title / ereview (Grammatical sheaf cohomology, its MODOS proof-assistant and WorkSchool 365 market for learning reviewers) title / ereview

short / ereview (The "double plus" definition of sheafification says that not-only the outer families-of-families are modulo the germ-equality, but-also the inner families are modulo the germ-equality. This outer-inner contrast is the hint that the "double plus" should be some inductive construction... that grammatical sheaf cohomology exists! And the MODOS proof-assistant is its cut-elimination confluence. The key technique is that the grammatical sieves (nerve) could be programmed such to inductively store both the (possibly incompatible) glued-data along with its differentials (incompatibilities) of the gluing. The significance of studying "family of families" grammatically instead of the semantic geometry-gluing is similar as the earlier significance of studying "equality of equalities" grammatically instead of the semantic homotopy-paths. And such research programme, prompted from the mathematicians Kosta Dosen and Pierre Cartier, would require some new WorkSchool365.com education market for paid tested learning peer reviewers.) short / ereview

reviewers / ereview (.) reviewers / ereview

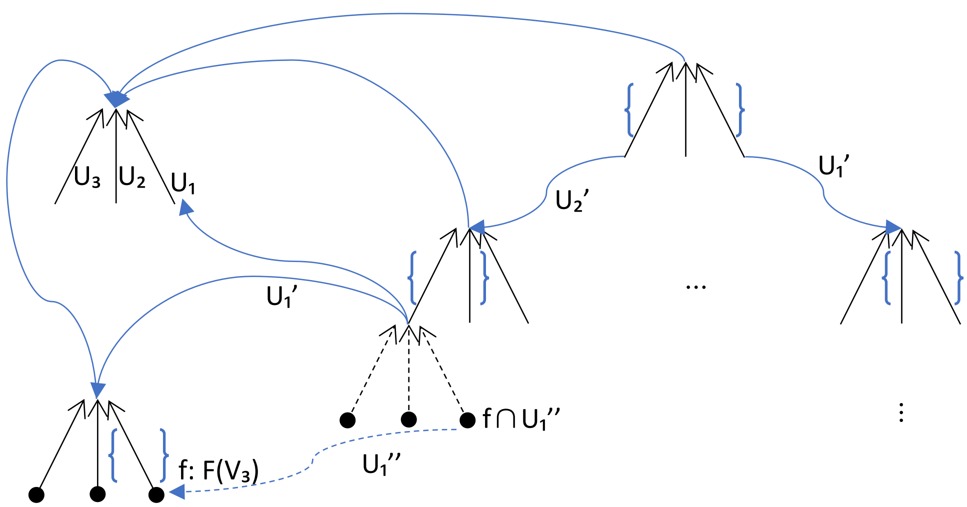

Diagram 1. Each nested basic sieve is some refinement of the fixed cover $\mathcal{U} = \{U_i \to U\}_{i \in I}$. The grammatical total/sum sieve is no longer one-to-one (mono) into the actual arrows of the site.

Lemma 03AS (https://stacks.math.columbia.edu/tag/03AS). Let $\mathcal{C}$ be a category. Let $\mathcal{U} = \{U_i \to U\}_{i \in I}$ be a family of morphisms with fixed target such that all fibre products $U_{i_0} \times_U \dots \times_U U_{i_p}$ exist in $\mathcal{C}$. Consider the chain complex $Z_{U,\cdot}$ of abelian presheaves

$$\dots \to \bigoplus_{i_0 i_1 i_2} Z_{U_{i_0} \times_U U_{i_1} \times_U U_{i_2}} \to \bigoplus_{i_0 i_1} Z_{U_{i_0} \times_U U_{i_1}} \to \bigoplus_{i_0} Z_{U_{i_0}} \to 0 \to$$

where the last nonzero term is placed in degree 0 and where the map

$$Z_{U_{i_0}\times_U \dots \times_U U_{i_{p+1}}} \longrightarrow Z_{U_{i_0}\times_U \dots \widehat{U_{i_j}} \dots \times_U U_{i_{p+1}}}$$

is given by $(-1)^j$ times the canonical map. Then there is an isomorphism

$$\mathrm{Hom}_{PAb(\mathcal{C})}\left(Z_{U,\cdot}, \mathcal{F}\right) = \check{\mathcal{C}}^{\cdot}(\mathcal{U}, \mathcal{F})$$

functorial in $\mathcal{F} \in \mathrm{Ob}\big(PAb(\mathcal{C})\big)$ ■

Note that any of the products $U_{i_0} \times_U \dots \times_U U_{i_p}$ may be empty. So how is the usual nerve modelled? Via the contravariant structure sheaf of the compactly-supported continuous functions, which is in fact also some covariant co-sheaf. Therefore, instead of

Lemma 03F5. Let $\mathcal{O}$ be a presheaf of rings on $\mathcal{C}$. The chain complex

$$Z_{U,\cdot} \otimes_{p,Z} \mathcal{O}$$

is exact in positive degrees ■

Oneself could dualize any co-sheaf $\mathcal{O}$ through the complex of the elementary projective sheaves (instead of the generators)

$$\mathrm{projSh}_{U_I}(M)(U_J) = \begin{cases} M, & U_J \subseteq U_I \\ 0, & \text{else} \end{cases}$$

with the boundary maps

$$\mathrm{projSh}_{U_{i_0}\times_U \dots \times_U U_{i_{p+1}}}\left(\mathcal{O}\left(U_{i_0} \times_U \dots \times_U U_{i_{p+1}}\right)\right)$$

$$\underset{\text{extension}_{\mathcal{O}}}{\longrightarrow} \mathrm{projSh}_{U_{i_0}\times_U \dots \widehat{U_{i_j}} \dots \times_U U_{i_{p+1}}}\left(\mathcal{O}\left(U_{i_0} \times_U \dots \widehat{U_{i_j}} \dots \times_U U_{i_{p+1}}\right)\right)$$

Note that this resulting complex would be the same as the linear dual of the $\mathrm{Hom}(-,\omega)$ through the dualizing complex of co-sheaves (Verdier dual)… The observation is that this duality is inevitable, so that homology of one (structure) co-sheaf and cohomology of another (coefficients) sheaf would be constructed simultaneously. Now how does the computational construction relate to the logical definition? This is Lemma 03AU and Lemma 03F7.

Lemma 03AU. For abelian presheaves only, not sheaves, there is a functorial quasi-isomorphism

$$\check{\mathcal{C}}^{\cdot}(\mathcal{U}, \mathcal{F}) \longrightarrow R\widetilde{H^0}(\mathcal{U}, \mathcal{F})$$

where the right-hand side indicates the derived functor

$$R\widetilde{H^0}(\mathcal{U}, -) \colon D^+\big(PAb(\mathcal{C})\big) \longrightarrow D^+(Z)$$

of the left exact functor $\widetilde{H^0}(\mathcal{U}, -) \colon PAb(\mathcal{C}) \longrightarrow Ab$ ■

Lemma 03F7. Let any abelian sheaf $\mathcal{F} \in \mathrm{Ob}\big(Ab(\mathcal{C})\big)$. Assume that $H^i\left(U_{i_0} \times_U \dots \times_U U_{i_p}, \mathcal{F}\right) = 0$ for all $i > 0$, all $p \geq 0$ and all $i_0, \dots, i_p \in I$. Then $\widetilde{H^p}(\mathcal{U}, \mathcal{F}) = H^p(U, \mathcal{F})$ ■

And both lemmas rely on the technique of moving into the total complex of some double complex such as $\check{\mathcal{C}}^{\cdot}(\mathcal{U}, \mathcal{I}^{\cdot})$ or the Cartan-Eilenberg resolution (Lemma 015I) for some injective resolution $\mathcal{F} \to \mathcal{I}^{\cdot}$. Anyway, to sense the idea, remember that for any acyclic resolution $\mathcal{F} \to \mathcal{S}^{\cdot}$, the long exact sequence allows to move through the double complex such as:

$$H^3\big(\Gamma(X, \mathcal{S}^{\cdot})\big) \coloneqq \frac{H^0(X, \ker d^3)}{H^0(X, \operatorname{im} d^2)} \xrightarrow{\sim} H^1(X, \ker d^2) \xrightarrow{\sim} H^2(X, \ker d^1)$$

$$\xrightarrow{\sim} H^3(X, \ker d^0) =: H^3(X, \mathcal{F})$$

In other words, the cellular degree can be inductively decreased at the cost of increasing the coefficients degree. And for the injective resolution of some presheaf, this increase in the coefficients degree signifies that the coefficients are more complicated such as “family of families of base values” (or “superposition of superpositions”, in the case of the co-presheaf). This suggests to construct some single storage container both for the gluing and for the differentials (incompatibilities) of this gluing, as sketched in the *Diagram 1*. Memo that the grammatical total/sum sieve is no longer one-to-one (mono) into the actual arrows of the site; any actual arrow may be factorized via many (the cell degree) codes.

