (* ~_2 @ F1 _o>CoMod^ z2Solz2'Sol *)
          | L F1 F2 f1Sol f1'Sol f1Solf1'Sol f2Sol f2'Sol f2Solf2'Sol
(* << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> *) ] .

      (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is (fSol
      _o>CoMod^ @'UnitTransfCoMod gSol) *)
      * have fSol_o_gSol_prop :=
      (@solveMorCoMod0P _ _ ( (Sol.toPolyMor fSol) o>CoMod (Sol.toPolyMor gSol) )).
        set fSol_o_gSol := (@solveMorCoMod0 _ _
        ( (Sol.toPolyMor fSol) o>CoMod (Sol.toPolyMor gSol) )) in fSol_o_gSol_prop.
        subst fSol_o_gSol; move: fSol_o_gSol_prop ;
          clear ; abstract tac_reduce_solveMorCoMod0.

      (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is (fSol
      _o>CoMod^ ~_1 @ F2 _o>CoMod^ z1Solz1'Sol) *)
      * move: (Sol.Destruct_codPair.morCoMod_codPairP fSol) => fSol_codPairP.
        { destruct fSol_codPairP as
              [ F1 F2  (*  ( @'UnitMorCoMod (Pair F1 F2) )%sol  *)
              | _F1 _F2 _Z1 _Z1' _z1  (*  ( ~_1 @ _F2 o>CoMod _z1 )%sol  *)
              | _F1 _F2 _Z2 _Z2' _z2  (*   ( ~_2 @ _F1 o>CoMod _z2 )%sol  *)
              | L F1 F2 f1 f2 (*  ( << f1 ,CoMod f2 >> )%sol  *) ] .

          (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is
         ( @'UnitMorCoMod (Pair F1 F2) _o>CoMod^ ~_1 @ _ _o>CoMod^ z1Solz1'Sol ) *)
          - clear ; abstract tac_reduce_solveMorCoMod0.

          (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is ( ( ~_1
          @ _F2 o>CoMod _z1 ) _o>CoMod^ ( ~_1 @ _ _o>CoMod^ z1Solz1'Sol ) ) *)
          - simpl; set same_blurb := ( _ gradeTransf_gg' : ( _ <= len )%coq_nat) .
            move: (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_PreP
             len _ _ _ _ ( ~_1 @ _ _o>CoMod^ z1Solz1'Sol )%sol _ _z1 same_blurb ) .
            clear ; abstract tac_reduce_solveMorCoMod0.

          (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is ( ( ~_2
          @ _F1 o>CoMod _z2 ) _o>CoMod^ ( ~_1 @ _ _o>CoMod^ z1Solz1'Sol ) ) *)
          - simpl; set same_blurb := ( _ gradeTransf_gg' : ( _ <= len )%coq_nat) .
            move: (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_PreP
              len _ _ _ _ ( ~_1 @ _ _o>CoMod^ z1Solz1'Sol )%sol _ _z2 same_blurb) .
            clear ; abstract tac_reduce_solveMorCoMod0.

          (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is ( << f1
```

```
      ,CoMod f2 >> _o>CoMod^ ( ~_1 @ _ _o>CoMod^ z1Solz1'Sol ) ) *)
      - simpl; set same_blurb := ( _ gradeTransf_gg' : ( _ <= len )%coq_nat) .
        move: (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_PreP
                 len _ _ _ _ z1Solz1'Sol _ f1 same_blurb).
        clear ; abstract tac_reduce_solveMorCoMod0.
    }

  (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is (fSol
  _o>CoMod^ ~_2 @ F1 _o>CoMod^ z2Solz2'Sol) *)
  * move: (Sol.Destruct_codPair.morCoMod_codPairP fSol) => fSol_codPairP.
    { destruct fSol_codPairP as
          [ F1 F2  (*  ( @'UnitMorCoMod (Pair F1 F2) )%sol  *)
          | _F1 _F2 _Z1 _Z1' _z1  (*  ( ~_1 @ _F2 o>CoMod _z1 )%sol  *)
          | _F1 _F2 _Z2 _Z2' _z2  (*   ( ~_2 @ _F1 o>CoMod _z2 )%sol  *)
          | L F1 F2 f1 f2 (*  ( << f1 ,CoMod f2 >> )%sol  *) ] .

      (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is
     ( @'UnitMorCoMod (Pair F1 F2) _o>CoMod^ ~_2 @ _ _o>CoMod^ z2Solz2'Sol ) *)
      - clear ; abstract tac_reduce_solveMorCoMod0.

      (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is ( ( ~_1
      @ _F2 o>CoMod _z1 ) _o>CoMod^ ( ~_2 @ _ _o>CoMod^ z2Solz2'Sol ) ) *)
      - simpl; set same_blurb := ( _ gradeTransf_gg' : ( _ <= len )%coq_nat) .
        move: (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_PreP
           len _ _ _ _ ( ~_2 @ _ _o>CoMod^ z2Solz2'Sol )%sol _ _z1 same_blurb).
        clear ; abstract tac_reduce_solveMorCoMod0.

      (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is ( ( ~_2
      @ _F1 o>CoMod _z2 ) _o>CoMod^ ( ~_2 @ _ _o>CoMod^ z2Solz2'Sol ) ) *)
      - simpl; set same_blurb := ( _ gradeTransf_gg' : ( _ <= len )%coq_nat) .
        move: (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_PreP
          len _ _ _ _ ( ~_2 @ _ _o>CoMod^ z2Solz2'Sol )%sol _ _z2 same_blurb ).
        clear ; abstract tac_reduce_solveMorCoMod0.

      (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is ( << f1
      ,CoMod f2 >> _o>CoMod^ ( ~_2 @ _ _o>CoMod^ z2Solz2'Sol ) ) *)
      - simpl; set same_blurb := ( _ gradeTransf_gg' : ( _ <= len )%coq_nat) .
        move: (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_PreP
                 len _ _ _ _ z2Solz2'Sol _ f2 same_blurb).
        clear ; abstract tac_reduce_solveMorCoMod0.
    }

  (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is (fSol
  _o>CoMod^ << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ) *)
  * { destruct fSol as
          [ F (* @'UnitMorCoMod F *)
          | _F1 _F2 Z1 z1Sol (* ~_1 @ F2 o>CoMod z1Sol *)
          | _F1 _F2 Z2 z2Sol (* ~_2 @ F1 o>CoMod z2Sol *)
          | L _F1 _F2 _f1Sol _f2Sol  (* << _f1Sol ,CoMod _f2Sol >> *) ] .