For the benefit of the lazy reader, *Diagram 2* is some instance of *Diagram 1* where all the refinements from the fixed top cover-sieve are identities. And the Coq *Code C1* is the nerve-sieve inductive type for such limited instances.

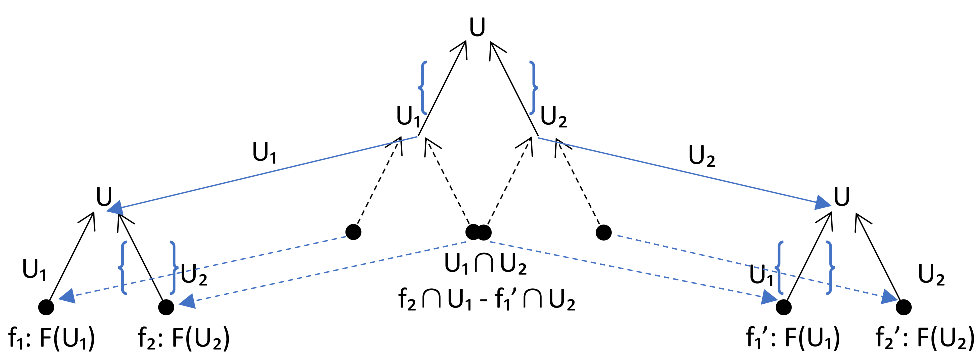

Diagram 2. Instance of *Diagram 1* where all the refinements from the fixed top cover-sieve are identities.

C1_format / coq

```
Variable nerveStruct : seq nat -> bool.
Variable topSieve : nat.

Inductive nerveSieve: forall (dimCoef: nat) (dimCell: nat) (cell: seq
nat), Type :=

| Diff_nerveSieve: forall (dimCoef: nat) (dimCell: nat) (cell: seq nat)
 (outer_: Fin.t (S dimCell) -> Fin.t (S topSieve))
 (innerCell_: forall (outerIndex: Fin.t (S dimCell)), seq nat)
 (cell_eq: forall (outerIndex: Fin.t (S dimCell)),
```

```
   seq.perm_eq cell ((to_nat (outer_ outerIndex): nat) :: (innerCell_
outerIndex)))
 (cell_nerveStruct: nerveStruct cell),
 forall (inner_nerveSieve: forall (outerIndex: Fin.t (S dimCell)),
   nerveSieve dimCoef dimCell (innerCell_ outerIndex)),
 nerveSieve (S dimCoef) (S dimCell) cell

| Glue_nerveSieve: forall (dimCoef: nat) (dimCell: nat) (cell: seq
nat),
  forall (inner_nerveSieve: forall (outerIndex: Fin.t (S topSieve)),
    nerveSieve dimCoef dimCell cell),
  nerveSieve (S dimCoef) dimCell cell

| Unit_nerveSieve:
  nerveSieve 0 0 [:: ].
```

C1_format / coq

Together with its "differential gluing" elimination-scheme into any (sheafified) sheaf:

C2_format / coq

```
Definition diffGluing: forall (dimCoef: nat),
forall (inner_coef: forall (outerIndex: Fin.t (S topSieve)),
                    forall (dimCell: nat) (cell: seq nat),
        nerveSieve dimCoef dimCell cell  -> sheafiCoef cell),
forall (dimCell: nat) (cell: seq nat),
nerveSieve (S dimCoef) dimCell cell -> sheafiCoef cell.
```

C2_format / coq

And concretely the codes below show one computed example of such sieve values in this nerve type, together with one computed example of such coefficient values in this sheafified type.

C3_format / coq

```
Glue_nerveSieve (two_cases
(Diff id (two_cases
     (Diff (one_cases (Fin.FS Fin.F1))
          (one_cases (Unit_nerveSieve xpredT 1)))
     (Diff (one_cases Fin.F1) (one_cases (Unit_nerveSieve xpredT 1)))))
(Diff (two_cases (Fin.FS Fin.F1) Fin.F1) (two_cases
     (Diff (one_cases Fin.F1) (one_cases (Unit_nerveSieve xpredT 1)))
     (Diff (one_cases (Fin.FS Fin.F1))
          (one_cases (Unit_nerveSieve xpredT 1))))))
  : nerveSieve xpredT 1 3 2 [:: 0; 1]
```

C3_format / coq

C4_format / coq

```
glue_shfyCoef
[< restrict_shfyCoef [:: 1; 0; 1]
     (diff_shfyCoef
        [< restrict_shfyCoef [:: 0; 1] (congr_shfyCoef dd) ;;
           restrict_shfyCoef [:: 0; 1] (congr_shfyCoef bb) >]) ;;
   restrict_shfyCoef [:: 0; 0; 1]
```

```
      (diff_shfyCoef
         [< restrict_shfyCoef [:: 0; 1] (congr_shfyCoef aa) ;;
            restrict_shfyCoef [:: 0; 1] (congr_shfyCoef cc) >]) >]
: sheafiCoef [:: 0; 1]
```
C4_format / coq

Now returning to the general situation of *Diagram 1*, the pseudo-*Code C4* shows the outline of the nerve-sieve inductive type:

C4_format / coq

```
Inductive nerveSieve: forall K (UU : K → Type sieve at U)
(_ : UU refines topCover along arrow u : U → topCoverUnion), forall
(G : open where data will be stored) (_ : G ⊆ U), forall (dim: nat)
(diffCell: forall i : {0, 1, ⋯, dim-1}, topCoverOpens), Type :=

| NerveSieve_Diff  (* at cell dim +1, at coeffiecients degree +1 *) :
forall K UU G dim diffCell,

forall (famSieve_ : forall Uk : UU, sieve at some open famVertex_Uk
along some pull arrow famPullArrow_Uk : Uk → famVertex_Uk and refining
the topCover),

forall (outerFactor_ : forall i : {0, 1, ⋯, dim+1}, is some open Uk :
UU with G ⊆ outerFactor_(i) ⊆ U),

forall (inner_nerveSieve : forall i : {0, 1, ⋯, dim+1},
  nerveSieve (famSieve_(outerFactor_ i))
             (the generator open V_i of famSieve_(outerFactor_ i) where
G factorizes)
             (fun j : {0, 1, ⋯, dim} => outerFactor_(if j<i then j else
j+1) as topCoverOpens)),

forall (G_weight : structCoSheaf G),

nerveSieve (SumSieve famSieve_ over UU) G
  (fun i : {0, 1, ⋯, dim+1} => topCoverOpen generator of outerFactor_i)

| NerveSieve_Gluing (* at same cell dim >= 0, at coefficients degree +1
*) : ⋯

| NerveSieve_Base (* at cell dim = 0, at coeffiecients degree = 0
*) : ⋯
```
C4_format / coq

Finally, below are the draft forms of the remaining *Code C5* for the particular *Diagram 2* and *Code C6* for the general *Diagram 1*:

https://github.com/1337777/cartier

and such research programme would require some new WorkSchool 365 education market for paid tested learning peer reviewers (sign-in via the Microsoft Marketplace):

https://workschool365.com

https://appsource.microsoft.com/en-us/product/office/WA200003598

Learning Reviewers Qualification Quiz: Q1. The MODOS end-goal is:

(A) proof-assistant for the computational logic of “family of families”.

(B) formalization of the correctness of the book “Categories for the Working Mathematician”.

(C) writing vertical pretty formulas in latex.

Q1 ; 30 / quiz Click or tap here to enter text. Q1 ; 30 / quiz

In Word, “Insert; Add-ins; WorkSchool 365” to play this Coq script or sign-in for learning reviewers. WorkSchool365.com

C5 / coq

```
Module Example.

From Coq Require Lia Vectors.Vector.
From mathcomp Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq fintype tuple
finfun.
Set Implicit Arguments. Unset Strict Implicit. Unset Printing Implicit Defensive.

Section tools.
Definition to_nat := fun m => fun i : Fin.t (S m) => (proj1_sig  (Fin.to_nat i)).

Definition one_cases : forall (P : Fin.t (S 0) -> Type)
 (P1 : P Fin.F1)  (p : Fin.t (S 0)), P p.
Proof. intros. apply: Fin.caseS'. apply P1.
clear p; intros p. pattern p. apply: Fin.case0.
Defined.

Definition two_cases : forall (P : Fin.t (S 1) -> Type)
 (P1 : P Fin.F1) (P2 : P (Fin.FS Fin.F1)) (p : Fin.t (S 1)), P p.
Proof. intros. apply: Fin.caseS'. apply P1.
apply: one_cases. apply P2.
Defined.

Variable dimCell : nat.
Variable typeAt_ : Fin.t (S dimCell) -> Type.

Inductive ilist : nat -> Type :=
  | Nil_ilist : ilist O
  | Cons_ilist : forall n (H : (n < (S dimCell))%coq_nat),
      typeAt_ ( Fin.of_nat_lt H) -> ilist n -> ilist (S n).

Definition to_ilist (t: forall i : Fin.t (S dimCell), typeAt_ i) n (H : (n <= S
dimCell)%coq_nat) : ilist n.
Proof. induction n.
- apply: Nil_ilist.
- apply: Cons_ilist.
  + exact: (t (Fin.of_nat_lt H)).
  + apply IHn. abstract(Lia.lia).
Defined.
End tools.

Section nerveSieve.
```

```
Variable nerveStruct : seq nat -> bool.
Variable topSieve : nat.

Inductive nerveSieve: forall (dimCoef: nat) (dimCell: nat) (cell: seq nat), Type :=

| Diff_nerveSieve: forall (dimCoef: nat) (dimCell: nat) (cell: seq nat)
 (outer_: Fin.t (S dimCell) -> Fin.t (S topSieve))
 (innerCell_: forall (outerIndex: Fin.t (S dimCell)), seq nat)
 (cell_eq: forall (outerIndex: Fin.t (S dimCell)),
   seq.perm_eq cell ((to_nat (outer_ outerIndex): nat) :: (innerCell_ outerIndex)))
 (cell_nerveStruct: nerveStruct cell),
 forall (inner_nerveSieve: forall (outerIndex: Fin.t (S dimCell)),
   nerveSieve dimCoef dimCell (innerCell_ outerIndex)),
 nerveSieve (S dimCoef) (S dimCell) cell

| Glue_nerveSieve: forall (dimCoef: nat) (dimCell: nat) (cell: seq nat),
  forall (inner_nerveSieve: forall (outerIndex: Fin.t (S topSieve)),
    nerveSieve dimCoef dimCell cell),
  nerveSieve (S dimCoef) dimCell cell

| Unit_nerveSieve:
  nerveSieve 0 0 [:: ].

Parameter sheafiCoef : forall (cell: seq nat), Type.
Parameter restrict_shfyCoef : forall (cell0: seq nat), forall (cell: seq nat) (i :
nat),
 perm_eq cell (i :: cell0) -> sheafiCoef cell0 -> sheafiCoef cell.
Parameter diff_shfyCoef : forall (dimCell: nat) (cell: seq nat),
  ilist (fun (outerIndex: Fin.t (S dimCell)) => sheafiCoef cell) (S dimCell) ->
sheafiCoef cell.
Parameter glue_shfyCoef : forall (cell: seq nat),
 ilist (fun (outerIndex: Fin.t (S topSieve)) => sheafiCoef ((to_nat outerIndex) ::
cell)) (S topSieve)  -> sheafiCoef cell.
Parameter congr_shfyCoef : forall (cell0: seq nat), forall (cell: seq nat),
 perm_eq cell cell0 -> sheafiCoef cell0 -> sheafiCoef cell.

Definition diffGluing: forall (dimCoef: nat),
forall (inner_coef: forall (outerIndex: Fin.t (S topSieve)), forall (dimCell: nat)
(cell: seq nat),
  nerveSieve dimCoef dimCell cell  -> sheafiCoef cell),
forall (dimCell: nat) (cell: seq nat),
nerveSieve (S dimCoef) dimCell cell -> sheafiCoef cell.
Proof. intros ? ?  ? ? ns. inversion ns; subst.
{ (* Diff_nerveSieve *)
  (* apply: (diff_shfyCoef (fun i : 'I_(_.+1) =>
              restrict_shfyCoef (cell_eq i)
                (inner_coef (outer_ i) _  (innerCell_ i) (inner_nerveSieve i)))). *)
  eapply diff_shfyCoef. refine (to_ilist _ (le_n _) ). intros i. apply:
restrict_shfyCoef.
  + apply: (cell_eq i).
  + eapply (inner_coef (outer_ i)). apply: inner_nerveSieve. }
{ (* Glue_nerveSieve *)
  (* apply: glue_shfyCoef (fun i : 'I_topSieve =>
              inner_coef i dimCell cell (inner_nerveSieve i)). *)
  apply glue_shfyCoef. refine (to_ilist _ (le_n _) ). intros i. eapply
restrict_shfyCoef.
  + exact: perm_refl.
  + eapply (inner_coef i). apply: (inner_nerveSieve i). }
Defined.

Definition diffGluing_unit:
```

```
forall (unit_coef: sheafiCoef [:: ]),
forall (dimCell: nat) (cell: seq nat),
nerveSieve 0 dimCell cell -> sheafiCoef cell.
Proof. intros ? ? ? ns. inversion ns; subst.
{ (* Unit_nerveSieve *) exact: unit_coef. }
Defined.
End nerveSieve.

Section example1_sheafiCoef.

Definition example1_nerveSieve: nerveSieve (fun _ => true) 1  2 1 [:: 0].
Proof. apply: Glue_nerveSieve. apply: two_cases.
{ unshelve eapply Diff_nerveSieve.
  apply: one_cases. exact: (@Fin.F1 1).
  apply: one_cases. exact: [:: ].
  apply: one_cases. reflexivity.
  reflexivity.
  apply: one_cases. apply Unit_nerveSieve.  }
{ unshelve eapply Diff_nerveSieve.
  apply: one_cases. exact: (@Fin.F1 1).
  apply: one_cases. exact: [:: ].
  apply: one_cases. reflexivity.
  reflexivity.
  apply: one_cases. apply Unit_nerveSieve.  }
Defined.

Notation Diff y x := (@Diff_nerveSieve _ _ _ _ _ y _ _ _ x).
Print example1_nerveSieve.
(* Glue_nerveSieve
  (two_cases (Diff (one_cases Fin.F1) (one_cases (Unit_nerveSieve xpredT 1)))
        (Diff (one_cases Fin.F1) (one_cases (Unit_nerveSieve xpredT 1))))
     : nerveSieve xpredT 1 2 1 [:: 0] *)

Variable aa bb cc dd : sheafiCoef [:: ].

Definition example1_sheafiCoef:  sheafiCoef [:: 0].
Proof. apply: (diffGluing _ example1_nerveSieve). apply: two_cases.
{  apply: diffGluing.
  { apply: two_cases.
    exact: (diffGluing_unit aa). exact: (diffGluing_unit bb). } }
{ apply: diffGluing.