      (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is
      ( @'UnitMorCoMod F _o>CoMod^ << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ) *)
      - clear ; abstract tac_reduce_solveMorCoMod0.

      (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is ( ( ~_1
    @ _F2 o>CoMod z1Sol ) _o>CoMod^ << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ) *)
      - simpl; set same_blurb := ( _ gradeTransf_gg' : ( _ <= len )%coq_nat) .
        move: (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_PreP
len _ _ _ _ ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> )%sol _ z1Sol same_blurb) .
        clear ; abstract tac_reduce_solveMorCoMod0.

      (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is ( ( ~_2
 @ _F1 o>CoMod z2Sol ) _o>CoMod^ ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ) ) *)
      - simpl; set same_blurb := ( _ gradeTransf_gg' : ( _ <= len )%coq_nat) .
        move: (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_PreP
len _ _ _ _ ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> )%sol _ z2Sol same_blurb) .
        clear ; abstract tac_reduce_solveMorCoMod0.
```

```
          (* gg' is f _o>CoMod^ g'g , to (fSol _o>CoMod^ g'Sol_gSol) , is ( <<
    _f1Sol ,CoMod _f2Sol >> _o>CoMod^ ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ) ) *)
          * simpl; set same_blurb1 := ( _ gradeTransf_gg' : ( _ <= len )%coq_nat).
            move: (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_PreP
        len _ _ _ _ f1Solf1'Sol _ ( << _f1Sol ,CoMod _f2Sol >> %sol) same_blurb1).
            simpl; set same_blurb2 := ( _ gradeTransf_gg' : ( _ <= len )%coq_nat).
            move: (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_PreP
        len _ _ _ _ f2Solf2'Sol _ ( << _f1Sol ,CoMod _f2Sol >> %sol) same_blurb2).
            clear ; abstract tac_reduce_solveMorCoMod0.
        }
    }
Qed.

Fixpoint solveTransfCoMod_TransfCoMod_PolyMorCoMod_Post len {struct len} :
  forall (G H : obCoMod) (hSol : 'morCoMod(0 G ~> H )0%sol)
    (F : obCoMod) (gSol g'Sol : 'morCoMod(0 F ~> G )0%sol)
    (g'Sol_gSol : 'transfCoMod(0 g'Sol ~> gSol )0%sol)
    (gradeTransf_gg' : (gradeTransf (Sol.toPolyTransf g'Sol_gSol
                                     ^o>CoMod_ Sol.toPolyMor hSol) <= len)%coq_nat),
    {g_ : 'morCoMod(0 F ~> H )0%sol &
          {g'_ : 'morCoMod(0 F ~> H )0%sol &
                 'transfCoMod(0 g_ ~> g'_ )0%sol } }.
Proof.
  case : len => [ | len ].

  (* len is O *)
  - ( move => G H hSol F gSol g'Sol g'Sol_gSol gradeTransf_gg' );
      exfalso; abstract tac_degradeTransf gradeTransf_gg'.

  (* len is (S len) *)
  (* gg' is g'g ^o>CoMod_ h , to  (g'Sol_gSol ^o>CoMod_ hSol) *)
  - move => G H hSol F gSol g'Sol g'Sol_gSol gradeTransf_gg'.
    { destruct g'Sol_gSol as
          [ F G gSol (* @'UnitTransfCoMod gSol *)
          | F1 F2 Z1 z1Sol z1'Sol z1Solz1'Sol (* ~_1 @ F2 _o>CoMod^ z1Solz1'Sol *)
          | F1 F2 Z2 z2Sol z2'Sol z2Solz2'Sol (* ~_2 @ F1 _o>CoMod^ z2Solz2'Sol *)
          | L F1 F2 f1Sol f1'Sol f1Solf1'Sol f2Sol f2'Sol f2Solf2'Sol
(* << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> *) ] .

      (* gg' is g'g ^o>CoMod_ h , to (g'Sol_gSol ^o>CoMod_ hSol) , is
      (@'UnitTransfCoMod gSol ^o>CoMod_ hSol) *)
      * have gSol_o_hSol_prop :=
      (@solveMorCoMod0P _ _ ( (Sol.toPolyMor gSol) o>CoMod (Sol.toPolyMor hSol) )).
        set gSol_o_hSol := (@solveMorCoMod0 _ _
        ( (Sol.toPolyMor gSol) o>CoMod (Sol.toPolyMor hSol) )) in gSol_o_hSol_prop.
        eexists. eexists. refine ( @'UnitTransfCoMod  gSol_o_hSol )%sol .

      (* gg' is g'g ^o>CoMod_ h , to (g'Sol_gSol ^o>CoMod_ hSol) , is (~_1 @ F2
      _o>CoMod^ z1Solz1'Sol ^o>CoMod_ hSol) *)
      * have [:blurb] z1Sol_o_hSol_z1'Sol_o_hSol :=
          (projT2 (projT2 (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_Post
                           len _ _ hSol _ _ _ z1Solz1'Sol blurb )));
            first by abstract tac_degradeTransf gradeTransf_gg' .
        eexists. eexists.
        refine ( ~_1 @ F2 _o>CoMod^ z1Sol_o_hSol_z1'Sol_o_hSol )%sol .

      (* gg' is g'g ^o>CoMod_ h , to (g'Sol_gSol ^o>CoMod_ hSol) , is (~_2 @ F1
      _o>CoMod^ z2Solz2'Sol ^o>CoMod_ hSol) *)
      * have [:blurb] z2Sol_o_hSol_z2'Sol_o_hSol :=
          (projT2 (projT2 (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_Post
                           len _ _ hSol _ _ _ z2Solz2'Sol blurb)));
            first by abstract tac_degradeTransf gradeTransf_gg' .

        eexists. eexists. refine ( ~_2 @ F1 _o>CoMod^ z2Sol_o_hSol_z2'Sol_o_hSol )%sol .

      (* gg' is g'g ^o>CoMod_ h , to (g'Sol_gSol ^o>CoMod_ hSol) , is ( <<
```

```
      f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ^o>CoMod_ hSol ) *)
      * move: (Sol.Destruct_domPair.morCoMod_domPairP hSol) => hSol_domPairP.
        { destruct hSol_domPairP as
              [ F1 F2  (*  ( @'UnitMorCoMod (Pair F1 F2) )%sol  *)
              | F1 F2 Z1 z1  (*  ( ~_1 @ F2 o>CoMod z1 )%sol  *)
              | F1 F2 Z2 z2  (*   ( ~_2 @ F1 o>CoMod z2 )%sol  *)
              | M M' F1 F2 f1 f2 (*  ( << f1 ,CoMod f2 >> )%sol  *) ] .