  { apply: two_cases.
    exact: (diffGluing_unit cc). exact: (diffGluing_unit dd). } }
Defined.

Notation "[< x2 ;; .. ;; xn >]" := (Cons_ilist x2 .. (Cons_ilist xn (@Nil_ilist _
_)) ..)
  (at level 0, format "[< '['  x2  ;;  '/'  ..  ;;  '/'  xn  ']' >]" ) .
Arguments Nil_ilist {_ _}.
Arguments restrict_shfyCoef [_] cell [_ _] _.
Arguments congr_shfyCoef {_ _ _} _.

Eval compute in example1_sheafiCoef.
(* = glue_shfyCoef
[< restrict_shfyCoef [:: 1; 0]
     (diff_shfyCoef [< restrict_shfyCoef [:: 0] cc >]) ;;
   restrict_shfyCoef [:: 0; 0]
     (diff_shfyCoef [< restrict_shfyCoef [:: 0] aa >]) >]
: sheafiCoef [:: 0] *)
End example1_sheafiCoef.
```

```
Section example2_sheafiCoef.

Example example2_nerveSieve: nerveSieve (fun _ => true) 1  3 2 [:: 0; 1].
Proof. apply: Glue_nerveSieve. apply: two_cases.
{ unshelve eapply Diff_nerveSieve.
  exact: id.
  { apply: two_cases. exact: [:: 1]. exact: [:: 0]. }
  { apply: two_cases. reflexivity. reflexivity. }
  reflexivity.
  { apply: two_cases.
    { unshelve eapply Diff_nerveSieve.
      apply: one_cases. exact: (Fin.FS (@Fin.F1 0)).
      apply: one_cases. exact: [::].
      apply: one_cases. reflexivity.
      reflexivity.
      apply: one_cases. apply: Unit_nerveSieve. }
    { unshelve eapply Diff_nerveSieve.
      apply: one_cases.  exact: (@Fin.F1 1).
      apply: one_cases.  exact: [::].
      apply: one_cases. reflexivity.
      reflexivity.
      apply: one_cases. apply: Unit_nerveSieve. } } }
{ unshelve eapply Diff_nerveSieve.
  { (* permute, not id inclusion *)
    apply: two_cases. exact: (Fin.FS (@Fin.F1 0)). exact: (@Fin.F1 1). }
  { apply: two_cases. exact: [:: 0]. exact: [:: 1]. }
  { apply: two_cases. reflexivity. reflexivity. }
  reflexivity.
  { apply: two_cases.
    { unshelve eapply Diff_nerveSieve.
      apply: one_cases. exact: (@Fin.F1 1).
      apply: one_cases. exact: [::].
      apply: one_cases. reflexivity.
      reflexivity.
      apply: one_cases. apply: Unit_nerveSieve. }
    { unshelve eapply Diff_nerveSieve.
      apply: one_cases. exact: (Fin.FS (@Fin.F1 0)).
      apply: one_cases. exact: [::].
      apply: one_cases. reflexivity.
      reflexivity.
      apply: one_cases. apply: Unit_nerveSieve. } } }
Defined.

Notation Diff y x := (@Diff_nerveSieve _ _ _ _ _ y _ _ _ x).
Print example2_nerveSieve.
(*Glue_nerveSieve (two_cases
(Diff id (two_cases
      (Diff (one_cases (Fin.FS Fin.F1))
          (one_cases (Unit_nerveSieve xpredT 1)))
      (Diff (one_cases Fin.F1) (one_cases (Unit_nerveSieve xpredT 1)))))
(Diff (two_cases (Fin.FS Fin.F1) Fin.F1) (two_cases
      (Diff (one_cases Fin.F1) (one_cases (Unit_nerveSieve xpredT 1)))
      (Diff (one_cases (Fin.FS Fin.F1))
          (one_cases (Unit_nerveSieve xpredT 1))))))
  : nerveSieve xpredT 1 3 2 [:: 0; 1] *)

Variable aa bb  : sheafiCoef [:: 0].
Variable  cc dd : sheafiCoef [:: 1].

Definition example2_sheafiCoef: sheafiCoef [:: 0; 1].
Proof. apply: (diffGluing _ example2_nerveSieve). apply: two_cases.
```

```
{ apply: diffGluing. intros _.
  { intros ? ? ns. inversion ns; subst. (* TODO: for case-analysis, everywhere should
use destructible ilist instead of function,
      together with conversion operation ilist -> function, similar as ssreflect finfun
*)
    { move:   (outer_ (@Fin.F1 dimCell0)) (innerCell_ (@Fin.F1 dimCell0))(cell_eq
(@Fin.F1 dimCell0)) (inner_nerveSieve (@Fin.F1 dimCell0)).
      apply: two_cases.
      { move => innerCell_0 cell_eq0 inner_nerveSieve0. inversion inner_nerveSieve0;
subst. apply: (congr_shfyCoef cell_eq0).
        exact: aa. }
      { move => innerCell_0 cell_eq0 inner_nerveSieve0. inversion inner_nerveSieve0;
subst. apply: (congr_shfyCoef cell_eq0).
        exact: cc. }  }
    { move:  (inner_nerveSieve (@Fin.F1 1)). move =>  inner_nerveSieve0. inversion
inner_nerveSieve0; subst.
      apply: glue_shfyCoef. refine (to_ilist _ (le_n 2) ).  apply: two_cases. exact:
aa. exact: cc. } } }
{ apply: diffGluing. intros _.
  { intros ? ? ns. inversion ns; subst.
    {  move:   (outer_ (@Fin.F1 dimCell0)) (innerCell_ (@Fin.F1 dimCell0))(cell_eq
(@Fin.F1 dimCell0)) (inner_nerveSieve (@Fin.F1 dimCell0)).
      apply: two_cases.
      { move => innerCell_0 cell_eq0 inner_nerveSieve0. inversion inner_nerveSieve0;
subst. apply: (congr_shfyCoef cell_eq0).
        exact: bb. }
      { move => innerCell_0 cell_eq0 inner_nerveSieve0. inversion inner_nerveSieve0;
subst. apply: (congr_shfyCoef cell_eq0).
        exact: dd. } }
    { move:  (inner_nerveSieve (@Fin.F1 1)). move =>  inner_nerveSieve0. inversion
inner_nerveSieve0; subst.
      apply: glue_shfyCoef. refine (to_ilist _ (le_n 2) ).  apply: two_cases. exact:
bb. exact: dd. } } }
Defined.

Notation "[< x2 ;; .. ;; xn >]" := (Cons_ilist x2 .. (Cons_ilist xn (@Nil_ilist _
_)) ..)
  (at level 0, format "[< '['  x2  ;;  '/'  ..  ;;  '/'  xn  ']' >]" ) .
Arguments Nil_ilist {_ _}.
Arguments restrict_shfyCoef [_] cell [_ _] _.
Arguments congr_shfyCoef {_ _ _} _.

Eval compute in example2_sheafiCoef. (* note that because of the permute instruction in
example2_nerveSieve,
then the order dd,bb is permuted as contrasted to aa,cc *)
(* = glue_shfyCoef
[< restrict_shfyCoef [:: 1; 0; 1]
     (diff_shfyCoef
        [< restrict_shfyCoef [:: 0; 1] (congr_shfyCoef dd) ;;
           restrict_shfyCoef [:: 0; 1] (congr_shfyCoef bb) >]) ;;
   restrict_shfyCoef [:: 0; 0; 1]
     (diff_shfyCoef
        [< restrict_shfyCoef [:: 0; 1] (congr_shfyCoef aa) ;;
           restrict_shfyCoef [:: 0; 1] (congr_shfyCoef cc) >]) >]
: sheafiCoef [:: 0; 1]
 *)
End example2_sheafiCoef.

End Example.
```
C5 / coq