          (* gg' is g'g ^o>CoMod_ h , to (g'Sol_gSol ^o>CoMod_ hSol) , is ( <<
      f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ^o>CoMod_ @'UnitMorCoMod (Pair F1 F2) ) *)
          - eexists. eexists. refine ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> )%sol .

          (* gg' is g'g ^o>CoMod_ h , to (g'Sol_gSol ^o>CoMod_ hSol) , is ( <<
          f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ^o>CoMod_ ~_1 @ F2 o>CoMod z1 *)
          - have [:blurb] f1Sol_o_z1_f1'Sol_o_z1 :=
              (projT2 (projT2 (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_Post
                                   len _ _ z1 _ _ _ f1Solf1'Sol blurb)));
                first by abstract tac_degradeTransf gradeTransf_gg' .
            eexists. eexists. refine ( f1Sol_o_z1_f1'Sol_o_z1 )%sol .

          (* gg' is g'g ^o>CoMod_ h , to (g'Sol_gSol ^o>CoMod_ hSol) , is ( <<
          f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ^o>CoMod_ ~_2 @ F1 o>CoMod z2 *)
          - have [:blurb] f2Sol_o_z2_f2'Sol_o_z2 :=
              (projT2 (projT2 (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_Post
                                   len _ _ z2 _ _ _ f2Solf2'Sol blurb )));
                first by abstract tac_degradeTransf gradeTransf_gg' .
            eexists. eexists. refine ( f2Sol_o_z2_f2'Sol_o_z2 )%sol .

          (* gg' is g'g ^o>CoMod_ h , to (g'Sol_gSol ^o>CoMod_ hSol) , is ( <<
          f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ^o>CoMod_ << f1 ,CoMod f2 >> *)
          - have [:blurb1] g'Sol_o_f1_gSol_o_f1 :=
              (projT2 (projT2 (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_Post
        len _ _ f1 _ _ _ ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> %sol ) blurb1 )));
                first by abstract tac_degradeTransf gradeTransf_gg' .

            have [:blurb2] g'Sol_o_f2_gSol_o_f2 :=
              (projT2 (projT2 (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_Post
        len _ _ f2 _ _ _ ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> %sol ) blurb2 )));
                first by abstract tac_degradeTransf gradeTransf_gg' .
            eexists. eexists.
            refine ( << g'Sol_o_f1_gSol_o_f1 ,^CoMod g'Sol_o_f2_gSol_o_f2 >> )%sol.
        }
    }
Defined.

Arguments solveTransfCoMod_TransfCoMod_PolyMorCoMod_Post
          !len _ _ _ _ _ _ !g'Sol_gSol _ : simpl nomatch .

Notation "g'g ^o>CoMod_ h @ gradeTransf_g'g" :=
  (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_Post
      _ _ _ h _ _ _ g'g gradeTransf_g'g)
    (at level 40 , h at next level) : sol_scope.

Fixpoint solveTransfCoMod_TransfCoMod_PolyMorCoMod_PostP len {struct len} :
  forall (G H : obCoMod) (hSol : 'morCoMod(0 G ~> H )0%sol)
    (F : obCoMod) (gSol g'Sol : 'morCoMod(0 F ~> G )0%sol)
    (g'Sol_gSol : 'transfCoMod(0 g'Sol ~> gSol )0%sol)
    (gradeTransf_gg' : (gradeTransf (Sol.toPolyTransf g'Sol_gSol
                                       ^o>CoMod_ Sol.toPolyMor hSol) <= len)%coq_nat),
    ( ( (solveMorCoMod0 (Sol.toPolyMor g'Sol o>CoMod Sol.toPolyMor hSol)
         = projT1(g'Sol_gSol ^o>CoMod_ hSol @ gradeTransf_gg')%sol) *
        (solveMorCoMod0 (Sol.toPolyMor gSol o>CoMod Sol.toPolyMor hSol)
         = projT1(projT2(g'Sol_gSol ^o>CoMod_ hSol @ gradeTransf_gg')%sol)) ) *
      ( Sol.toPolyTransf (projT2(projT2(g'Sol_gSol ^o>CoMod_ hSol
                                             @ gradeTransf_gg')%sol))
     <~~2 (Sol.toPolyTransf g'Sol_gSol ^o>CoMod_ Sol.toPolyMor hSol)%poly ) )%type.
Proof.
```

```
  case : len => [ | len ].

  (* len is O *)
  - ( move => G H hSol F gSol g'Sol g'Sol_gSol gradeTransf_gg' );
      exfalso; abstract tac_degradeTransf gradeTransf_gg'.

  (* len is (S len) *)
  (* gg' is g'g ^o>CoMod_ h , to  (g'Sol_gSol ^o>CoMod_ hSol) *)
  - move => G H hSol F gSol g'Sol g'Sol_gSol gradeTransf_gg'.
    { destruct g'Sol_gSol as
          [ F G gSol (* @'UnitTransfCoMod gSol *)
          | F1 F2 Z1 z1Sol z1'Sol z1Solz1'Sol (* ~_1 @ F2 _o>CoMod^ z1Solz1'Sol *)
          | F1 F2 Z2 z2Sol z2'Sol z2Solz2'Sol (* ~_2 @ F1 _o>CoMod^ z2Solz2'Sol *)
          | L F1 F2 f1Sol f1'Sol f1Solf1'Sol f2Sol f2'Sol f2Solf2'Sol
(* << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> *) ] .

      (* gg' is g'g ^o>CoMod_ h , to (g'Sol_gSol ^o>CoMod_ hSol) , is
      (@'UnitTransfCoMod gSol ^o>CoMod_ hSol) *)
      * have gSol_o_hSol_prop :=
      (@solveMorCoMod0P _ _ ( (Sol.toPolyMor gSol) o>CoMod (Sol.toPolyMor hSol) )).
        set gSol_o_hSol := (@solveMorCoMod0 _ _ ( (Sol.toPolyMor gSol)
                             o>CoMod (Sol.toPolyMor hSol) )) in gSol_o_hSol_prop.
        subst gSol_o_hSol; move:  gSol_o_hSol_prop ;
          clear ; abstract tac_reduce_solveMorCoMod0.

      (* gg' is g'g ^o>CoMod_ h , to (g'Sol_gSol ^o>CoMod_ hSol) , is (~_1 @ F2
      _o>CoMod^ z1Solz1'Sol ^o>CoMod_ hSol) *)
      * simpl; set same_blurb := ( _ gradeTransf_gg' : ( _ <= len )%coq_nat) .
        move: (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_PostP
                len _ _ hSol _ _ _ z1Solz1'Sol same_blurb );
          clear ; abstract tac_reduce_solveMorCoMod0.