And for the general situation:

```
C6 / coq (** # #
#+TITLE: cartierSolution0.v

https://github.com/1337777/cartier/blob/master/cartierSolution0.v

QUICK START FROM Inductive nerveSieve

-----

#+BEGIN_SRC coq :exports both :results silent # # **)
From Coq Require Lia.
From Coq Require Import RelationClasses Setoid SetoidClass
     Classes.Morphisms_Prop RelationPairs CRelationClasses CMorphisms.
From mathcomp Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq fintype tuple
finfun.

Set Implicit Arguments. Unset Strict Implicit. Unset Printing Implicit Defensive.
Set Primitive Projections. Set Universe Polymorphism.

Close Scope bool. Declare Scope poly_scope. Delimit Scope poly_scope with poly. Open
Scope poly.

Module Type GENE.

Class relType : Type := RelType
{ _type_relType : Type;
  _rel_relType : crelation _type_relType;
  _equiv_relType :> Equivalence _rel_relType }.
About relType.
Coercion _type_relType : relType >-> Sortclass.

Definition equiv {A: Type} {R: crelation A} `{Equivalence A R} : crelation A := R.

 (* TODO: keep or comment *)
Arguments _rel_relType : simpl never.
Arguments _equiv_relType : simpl never.
Arguments equiv : simpl never.

Notation " x == y " := (@equiv (* (@_type_relType _) *) _ (@_rel_relType _)
(@_equiv_relType _) x y)
  (at level 70, no associativity) : type_scope.
Notation LHS := (_ : fun XX => XX == _).
Notation RHS := (_ : fun XX => _ == XX).
Notation "[|  x  ; .==. |]" := (exist (fun t => (_ == _)) x _) (at level 10, x at next
level) : poly_scope.
Notation "[|  x  ; .=. |]" := (exist (fun t => (_ = _)) x _) (at level 10, x at next
level) : poly_scope.

Parameter vertexGene : eqType.

Parameter arrowGene : vertexGene -> vertexGene -> relType.
Notation "''Gene' ( V ~> U )" := (@arrowGene U V)
(at level 0, format "''Gene' (  V  ~>  U  )") : poly_scope.

Parameter composGene :
forall U, forall V W, 'Gene( W ~> V ) -> 'Gene( V ~> U ) -> 'Gene( W ~> U ).
Notation "wv o:>gene vu" := (@composGene _ _ _ wv vu)
(at level 40, vu at next level) : poly_scope.
```

```
Declare Instance composGene_Proper: forall U V W, Proper (equiv ==> equiv ==> equiv)
(@composGene U V W).

Parameter identGene : forall {U : vertexGene}, 'Gene( U ~> U ).

Parameter composGene_compos :
forall (U V : vertexGene) (vu : 'Gene( V ~> U ))
        (W : vertexGene) (wv : 'Gene( W ~> V )),
forall X (xw : 'Gene( X ~> W )),
  xw o:>gene ( wv o:>gene vu ) == ( xw o:>gene wv ) o:>gene vu.
Parameter composGene_identGene :
forall (U V : vertexGene) (vu : 'Gene( V ~> U )),
  (@identGene V) o:>gene vu == vu .
Parameter identGene_composGene :
forall (U : vertexGene), forall (W : vertexGene) (wv : 'Gene( W ~> U )),
  wv o:>gene (@identGene U) == wv.

Notation typeOf_objects_functor := (vertexGene -> relType).

Class relFunctor (F : typeOf_objects_functor) (G G' : vertexGene)  : Type := RelFunctor
{ _fun_relFunctor : 'Gene( G' ~> G ) -> F G -> F G' ;
  _congr_relFunctor :> Proper (equiv ==> @equiv _ _ (@_equiv_relType ( F G ))
                         ==> @equiv _ _ (@_equiv_relType ( F G'))) _fun_relFunctor ; }.

Coercion _fun_relFunctor : relFunctor >-> Funclass.

Definition typeOf_arrows_functor (F : typeOf_objects_functor)
:= forall G G' : vertexGene, relFunctor F G G' .

Definition fun_arrows_functor_ViewOb := composGene.

Notation "wv o>gene vu" := (@fun_arrows_functor_ViewOb _ _ _ wv vu)
(at level 40, vu at next level) : poly_scope.

Definition fun_transf_ViewObMor (G H: vertexGene) (g: 'Gene( H ~> G )) (H':
vertexGene) :
'Gene(H' ~> H) -> 'Gene(H' ~> G) .
Proof. exact: ( fun h =>  h o:>gene g ). Defined.

(* TODO: REDO GENERAL fun_transf_ViewObMor_Proper *)
Global Instance fun_transf_ViewObMor_Proper G H g H' : Proper (equiv ==> equiv)
(@fun_transf_ViewObMor G H g H').
Proof.    move. intros ? ? Heq. unfold fun_transf_ViewObMor. rewrite -> Heq;
reflexivity.
Qed.

Notation "wv :>gene vu" := (@fun_transf_ViewObMor _ _ vu _ wv)
(at level 40, vu at next level) : poly_scope.

Definition typeOf_functorialCompos_functor (F : typeOf_objects_functor)
 (F_ : typeOf_arrows_functor F)  :=
  forall G G' (g : 'Gene( G' ~> G)) G'' (g' : 'Gene( G'' ~> G')) (f : F G),
    F_ _ _ g' (F_ _ _ g f) ==
    F_ _ _ ( g' o>gene g (*? or  g' :>gene g  or  g' o:>gene g ?*) ) f.

Definition typeOf_functorialIdent_functor (F : typeOf_objects_functor)
 (F_ : typeOf_arrows_functor F) :=
  forall G (f : F G), F_ _ _ (@identGene G) f == f.

Record functor := Functor
```

```
 { _objects_functor :> typeOf_objects_functor ;
   _arrows_functor :> (* :> ??? *) typeOf_arrows_functor _objects_functor;
   _functorialCompos_functor : typeOf_functorialCompos_functor _arrows_functor;
   _functorialIdent_functor : typeOf_functorialIdent_functor _arrows_functor;
 }.

Notation "g o>functor_ [ F ] f" := (@_arrows_functor F _ _ g f)
  (at level 40, f at next level) : poly_scope.
Notation "g o>functor_ f" := (@_arrows_functor _ _ _ g f)
  (at level 40, f at next level) : poly_scope.

Definition equiv_rel_functor_ViewOb (G H : vertexGene) : crelation 'Gene( H ~> G ).
Proof.    exact: equiv.
Defined.
(* (* no lack for now, unless want uniformity of the (opaque) witness... *)
Arguments equiv_rel_functor_ViewOb /.
 *)

Definition functor_ViewOb (G : vertexGene) : functor.
Proof. unshelve eexists.
- (* typeOf_objects_functor *) intros H. exact: 'Gene( H ~> G ).
- (* typeOf_arrows_functor *) intros H H'. exists (@fun_arrows_functor_ViewOb G H H').
  abstract (typeclasses eauto).
- (* typeOf_functorialCompos_functor *) abstract (move; intros; exact:
composGene_compos).
- (* typeOf_functorialIdent_functor *) abstract (move; intros; exact:
composGene_identGene).
Defined.

Definition _functorialCompos_functor' {F : functor} :
   forall G G' (g : 'Gene( G' ~> G)) G'' (g' : 'Gene( G'' ~> G')) (f : F G),
   g' o>functor_ [ F ] (g o>functor_ [ F ]  f)
   == (g' o>functor_ [ functor_ViewOb G ] g) o>functor_ [ F ] f
:= @_functorialCompos_functor F.

Class relTransf (F E : typeOf_objects_functor) (G : vertexGene) : Type := RelTransf
{ _fun_relTransf : F G -> E G ;
  _congr_relTransf :> Proper (@equiv _ _ (@_equiv_relType ( F G ))
                          ==> @equiv _  _ (@_equiv_relType ( E G))) _fun_relTransf ; }.

Coercion _fun_relTransf : relTransf >-> Funclass.

Notation typeOf_arrows_transf F E
:= (forall G : vertexGene, relTransf F E G) .

Definition typeOf_natural_transf (F E : functor)
 (ee : typeOf_arrows_transf F E) :=
  forall G G' (g : 'Gene( G' ~> G )) (f : F G),
  g o>functor_[E] (ee G f) == ee G' (g o>functor_[F] f).