      (* gg' is g'g ^o>CoMod_ h , to (g'Sol_gSol ^o>CoMod_ hSol) , is (~_2 @ F1
      _o>CoMod^ z2Solz2'Sol ^o>CoMod_ hSol) *)
      * simpl; set same_blurb := ( _ gradeTransf_gg' : ( _ <= len )%coq_nat) .
        move: (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_PostP
                len _ _ hSol _ _ _ z2Solz2'Sol same_blurb).
        clear ; abstract tac_reduce_solveMorCoMod0.

      (* gg' is g'g ^o>CoMod_ h , to (g'Sol_gSol ^o>CoMod_ hSol) , is ( <<
      f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ^o>CoMod_ hSol ) *)
      * move: (Sol.Destruct_domPair.morCoMod_domPairP hSol) => hSol_domPairP.
        { destruct hSol_domPairP as
            [ F1 F2  (*  ( @'UnitMorCoMod (Pair F1 F2) )%sol  *)
            | F1 F2 Z1 z1  (*  ( ~_1 @ F2 o>CoMod z1 )%sol  *)
            | F1 F2 Z2 z2  (*   ( ~_2 @ F1 o>CoMod z2 )%sol  *)
            | M M' F1 F2 f1 f2 (*  ( << f1 ,CoMod f2 >> )%sol  *) ] .

          (* gg' is g'g ^o>CoMod_ h , to (g'Sol_gSol ^o>CoMod_ hSol) , is ( <<
      f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ^o>CoMod_ @'UnitMorCoMod (Pair F1 F2) ) *)
          - clear ; abstract tac_reduce_solveMorCoMod0.

          (* gg' is g'g ^o>CoMod_ h , to (g'Sol_gSol ^o>CoMod_ hSol) , is ( <<
          f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ^o>CoMod_ ~_1 @ F2 o>CoMod z1 *)
          - simpl; set same_blurb := ( _ gradeTransf_gg' : ( _ <= len )%coq_nat) .
            move: (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_PostP
                    len _ _ z1 _ _ _ f1Solf1'Sol same_blurb).
            clear ; abstract tac_reduce_solveMorCoMod0.

          (* gg' is g'g ^o>CoMod_ h , to (g'Sol_gSol ^o>CoMod_ hSol) , is ( <<
          f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ^o>CoMod_ ~_2 @ F1 o>CoMod z2 *)
          - simpl; set same_blurb := ( _ gradeTransf_gg' : ( _ <= len )%coq_nat) .
            move: (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_PostP
                    len _ _ z2 _ _ _ f2Solf2'Sol same_blurb ).
            clear ; abstract tac_reduce_solveMorCoMod0.

          (* gg' is g'g ^o>CoMod_ h , to (g'Sol_gSol ^o>CoMod_ hSol) , is ( <<
```

```
        f1Solf1'Sol ,^CoMod f2Solf2'Sol >> ^o>CoMod_ << f1 ,CoMod f2 >> *)
        - simpl; set same_blurb1 := ( _ gradeTransf_gg' : ( _ <= len )%coq_nat) .
          move: (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_PostP
    len _ _ f1 _ _ _ ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> %sol ) same_blurb1).
          simpl; set same_blurb2 := ( _ gradeTransf_gg' : ( _ <= len )%coq_nat) .
          move: (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_PostP
    len _ _ f2 _ _ _ ( << f1Solf1'Sol ,^CoMod f2Solf2'Sol >> %sol ) same_blurb2).
          clear ; abstract tac_reduce_solveMorCoMod0.
      }
  }
Qed.

Fixpoint solveTransfCoMod len {struct len} :
  forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 )
    (gg' : 'transfCoMod(0 g ~> g' )0 ),
  forall gradeTransf_gg' : (gradeTransf gg' <= len)%coq_nat,
    { g_g'_ : { g_ : 'morCoMod(0 F ~> G )0 %sol &
        { g'_ : 'morCoMod(0 F ~> G )0 %sol & 'transfCoMod(0 g_ ~> g'_ )0 %sol } }
    | ( ( solveMorCoMod0 g = projT1 g_g'_ ) *
        ( solveMorCoMod0 g' = projT1 (projT2 g_g'_) ) )%type }.
Proof.
  case : len => [ | len ].

  (* len is O *)
  - ( move => F G g g' gg' gradeTransf_gg' ); exfalso; abstract tac_degradeTransf gradeTransf_gg'.

  (* len is (S len) *)
  - move => F G g g' gg'; case : F G g g' / gg' =>
    [ F G g g' g'g g'0 g'' g''g' eqMor (* g''g' o^CoMod g'g *)
    | F G g g' g'g E f (* f _o>CoMod^ g'g *)
    | G H h F g g' g'g (* g'g ^o>CoMod_ h *)
    | F G g (* @'UnitTransfCoMod g *)
    | F1 F2 Z1 z1 z1' z1z1' (* ~_1 @ F2 _o>CoMod^ z1z1' *)
    | F1 F2 Z2 z2 z2' z2z2' (* ~_2 @ F1 _o>CoMod^ z2z2' *)
    | L F1 F2 f1 f1' f1f1' f2 f2' f2f2' (* << f1f1' ,^CoMod f2f2' >> *)
    ] gradeTransf_gg' .

    (* gg' is g''g' o^CoMod g'g *)
    + all: cycle 1.
    (* gg' is f _o>CoMod^ g'g *)
    + all: cycle 1.
    (* gg' is g'g ^o>CoMod_ h *)
    + all: cycle 1.

    (* gg' is @'UnitTransfCoMod g *)
    + have  gSol_prop := (@solveMorCoMod0P _ _ g).
      set gSol :=  (@solveMorCoMod0 _ _ g) in gSol_prop.
      unshelve eexists. eexists. eexists. refine ( @'UnitTransfCoMod gSol )%sol.
      clear ; abstract tac_reduce_solveMorCoMod0.

    (* gg' is ~_1 @ F2 _o>CoMod^ z1z1' *)
    + have [:blurb] z1Sol_z1'Sol_prop :=
        (proj2_sig (solveTransfCoMod len _ _ _ _ z1z1' blurb));
          first by abstract tac_degradeTransf gradeTransf_gg'.
      move: (solveTransfCoMod len _ _ _ _ z1z1' blurb) z1Sol_z1'Sol_prop
      =>  z1Sol_z1'Sol z1Sol_z1'Sol_prop.
      unshelve eexists. eexists. eexists.
      refine ( ~_1 @ F2 _o>CoMod^ (projT2 (projT2 (proj1_sig z1Sol_z1'Sol))))%sol.
      move:  z1Sol_z1'Sol_prop; clear; abstract tac_reduce_solveMorCoMod0.