Record transf (F : functor) (E : functor) := Transf
{ _arrows_transf :> typeOf_arrows_transf F E ;
  _natural_transf : typeOf_natural_transf _arrows_transf;
}.

Notation "f :>transf_ [ G ] ee" := (@_arrows_transf _ _ ee G f)
  (at level 40, ee at next level) : poly_scope.

Notation "f :>transf_ ee" := (@_arrows_transf _ _ ee _ f)
  (at level 40, ee at next level) : poly_scope.
```

```
Definition transf_ViewObMor (G : vertexGene) (H : vertexGene) (g : 'Gene( H ~> G )) :
transf (functor_ViewOb H) (functor_ViewOb G).
Proof.  unshelve eexists.
- (* _arrows_transf *)  unshelve eexists.
  + (* _fun_relTransf *) exact: (fun_transf_ViewObMor g).
  + (* _congr_relTransf  *) exact: fun_transf_ViewObMor_Proper.
- (* _natural_transf *)abstract (move; simpl; intros; exact: composGene_compos).
Defined.

Definition _functorialCompos_functor'' {F : functor} :
   forall G G' (g : 'Gene( G' ~> G)) G'' (g' : 'Gene( G'' ~> G')) (f : F G),
   g' o>functor_ [ F ] (g o>functor_ [ F ]  f)
   == (g' :>transf_ (transf_ViewObMor g)) o>functor_ [ F ] f
:= @_functorialCompos_functor F.

Record sieveFunctor (U : vertexGene) : Type :=
  { _functor_sieveFunctor :> functor ;
    _transf_sieveFunctor : transf _functor_sieveFunctor (functor_ViewOb U) ; }.

Lemma transf_sieveFunctor_Proper (U : vertexGene) (UU : sieveFunctor U) H:
Proper (equiv ==> equiv) (_transf_sieveFunctor UU H).
  apply: _congr_relTransf.
Qed.

Notation "''Sieve' ( G' ~> G | VV )" := (@_functor_sieveFunctor G VV G')
     (at level 0, format "''Sieve' (  G'  ~>  G  |  VV  )") : poly_scope.
Notation "h o>sieve_ v " := (h o>functor_[@_functor_sieveFunctor _ _] v)
          (at level 40, v at next level, format "h  o>sieve_  v") : poly_scope.
Notation "v :>sieve_" := (v :>transf_ (_transf_sieveFunctor _)) (at level 40) :
poly_scope.

Record preSieve (U : vertexGene) : Type :=
  { _functor_preSieve :>  vertexGene -> Type;
    _transf_preSieve : forall G : vertexGene, (_functor_preSieve G) -> (functor_ViewOb
U G) ; }.

Arguments _transf_preSieve {_ _ _} .

Notation "''preSieve' ( G' ~> G | VV )" := (@_functor_preSieve G VV G')
     (at level 0, format "''preSieve' (  G'  ~>  G  |  VV  )") : poly_scope.
Notation "v :>preSieve_" := (@_transf_preSieve _ _ _ v) (at level 40) : poly_scope.

Global Ltac cbn_ := cbn -[equiv _type_relType _rel_relType _equiv_relType
_objects_functor _arrows_functor functor_ViewOb
                             _arrows_transf transf_ViewObMor _functor_sieveFunctor
_functor_preSieve _transf_sieveFunctor _transf_preSieve].
Global Ltac cbn_equiv := unfold _rel_relType, equiv; cbn -[ _arrows_functor
functor_ViewOb
                             _arrows_transf transf_ViewObMor _functor_sieveFunctor
_functor_preSieve _transf_sieveFunctor _transf_preSieve].
Global Ltac cbn_view := cbn -[equiv _type_relType _rel_relType _equiv_relType
_objects_functor  _arrows_functor
                             _arrows_transf  _functor_sieveFunctor _functor_preSieve
_transf_sieveFunctor].
Global Ltac cbn_functor := cbn -[equiv _type_relType _rel_relType _equiv_relType
functor_ViewOb
                               _arrows_transf transf_ViewObMor _functor_sieveFunctor
_functor_preSieve _transf_sieveFunctor _transf_preSieve].
Global Ltac cbn_transf := cbn -[equiv _type_relType _rel_relType _equiv_relType
_arrows_functor functor_ViewOb
```

```
                              transf_ViewObMor _functor_sieveFunctor _functor_preSieve
_transf_sieveFunctor _transf_preSieve].
Global Ltac cbn_sieve := cbn -[equiv _type_relType _rel_relType _equiv_relType
functor_ViewOb
                                transf_ViewObMor ].

Tactic Notation "cbn_" "in" hyp_list(H) := cbn -[equiv _type_relType _rel_relType
_equiv_relType _objects_functor _arrows_functor functor_ViewOb
                            _arrows_transf transf_ViewObMor _functor_sieveFunctor
_functor_preSieve _transf_sieveFunctor _transf_preSieve] in H.
Tactic Notation "cbn_equiv" "in" hyp_list(H) := unfold _rel_relType, equiv in H; cbn -
[ _arrows_functor functor_ViewOb
                            _arrows_transf transf_ViewObMor _functor_sieveFunctor
_functor_preSieve _transf_sieveFunctor _transf_preSieve] in H.
Tactic Notation "cbn_view" "in"  hyp_list(H) := cbn -[equiv _type_relType _rel_relType
_equiv_relType _objects_functor  _arrows_functor
                            _arrows_transf  _functor_sieveFunctor _functor_preSieve
_transf_sieveFunctor _transf_preSieve] in H.
Tactic Notation "cbn_functor" "in" hyp_list(H) := cbn -[equiv _type_relType
_rel_relType _equiv_relType functor_ViewOb
                              _arrows_transf transf_ViewObMor _functor_sieveFunctor
_functor_preSieve _transf_sieveFunctor _transf_preSieve] in H.
Tactic Notation "cbn_transf" "in" hyp_list(H) := cbn -[equiv _type_relType _rel_relType
_equiv_relType  _arrows_functor functor_ViewOb
                              transf_ViewObMor _functor_sieveFunctor _functor_preSieve
_transf_sieveFunctor _transf_preSieve] in H.
Tactic Notation "cbn_sieve" "in" hyp_list(H) := cbn -[equiv _type_relType _rel_relType
_equiv_relType   functor_ViewOb
                                transf_ViewObMor ] in H.

End GENE.

Module Type COMOD (Gene : GENE).
Import Gene.

Ltac tac_unsimpl := repeat
lazymatch goal with
| [ |- context [@fun_transf_ViewObMor ?G ?H ?g ?H' ?h] ] =>
change (@fun_transf_ViewObMor G H g H' h) with
(h :>transf_ (transf_ViewObMor g))
| [ |- context [@fun_arrows_functor_ViewOb ?U ?V ?W ?wv ?vu] ] =>
change (@fun_arrows_functor_ViewOb U V W wv vu) with
(wv o>functor_[functor_ViewOb U] vu)

(* no lack*)
| [ |- context [@equiv_rel_functor_ViewOb ?G ?H ?x ?y] ] =>
  change (@equiv_rel_functor_ViewOb G H x y) with
(@equiv _ _ (@_equiv_relType ( (functor_ViewOb G) H )) x y)
(* | [ |- context [@equiv_rel_arrowSieve ?G ?G' ?g ?H ?x ?y] ] =>
  change (@equiv_rel_arrowSieve G G' g H x y) with
(@equiv _ (@_rel_relType ( (arrowSieve g) H )) x y)  *)
end.

Definition transf_Compos :
forall (F F'' F' : functor) (ff_ : transf F'' F') (ff' : transf F' F),
transf F'' F.
Proof.  intros. unshelve eexists.
- intros G. unshelve eexists. intros f. exact:((f :>transf_ ff_) :>transf_ ff').
  abstract(solve_proper).
(*   exists (Basics.compose (ff' G) (ff_ G) ).  abstract(typeclasses eauto). *)
- abstract (move; cbn_; intros; (* unfold Basics.compose; *)
```

```
    rewrite -> _natural_transf , _natural_transf; reflexivity).
Defined.

Definition transf_Ident :
forall (F : functor), transf F F.
Proof.  intros. unshelve eexists.
- intros G. exists id.
  abstract(simpl_relation).
- abstract (move; cbn_; intros; reflexivity).
Defined.

Definition typeOf_commute_sieveTransf
(G : vertexGene) (V1 V2 : sieveFunctor G) (vv : transf V1 V2)  : Type :=
  forall (H : vertexGene)  (v : 'Sieve( H ~> G | V1 )),
  (v :>transf_ vv) :>sieve_ ==  v :>sieve_ .

Record sieveTransf G (V1 V2 : sieveFunctor G) : Type :=
  { _transf_sieveTransf :> transf V1 V2 ;
    _commute_sieveTransf : typeOf_commute_sieveTransf _transf_sieveTransf} .

Instance fun_transf_ViewObMor_measure {G H: vertexGene} {g: 'Gene( H ~> G )} {H':
vertexGene}:
 @Measure 'Gene(H' ~> H)  'Gene(H' ~> G) (@fun_transf_ViewObMor G H g H') := { }.

Definition sieveTransf_Compos :
forall U (F F'' F' : sieveFunctor U) (ff_ : sieveTransf F'' F') (ff' : sieveTransf F'
F),
sieveTransf F'' F.
Proof.  intros. unshelve eexists.
- exact: (transf_Compos ff_ ff').
- abstract(move; intros; cbn_transf; autounfold; do 2 rewrite -> _commute_sieveTransf;
reflexivity).
Defined.

Definition sieveTransf_Ident :
forall U (F  : sieveFunctor U) , sieveTransf F F.
Proof.  intros. unshelve eexists.
- exact: (transf_Ident F).
- abstract(move; intros; reflexivity).
Defined.

Definition identSieve (G: vertexGene) : sieveFunctor G.
unshelve eexists.
exact: (functor_ViewOb G).
exact: (transf_Ident (functor_ViewOb G)).
Defined.

Definition sieveTransf_identSieve :
forall U (F  : sieveFunctor U) , sieveTransf F (identSieve U).
Proof.  intros. unshelve eexists.
- exact: (_transf_sieveFunctor F).
- abstract(move; intros; reflexivity).
Defined.
(* TODO MERE WITH sieveTransf_identSieve *)
Lemma sieveTransf_sieveFunctor G (VV : sieveFunctor G) :
sieveTransf VV (identSieve G).
Proof. unshelve eexists. exact: _transf_sieveFunctor.
- (* _commute_sieveTransf *) abstract(move; reflexivity).
Defined.

Record sieveEquiv G (V1 V2 : sieveFunctor G) : Type :=
```

```
  { _sieveTransf_sieveEquiv :> sieveTransf V1 V2 ;
  _revSieveTransf_sieveEquiv : sieveTransf V2 V1 ;
  _injProp_sieveEquiv : forall H v, (v :>transf_[H] _revSieveTransf_sieveEquiv )
                           :>transf_ _sieveTransf_sieveEquiv == v ;
_surProp_sieveEquiv : forall H v, (v :>transf_[H] _sieveTransf_sieveEquiv )
                           :>transf_ _revSieveTransf_sieveEquiv == v } .

Definition rel_sieveEquiv G : crelation (sieveFunctor G) := fun V1 V2 => sieveEquiv V1
V2.

Instance equiv_sieveEquiv G: Equivalence (@rel_sieveEquiv G ).
unshelve eexists.
- intros V1. unshelve eexists. exact (sieveTransf_Ident _). exact (sieveTransf_Ident
_).
abstract (reflexivity). abstract (reflexivity).
- intros V1 V2 Hseq. unshelve eexists.
   exact (_revSieveTransf_sieveEquiv Hseq). exact (_sieveTransf_sieveEquiv Hseq).
abstract(intros; rewrite -> _surProp_sieveEquiv; reflexivity).
abstract(intros; rewrite -> _injProp_sieveEquiv; reflexivity).
- intros V1 V2 V3 Hseq12 Hseq23. unshelve eexists. exact (sieveTransf_Compos Hseq12
Hseq23).
exact (sieveTransf_Compos (_revSieveTransf_sieveEquiv Hseq23)
(_revSieveTransf_sieveEquiv Hseq12)).
abstract(intros; cbn_transf;  rewrite -> _injProp_sieveEquiv; rewrite ->
_injProp_sieveEquiv; reflexivity).
abstract(intros; cbn_transf;  rewrite -> _surProp_sieveEquiv; rewrite ->
_surProp_sieveEquiv; reflexivity).
Defined.

Section interSieve.

 Section Section1.
 Variables (G : vertexGene) (VV : sieveFunctor G)
           (G' : vertexGene) (g : 'Gene( G' ~> G ))
           (UU : sieveFunctor G').

Record type_interSieve H :=
  { _factor_interSieve : 'Sieve( H ~> _ | UU ) ;
    _whole_interSieve : 'Sieve( H ~> _ | VV ) ;
    _wholeProp_interSieve : _whole_interSieve :>sieve_
        == (_factor_interSieve :>sieve_) o>functor_[functor_ViewOb G] g }.

Definition rel_interSieve H : crelation (type_interSieve H).
intros v v'. exact (((_factor_interSieve v == _factor_interSieve v') *
(_whole_interSieve v == _whole_interSieve v')) %type ).
Defined.

Instance equiv_interSieve H : Equivalence (@rel_interSieve H).
abstract(unshelve eexists;
[ (move; intros; move; split; reflexivity)
| (move; intros ? ? [? ?]; move; intros; split; symmetry; assumption)
| (move; intros ? ? ? [? ?] [? ?]; move; intros; split; etransitivity; eassumption)]).
Qed.

Definition interSieve : sieveFunctor G'.
Proof. unshelve eexists.
{ (* functor *) unshelve eexists.
  - (* typeOf_objects_functor *) intros H.
    + (* relType *) unshelve eexists. exact (type_interSieve H).
      exact (@rel_interSieve H).
      exact (@equiv_interSieve H).
```

```
  - (* typeOf_arrows_functor *) unfold typeOf_arrows_functor. intros H H'.
    +  (* relFunctor *) unshelve eexists.
      * (* -> *) cbn_. intros h vg'. unshelve eexists.
          exact: (h o>sieve_ (_factor_interSieve vg')).
          exact: (h o>sieve_ (_whole_interSieve vg')).
          abstract(cbn_; tac_unsimpl; rewrite <- 2!_natural_transf;
          rewrite -> _wholeProp_interSieve, _functorialCompos_functor'; reflexivity).
      * (* Proper *) abstract(move; autounfold;
      intros h1 h2 Heq_h vg'1 vg'2; case => /= Heq_vg' Heq_vg'0;
      split; cbn_; rewrite -> Heq_h;  [rewrite -> Heq_vg' | rewrite -> Heq_vg'0];
reflexivity).
  - (* typeOf_functorialCompos_functor *) abstract(move; intros; autounfold; split;
cbn_;
  rewrite -> _functorialCompos_functor; reflexivity).
  - (* typeOf_functorialIdent_functor *) abstract(move; intros; autounfold; split;
cbn_;
    rewrite -> _functorialIdent_functor; reflexivity). }
{ (* transf *) unshelve eexists.
  - (* typeOf_arrows_transf *) intros H. unshelve eexists.
    + (* -> *) cbn_; intros vg'. exact: ((_factor_interSieve vg') :>sieve_).
    + (* Proper *)  abstract(move; autounfold; cbn_;
    intros vg'1 vg'2; case => /= Heq0 Heq; rewrite -> Heq0; reflexivity).
  - (* typeOf_natural_transf *) abstract(move; cbn -[_arrows_functor]; intros;
  rewrite -> _natural_transf; reflexivity). }
Defined.

Lemma transf_interSieve_Eq  H  (v : 'Sieve(H ~> _ | interSieve )) :
 ((_factor_interSieve v) :>sieve_ ) == (v :>sieve_ ) .
Proof. reflexivity.
Qed.

Global Instance whole_interSieve_Proper  H : Proper (equiv ==> equiv)
 (@_whole_interSieve  H : 'Sieve( H ~> _ | interSieve  ) -> 'Sieve( H ~> _ | VV )).
Proof.    move. cbn_. intros v1 v2 [Heq Heq']. exact Heq'.
Qed.

Global Instance factor_interSieve_Proper  H : Proper (equiv ==> equiv)
 (@_factor_interSieve  H : 'Sieve( H ~> _ | interSieve  ) -> 'Sieve( H ~> _ | UU )).
Proof.    move. cbn_. intros v1 v2 [Heq Heq']. exact Heq.
Qed.

Definition interSieve_projWhole : transf interSieve VV.