    (* gg' is ~_2 @ F1 _o>CoMod^ z2z2' *)
    + have [:blurb] z2Sol_z2'Sol_prop :=
        (proj2_sig (solveTransfCoMod len _ _ _ _ z2z2' blurb));
          first by abstract tac_degradeTransf gradeTransf_gg'.
      move: (solveTransfCoMod len _ _ _ _ z2z2' blurb) z2Sol_z2'Sol_prop
      =>  z2Sol_z2'Sol z2Sol_z2'Sol_prop.
      unshelve eexists. eexists. eexists.
```

```
    refine ( ~_2 @ F1 _o>CoMod^ (projT2 (projT2 (proj1_sig z2Sol_z2'Sol))))%sol.
    move:  z2Sol_z2'Sol_prop; clear; abstract tac_reduce_solveMorCoMod0.

  (* gg' is << f1f1' ,^CoMod f2f2' >> *)
  + have [:blurb1] f1Sol_f1'Sol_prop :=
      (proj2_sig (solveTransfCoMod len _ _ _ _ f1f1' blurb1));
        first by abstract tac_degradeTransf gradeTransf_gg'.
    move: (solveTransfCoMod len _ _ _ _ f1f1' blurb1) f1Sol_f1'Sol_prop
    => f1Sol_f1'Sol f1Sol_f1'Sol_prop .
    have [:blurb] f2Sol_f2'Sol_prop :=
      (proj2_sig (solveTransfCoMod len _ _ _ _ f2f2' blurb));
        first by abstract tac_degradeTransf gradeTransf_gg'.
    move: (solveTransfCoMod len _ _ _ _ f2f2' blurb) f2Sol_f2'Sol_prop
    => f2Sol_f2'Sol f2Sol_f2'Sol_prop .
    unshelve eexists. eexists. eexists.
    refine ( << (projT2 (projT2 (proj1_sig f1Sol_f1'Sol)))
                ,^CoMod  (projT2 (projT2 (proj1_sig f2Sol_f2'Sol))) >> )%sol.
    move: f1Sol_f1'Sol_prop f2Sol_f2'Sol_prop;
      clear; abstract tac_reduce_solveMorCoMod0.

  (* gg' is g''g' o^CoMod g'g *)
  + have solveTransfCoMod_len := solveTransfCoMod len. clear solveTransfCoMod.

    Definition solveTransfCoMod_sub_PolyTransfCoMod :
      forall (len : nat) (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0)
        (g'g : 'transfCoMod(0 g' ~> g )0) (g'0 g'' : 'morCoMod(0 F ~> G )0)
        (g''g' : 'transfCoMod(0 g'' ~> g'0 )0) (eqMor : g'0 <~>1 g')
        (gradeTransf_gg' : (gradeTransf (g''g' o^CoMod g'g # eqMor)
                             <= len.+1)%coq_nat)
        (solveTransfCoMod_len : forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0)
                                   (gg' : 'transfCoMod(0 g ~> g' )0),
            (gradeTransf gg' <= len)%coq_nat ->
            {g_g'_ : {g_ : 'morCoMod(0 F ~> G )0%sol &
                           {g'_ : 'morCoMod(0 F ~> G )0%sol &
                                  'transfCoMod(0 g_ ~> g'_ )0%sol}} |
             ((solveMorCoMod0 g = projT1 g_g'_) *
              (solveMorCoMod0 g' = projT1 (projT2 g_g'_)))%type}),
        {g_g'_ : {g_ : 'morCoMod(0 F ~> G )0%sol &
                       {g'_ : 'morCoMod(0 F ~> G )0%sol &
                              'transfCoMod(0 g_ ~> g'_ )0%sol}} |
         ((solveMorCoMod0 g'' = projT1 g_g'_) *
          (solveMorCoMod0 g = projT1 (projT2 g_g'_)))%type}.
    Proof.
      intros; have [:blurb_] g''Sol_prop :=
        (fst ( ( (proj2_sig (solveTransfCoMod_len _ _ _ _ g''g' blurb_)))));
          first by abstract tac_degradeTransf gradeTransf_gg'.
      move: (proj1_sig ( ( (solveTransfCoMod_len _ _ _ _ g''g' blurb_))))
  g''Sol_prop (snd ( ( (proj2_sig (solveTransfCoMod_len _ _ _ _ g''g' blurb_)))))
      => g''Sol_g'Sol g''Sol_prop g'Sol0_prop.
      have [:blurb'] g'Sol_prop :=
        (fst ( ( (proj2_sig (solveTransfCoMod_len _ _ _ _ g'g blurb')))));
          first by abstract tac_degradeTransf gradeTransf_gg'.
      move: (proj1_sig ( ( (solveTransfCoMod_len _ _ _ _ g'g blurb'))))
     g'Sol_prop (snd ( ( (proj2_sig (solveTransfCoMod_len _ _ _ _ g'g blurb')))))
      => g'Sol_gSol g'Sol_prop gSol_prop.
      clear solveTransfCoMod_len.
      have eqMor' : (Sol.toPolyMor (projT1 (projT2 g''Sol_g'Sol))
                               <~>1 (Sol.toPolyMor (projT1 g'Sol_gSol))%poly)
        by abstract
             ( apply: EqMorCoMod.eq_convMorCoMod;
               rewrite -g'Sol0_prop  -g'Sol_prop ;
               congr Sol.toPolyMor; congr solveMorCoMod0;
               apply: EqMorCoMod.convMorCoMod_eq;
               apply: eqMor
             ) .
      clear g'Sol0_prop g'Sol_prop .
      have [:blurb] g''Sol_g'Sol_o'_g'Sol_gSol_prop :=
```

```
    (fst (@solveTransfCoMod_PolyTransfCoModP
            (gradeTransf ((Sol.toPolyTransf (projT2 (projT2 g''Sol_g'Sol)))
           o^CoMod (Sol.toPolyTransf (projT2 (projT2 g'Sol_gSol))) # eqMor'))
            _ _ _ _ (projT2 (projT2 g'Sol_gSol))
            _ _ (projT2 (projT2 g''Sol_g'Sol)) eqMor' blurb eqMor'
    (*memo: this argument is same eqMor' as earlier argument *) ));
      first by clear; abstract reflexivity.
  move: (@solveTransfCoMod_PolyTransfCoMod
           (gradeTransf ((Sol.toPolyTransf (projT2 (projT2 g''Sol_g'Sol)))
           o^CoMod (Sol.toPolyTransf (projT2 (projT2 g'Sol_gSol))) # eqMor'))
           _ _ _ _ (projT2 (projT2 g'Sol_gSol))
           _ _ (projT2 (projT2 g''Sol_g'Sol)) eqMor' blurb )
          g''Sol_g'Sol_o'_g'Sol_gSol_prop
  => g''Sol_g'Sol_o'_g'Sol_gSol g''Sol_g'Sol_o'_g'Sol_gSol_prop.
  exists  g''Sol_g'Sol_o'_g'Sol_gSol.
  move: g''Sol_prop gSol_prop g''Sol_g'Sol_o'_g'Sol_gSol_prop;
    clear; abstract tac_reduce_solveMorCoMod0.
Defined.

apply: (solveTransfCoMod_sub_PolyTransfCoMod gradeTransf_gg'
                                             solveTransfCoMod_len).

(* gg' is f _o>CoMod^ g'g , to  (fSol _o>CoMod^ g'Sol_gSol) *)
+ have solveTransfCoMod_len := solveTransfCoMod len. clear solveTransfCoMod.

Definition solveTransfCoMod_sub_TransfCoMod_PolyMorCoMod_Pre :
  forall (len : nat) (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0)
    (g'g : 'transfCoMod(0 g' ~> g )0) (E : obCoMod)
    (f : 'morCoMod(0 E ~> F )0)
    (gradeTransf_gg' : (gradeTransf (f _o>CoMod^ g'g) <= len.+1)%coq_nat)
    (solveTransfCoMod_len : forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0)
                          (gg' : 'transfCoMod(0 g ~> g' )0),
        (gradeTransf gg' <= len)%coq_nat ->
        {g_g'_ : {g_ : 'morCoMod(0 F ~> G )0%sol &
                 {g'_ : 'morCoMod(0 F ~> G )0%sol &
                        'transfCoMod(0 g_ ~> g'_ )0%sol}} |
         ((solveMorCoMod0 g = projT1 g_g'_) *
          (solveMorCoMod0 g' = projT1 (projT2 g_g'_)))%type}),
    {g_g'_  : {g_ : 'morCoMod(0 E ~> G )0%sol &
               {g'_ : 'morCoMod(0 E ~> G )0%sol &
                      'transfCoMod(0 g_ ~> g'_ )0%sol}} |
     ((solveMorCoMod0 (f o>CoMod g') = projT1 g_g'_) *
      (solveMorCoMod0 (f o>CoMod g) = projT1 (projT2 g_g'_)))%type}.
Proof.
  intros. have [:blurb] g'Sol_gSol_prop :=
            (proj2_sig (solveTransfCoMod_len _ _ _ _ g'g blurb));
              first by abstract tac_degradeTransf gradeTransf_gg'.
  move: (solveTransfCoMod_len _ _ _ _ g'g blurb) g'Sol_gSol_prop
  => g'Sol_gSol g'Sol_gSol_prop.
  have fSol_prop := (@solveMorCoMod0P _ _ f).
  set fSol := (@solveMorCoMod0 _ _ f) in fSol_prop.
  have [:blurb'] fSol_o_g'Sol_gSol_prop :=
    (fst (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_PreP
            (gradeTransf ((Sol.toPolyMor fSol)
      _o>CoMod^ (Sol.toPolyTransf (projT2 (projT2 (proj1_sig g'Sol_gSol))))))
            _ _ _ _ (projT2 (projT2 (proj1_sig g'Sol_gSol))) _ fSol blurb'));
      first by clear; abstract reflexivity.
  move: (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_Pre
           (gradeTransf ((Sol.toPolyMor fSol)
     _o>CoMod^ (Sol.toPolyTransf (projT2 (projT2 (proj1_sig g'Sol_gSol))))))
           _ _ _ _ (projT2 (projT2 (proj1_sig g'Sol_gSol)))
           _ fSol blurb') fSol_o_g'Sol_gSol_prop
  => fSol_o_g'Sol_gSol fSol_o_g'Sol_gSol_prop.
  exists fSol_o_g'Sol_gSol .
  subst fSol; move: g'Sol_gSol_prop fSol_o_g'Sol_gSol_prop;
    clear; abstract tac_reduce_solveMorCoMod0.
Defined.
```

```
      apply: (solveTransfCoMod_sub_TransfCoMod_PolyMorCoMod_Pre
                gradeTransf_gg' solveTransfCoMod_len).

    (* gg' is g'g ^o>CoMod_ h *)
    + have solveTransfCoMod_len := solveTransfCoMod len. clear solveTransfCoMod.

      Definition solveTransfCoMod_sub_TransfCoMod_PolyMorCoMod_Post :
        forall (len : nat) (G H : obCoMod) (h : 'morCoMod(0 G ~> H )0)
          (F : obCoMod) (g g' : 'morCoMod(0 F ~> G )0)
          (g'g : 'transfCoMod(0 g' ~> g )0)
          (gradeTransf_gg' : (gradeTransf (g'g ^o>CoMod_ h) <= len.+1)%coq_nat)
          (solveTransfCoMod_len : forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0)
                                    (gg' : 'transfCoMod(0 g ~> g' )0),
              (gradeTransf gg' <= len)%coq_nat ->
              {g_g'_ : {g_ : 'morCoMod(0 F ~> G )0%sol &
                             {g'_ : 'morCoMod(0 F ~> G )0%sol &
                                    'transfCoMod(0 g_ ~> g'_ )0%sol}} |
               ((solveMorCoMod0 g = projT1 g_g'_) *
                (solveMorCoMod0 g' = projT1 (projT2 g_g'_)))%type}),
          {g_g'_ : {g_ : 'morCoMod(0 F ~> H )0%sol &
                         {g'_ : 'morCoMod(0 F ~> H )0%sol &
                                'transfCoMod(0 g_ ~> g'_ )0%sol}} |
           ((solveMorCoMod0 (g' o>CoMod h) = projT1 g_g'_) *
            (solveMorCoMod0 (g o>CoMod h) = projT1 (projT2 g_g'_)))%type}.
      Proof.
        intros; have [:blurb] g'Sol_gSol_prop :=
          (proj2_sig (solveTransfCoMod_len _ _ _ _ g'g blurb));
            first by abstract tac_degradeTransf gradeTransf_gg'.
        move: (solveTransfCoMod_len _ _ _ _ g'g blurb) g'Sol_gSol_prop
        => g'Sol_gSol g'Sol_gSol_prop.
        have hSol_prop := (@solveMorCoMod0P _ _ h).
        set hSol := (@solveMorCoMod0 _ _ h) in hSol_prop.
        have [:blurb'] g'Sol_gSol_o_hSol_prop :=
          (fst (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_PostP
          (gradeTransf ((Sol.toPolyTransf (projT2 (projT2 (proj1_sig g'Sol_gSol))))
                        ^o>CoMod_ (Sol.toPolyMor hSol)))
          _ _ hSol _ _ _ (projT2 (projT2 (proj1_sig g'Sol_gSol))) blurb'));
            first by clear; abstract reflexivity.
        move: (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_Post
          (gradeTransf ((Sol.toPolyTransf (projT2 (projT2 (proj1_sig g'Sol_gSol))))
                        ^o>CoMod_ (Sol.toPolyMor hSol))) _ _ hSol _ _ _
          (projT2 (projT2 (proj1_sig g'Sol_gSol))) blurb') g'Sol_gSol_o_hSol_prop
        => g'Sol_gSol_o_hSol g'Sol_gSol_o_hSol_prop.
        exists g'Sol_gSol_o_hSol .
        move: g'Sol_gSol_prop g'Sol_gSol_o_hSol_prop;
          clear; abstract tac_reduce_solveMorCoMod0.
      Defined.

      apply: (solveTransfCoMod_sub_TransfCoMod_PolyMorCoMod_Post
                gradeTransf_gg' solveTransfCoMod_len).
Defined.

Arguments solveTransfCoMod !len _ _ _ _ !gg' _ : clear implicits , simpl nomatch .

Lemma solveTransfCoMod_len :
  forall len (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 )
    (gg' : 'transfCoMod(0 g ~> g' )0 )
    (gradeTransf_gg'_len : (gradeTransf gg' <= len)%coq_nat),
  forall len' (gradeTransf_gg'_len' : (gradeTransf gg' <= len')%coq_nat),
    @solveTransfCoMod len _ _ _ _ _ gradeTransf_gg'_len
    = @solveTransfCoMod len' _ _ _ _ _ gradeTransf_gg'_len' .
Proof.
  induction len as [ | len ].
  - ( move => ? ? ? ? ?  gradeTransf_gg'_len ); exfalso;
     clear -gradeTransf_gg'_len; by abstract tac_degradeTransf gradeTransf_gg'_len.
  - intros. destruct len'.
```

```
    + exfalso; clear -gradeTransf_gg'_len';
        by abstract tac_degradeTransf gradeTransf_gg'_len'.
    + destruct gg' ;  cycle 3.
      * reflexivity.
      * simpl. erewrite IHlen. reflexivity.
      * simpl. erewrite IHlen. reflexivity.
      * simpl. unfold ssr_have.
        erewrite (IHlen _ _ _ _ gg'1).
        erewrite (IHlen _ _ _ _ gg'2). reflexivity.
      * simpl. unfold ssr_have. unfold solveTransfCoMod_sub_PolyTransfCoMod.
        unfold ssr_have. erewrite (IHlen _ _ _ _ gg'1).
        erewrite (IHlen _ _ _ _ gg'2). reflexivity.
      * simpl. unfold ssr_have.
        rewrite [in LHS]/solveTransfCoMod_sub_TransfCoMod_PolyMorCoMod_Pre.
        unfold ssr_have. erewrite IHlen. reflexivity.
      * simpl. unfold ssr_have.
        rewrite [in LHS]/solveTransfCoMod_sub_TransfCoMod_PolyMorCoMod_Post.
        unfold ssr_have. erewrite IHlen. reflexivity.
Qed.

Definition solveTransfCoMod0 :
  forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 ) (gg' : 'transfCoMod(0 g ~> g' )0 ),
    { g_g'_ : { g_ : 'morCoMod(0 F ~> G )0 %sol &
                      { g'_ : 'morCoMod(0 F ~> G )0 %sol &
                              'transfCoMod(0 g_ ~> g'_ )0 %sol } }
    | ( ( solveMorCoMod0 g = projT1 g_g'_ ) *
        ( solveMorCoMod0 g' = projT1 (projT2 g_g'_) ) )%type }.
Proof.
  intros; apply: (@solveTransfCoMod (gradeTransf gg') F G g g' gg'); constructor.
Defined.

Lemma solveTransfCoMod0_len :
  forall len (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 )
    (gg' : 'transfCoMod(0 g ~> g' )0 )
    (gradeTransf_gg'_len : (gradeTransf gg' <= len)%coq_nat),
    @solveTransfCoMod0  _ _ _ _ gg'
    = @solveTransfCoMod len _ _ _ _ _ gradeTransf_gg'_len.
Proof. intros. erewrite solveTransfCoMod_len. reflexivity. Qed.

Lemma solveTransfCoMod0_Project1_Transf :
  forall (F1 F2 Z1 : obCoMod) (z1 z1' : 'morCoMod(0 F1 ~> Z1 )0)
    (z1z1' : 'transfCoMod(0 z1 ~> z1' )0),
    (proj1_sig (solveTransfCoMod0 ( ~_1 @ F2 _o>CoMod^ z1z1' )%poly))
= (existT _ (~_1  o>CoMod (projT1 (proj1_sig (solveTransfCoMod0 z1z1'))))%sol
(existT _ (~_1  o>CoMod (projT1 (projT2 (proj1_sig (solveTransfCoMod0 z1z1')))))%sol
    ( ~_1 _o>CoMod^ (projT2 (projT2 (proj1_sig (solveTransfCoMod0 z1z1')))))%sol)).
Proof.
  intros. rewrite [solveTransfCoMod0 in LHS]lock.
  erewrite solveTransfCoMod0_len.
  rewrite -lock /solveTransfCoMod0. simpl. reflexivity.
Qed.

Lemma solveTransfCoMod0_PolyTransfCoMod :
  forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0) (g'g : 'transfCoMod(0 g' ~> g )0)
    (g'0 g'' : 'morCoMod(0 F ~> G )0) (g''g' : 'transfCoMod(0 g'' ~> g'0 )0)
    (eqMor : g'0 <~>1 g') eqMor',
    proj1_sig (solveTransfCoMod0 (g''g' o^CoMod g'g # eqMor)%poly)
    = ((projT2 (projT2 (proj1_sig (solveTransfCoMod0 g''g'))))
      o^CoMod (projT2 (projT2 (proj1_sig (solveTransfCoMod0 g'g)))) # eqMor')%sol.
Proof.
  intros. rewrite [solveTransfCoMod0 in LHS]lock. move: eqMor'.
  do 2  erewrite solveTransfCoMod0_len.
  intros. erewrite solveTransfCoMod_PolyTransfCoMod0_len.
  rewrite -lock /solveTransfCoMod0. simpl. reflexivity.
Qed.

(**ETC : ... *)
```

```
Fixpoint solveTransfCoModP len {struct len} :
forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 ) (gg' : 'transfCoMod(0 g ~> g' )0 ),
forall gradeTransf_gg' : (gradeTransf gg' <= len)%coq_nat,
  (Sol.toPolyTransf (projT2 (projT2 (proj1_sig
               (@solveTransfCoMod len _ _ _ _ gg' gradeTransf_gg'))))) <~~2 gg'.
Proof.
  case : len => [ | len ].

  (* len is O *)
  - ( move => F G g g' gg' gradeTransf_gg' );
      exfalso; abstract tac_degradeTransf gradeTransf_gg'.

  (* len is (S len) *)
  - move => F G g g' gg'; case : F G g g' / gg' =>
    [ F G g g' g'g g'0 g'' g''g' eqMor (* g''g' o^CoMod g'g *)
    | F G g g' g'g E f (* f _o>CoMod^ g'g *)
    | G H h F g g' g'g (* g'g ^o>CoMod_ h *)
    | F G g (* @'UnitTransfCoMod g *)
    | F1 F2 Z1 z1 z1' z1z1' (* ~_1 @ F2 _o>CoMod^ z1z1' *)
    | F1 F2 Z2 z2 z2' z2z2' (* ~_2 @ F1 _o>CoMod^ z2z2' *)
    | L F1 F2 f1 f1' f1f1' f2 f2' f2f2' (* << f1f1' ,^CoMod f2f2' >> *)
    ] gradeTransf_gg' .

    (* gg' is g''g' o^CoMod g'g *)
    + all: cycle 1.
    (* gg' is f _o>CoMod^ g'g *)
    + all: cycle 1.
    (* gg' is g'g ^o>CoMod_ h *)
    + all: cycle 1.

    (* gg' is @'UnitTransfCoMod g *)
    + have  gSol_prop := (@solveMorCoMod0P _ _ g).
      set gSol :=  (@solveMorCoMod0 _ _ g) in gSol_prop.
      clear -gSol_prop; abstract tac_reduce_solveMorCoMod0.

    (* gg' is ~_1 @ F2 _o>CoMod^ z1z1' *)
    + simpl; set same_blurb := (_ gradeTransf_gg' : ( _ <= len )%coq_nat) .
      move: (solveTransfCoModP len _ _ _ _ z1z1' same_blurb).
      clear; abstract tac_reduce_solveMorCoMod0.

    (* gg' is ~_2 @ F1 _o>CoMod^ z2z2' *)
    + simpl; set same_blurb := (_ gradeTransf_gg' : ( _ <= len )%coq_nat) .
      move: (solveTransfCoModP len _ _ _ _ z2z2' same_blurb).
      clear; abstract tac_reduce_solveMorCoMod0.

    (* gg' is << f1f1' ,^CoMod f2f2' >> *)
    + simpl; set same_blurb1 := (_ gradeTransf_gg' : ( _ <= len )%coq_nat) .
      move: (solveTransfCoModP len _ _ _ _ f1f1' same_blurb1).
      set same_blurb2 := ( ( ( _ gradeTransf_gg'  _ _ ) ) : ( _ <= len)%coq_nat) .
      move: (solveTransfCoModP len _ _ _ _ f2f2' same_blurb2).
      clear; abstract tac_reduce_solveMorCoMod0.

    (* gg' is g''g' o^CoMod g'g *)
    + simpl; set same_blurb' := (_ gradeTransf_gg' : ( _ <= len )%coq_nat) .
      move: (solveTransfCoModP len _ _ _ _ g'g same_blurb').
      set same_blurb_ := (_ gradeTransf_gg' : ( _ <= len )%coq_nat) .
      move: (solveTransfCoModP len _ _ _ _ g''g' same_blurb_).
      set same_eqMor' := ( _ eqMor _ _ _ _ : ( _ <~>1 _ )%poly) .
      set same_blurb_refl := (_ same_eqMor' : ( _ <= _ )%coq_nat) .
      move: (snd (@solveTransfCoMod_PolyTransfCoModP _ _ _ _ _ _ _ _ _
                  same_eqMor' same_blurb_refl same_eqMor' (*memo: same*) )).
      clear; abstract tac_reduce_solveMorCoMod0.

    (* gg' is f _o>CoMod^ g'g , to  (fSol _o>CoMod^ g'Sol_gSol) *)
    + simpl; set same_blurb := (_ gradeTransf_gg' : ( _ <= len )%coq_nat) .
      move: (solveTransfCoModP len _ _ _ _ g'g same_blurb).
```

```
      have fSol_prop := (@solveMorCoMod0P _ _ f).
      set fSol := (@solveMorCoMod0 _ _ f) in fSol_prop.
      move: fSol_prop.
      set same_blurb_refl := (_ f _ : ( _ <= _ )%coq_nat) .
      move: (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_PreP _ _ _ _ _ _ _ _
                                                              same_blurb_refl).
      clear; abstract tac_reduce_solveMorCoMod0.

    (* gg' is g'g ^o>CoMod_ h *)
    + simpl; set same_blurb := (_ gradeTransf_gg' : ( _ <= len )%coq_nat) .
      move: (solveTransfCoModP len _ _ _ _ g'g same_blurb).
      have hSol_prop := (@solveMorCoMod0P _ _ h).
      set hSol := (@solveMorCoMod0 _ _ h) in hSol_prop.
      move: hSol_prop.
      set same_blurb_refl := (_ h _ _ _ _ : ( _ <= _ )%coq_nat) .
      move: (@solveTransfCoMod_TransfCoMod_PolyMorCoMod_PostP _ _ _ _ _ _ _ _
                                                               same_blurb_refl).
      clear; abstract tac_reduce_solveMorCoMod0.
Qed.

Lemma solveTransfCoMod0P :
forall (F G : obCoMod) (g g' : 'morCoMod(0 F ~> G )0 ) (gg' : 'transfCoMod(0 g ~> g' )0 ),
  (Sol.toPolyTransf (projT2 (projT2 (proj1_sig
                                (@solveTransfCoMod0 _ _ _ _ gg'))))) <~~2 gg' .
Proof. intros. apply: solveTransfCoModP . Qed.

End Resolve.
End Transf.
End TWOFOLD.
```

Voila.