Proof. unshelve eexists. unshelve eexists.
- (* -> *) exact: _whole_interSieve.
- (* Proper *) exact: whole_interSieve_Proper. (* abstract (typeclasses eauto).  *)
- (* typeOf_natural_transf *) abstract(intros H H' h f; cbn_; reflexivity).
Defined.

Definition interSieve_projFactor : sieveTransf interSieve UU.
Proof. unshelve eexists.  unshelve eexists. unshelve eexists.
- (* -> *) exact: _factor_interSieve.
- (* Proper *) exact: factor_interSieve_Proper. (* abstract (typeclasses eauto).  *)
- (* typeOf_natural_transf *) abstract(intros H H' h f; cbn_;  reflexivity).
- (* _commute_sieveTransf *) abstract(move; cbn_; intros; reflexivity).
Defined.

End Section1.

Definition pullSieve G VV G' g := @interSieve G  VV G' g (identSieve G').
Definition meetSieve G VV UU := @interSieve G VV G (@identGene G) UU.
```

```
Definition pullSieve_projWhole G VV G' g :
transf (@pullSieve G VV G' g) VV
:= (@interSieve_projWhole G  VV G' g (identSieve G')).

Definition pullSieve_projFactor G VV G' g :
sieveTransf (@pullSieve G VV G' g) (identSieve G')
:= (@interSieve_projFactor G  VV G' g (identSieve G')).

Definition meetSieve_projFactor G VV UU :
sieveTransf (@meetSieve G VV UU) UU := @interSieve_projFactor G VV G (@identGene G)
UU  .

Definition meetSieve_projWhole G VV UU :
sieveTransf (@meetSieve G VV UU) VV.
exists (interSieve_projWhole _ _ _).
intros H v; simpl. rewrite -> _wholeProp_interSieve.
(* HERE *) abstract(exact: identGene_composGene).
Defined.

End interSieve.

Existing Instance whole_interSieve_Proper.
Existing Instance factor_interSieve_Proper.

Section sumSieve.

Section Section1.
Variables (G : vertexGene) (VV : preSieve G).

Record typeOf_outer_sumSieve :=
  { _object_typeOf_outer_sumSieve :> vertexGene ;
    _arrow_typeOf_outer_sumSieve :> 'preSieve( _object_typeOf_outer_sumSieve ~> G |
VV ) }.

(* higher/congruent structure is possible... *)
Variables (WP_ : forall (object_: vertexGene) (outer_: 'preSieve( object_ ~> G | VV )),
 sieveFunctor object_).


(*
(* higher/congruent structure is possible... *)
Definition typeOf_sieveCongr :=
  forall (object_ : vertexGene)
  (outer_ outer_' : 'preSieve( object_ ~> _ | VV )),
outer_  == outer_' ->
sieveEquiv (WP_  outer_) (WP_  outer_').

Variables WP_congr : typeOf_sieveCongr. *)

Record type_sumSieve H :=
  { _object_sumSieve : vertexGene ;
    _outer_sumSieve : 'preSieve( _object_sumSieve ~> G | VV ) ;
    _inner_sumSieve : 'Sieve( H ~> _ | WP_ _outer_sumSieve ) }.

Inductive rel_sumSieve H  (wv : type_sumSieve H) : type_sumSieve H  -> Type :=
| Rel_sumSieve : forall
  (inner': (WP_  (_outer_sumSieve wv)) H),
  (* higher/congruent structure is possible... *)
  (* inner' :>transf_ (WP_congr Heq_outer) == (_inner_sumSieve wv) -> *)
  inner'  == (_inner_sumSieve wv) ->
  rel_sumSieve wv
```

```
  {| _object_sumSieve := _ ;
  _outer_sumSieve := _ ;
  _inner_sumSieve := inner' |}.

Instance rel_sumSieve_Equivalence H : Equivalence (@rel_sumSieve H).
abstract(unshelve eexists;
      [ (intros [object_wv outer_wv inner_wv]; constructor; reflexivity)
      | (* intros wv1 wv2 []. *) (intros [object_wv1 outer_wv1 inner_wv1] [object_wv2
outer_wv2 inner_wv2] [];
       constructor; symmetry; assumption)
      | (intros wv1 wv2 wv3 Heq12 Heq23; destruct Heq23 as [ inner3  Heq23'];
      destruct Heq12 as [ inner2  Heq12']; simpl; constructor; simpl;
       rewrite -> Heq23'; simpl; rewrite -> Heq12'; simpl; reflexivity)]).
Qed.

(* TODO: sumSieve_projOuter : sumSieve -> UU  *)
Definition sumSieve : sieveFunctor G.
Proof. unshelve eexists.
{ (* functor *) unshelve eexists.
  - (* typeOf_objects_functor *) intros H.
    + (* relType *) unshelve eexists. exact (type_sumSieve H).
    + (* Setoid *)  exact (@rel_sumSieve H).
     (* exists (equiv @@ (@compos_sumSieve H))%signature. *)
    + (* Equivalence *) exact: rel_sumSieve_Equivalence.
  - (* typeOf_arrows_functor *) move. intros H H'.
    (* relFunctor *) unshelve eexists.
    + (* -> *) simpl. intros h wv. unshelve eexists.
       exact: (_object_sumSieve wv). exact: (_outer_sumSieve wv).
       exact: (h o>sieve_ _inner_sumSieve wv).
    + (* Proper *) abstract(move;  autounfold; simpl;
    intros h1 h2 Heq_h [object_wv1 outer_wv1 inner_wv1] wv2 Heq; tac_unsimpl;
    case: wv2 / Heq => /= [ inner_wv2  Heq12']; constructor; simpl;
    rewrite -> Heq_h , Heq12'; reflexivity).
  - (* typeOf_functorialCompos_functor *) abstract(intros H H' h H'' h' [object_wv
outer_wv inner_wv];
     simpl; constructor; simpl; rewrite -> _functorialCompos_functor; reflexivity).
  - (* typeOf_functorialIdent_functor *) abstract(intros H [object_wv outer_wv
inner_wv];
  simpl; constructor; simpl; rewrite ->  _functorialIdent_functor; reflexivity). }
{ (* transf *) unshelve eexists.
  - (* typeOf_arrows_transf *) intros H. unshelve eexists.
    + (* -> *) simpl; intros wv. exact: ((_inner_sumSieve wv :>sieve_) o>functor_
(_outer_sumSieve wv :>preSieve_)).
    + (* Proper *) abstract(move; autounfold; simpl;
    intros wv1 wv2 Heq; tac_unsimpl;
    case: wv2 / Heq => /= [ inner_wv2  Heq12']; tac_unsimpl;
    rewrite -> Heq12'; reflexivity).
  - (* typeOf_natural_transf *) abstract(move; cbn_functor; move; cbn_functor; intros H
H' h wv;
  rewrite -> _functorialCompos_functor'; rewrite -> _natural_transf; reflexivity). }
Defined.

End Section1.

Section genSieve.

Definition genSieve (U : vertexGene) (UU : preSieve U)
  := (sumSieve (fun (object: vertexGene) (outer: 'preSieve( object ~> U | UU )) =>
identSieve object ) ).

Definition preSieveTransf_unit  (U : vertexGene) (UU : preSieve U)  :
```

```
  forall G (outer: 'preSieve( G ~> U | UU )), 'Sieve( G ~> U | (genSieve UU)  ) .
Proof. intros. exists _ outer. exact: (identGene). Defined.

Definition transf_of_preSieveTransf
  (U : vertexGene) (UU : preSieve U) (V : vertexGene) (VV : sieveFunctor V)
   (ff : forall G, 'preSieve( G ~> U | UU ) -> 'Sieve( G ~> V | VV) ) :
   transf (genSieve UU) VV .
Proof. unshelve eexists. unshelve eexists.
- (* -> *) intros u. exact ( (_inner_sumSieve u) o>functor_ (ff _ (_outer_sumSieve
u)) ).
- (* Proper *) abstract(move; move => u1 u2 [inner_u Heq]; cbn_transf; rewrite -> Heq;
reflexivity).
- (* typeOf_natural_transf *) abstract(intros H H' h u; cbn_sieve; rewrite ->
_functorialCompos_functor'; reflexivity).
Defined.

Definition preSieveTransf_of_transf (U : vertexGene) (UU : preSieve U) (V : vertexGene)
(VV : sieveFunctor V)
(ff : transf (genSieve UU) VV ) := (fun G (outer: 'preSieve( G ~> U | UU )) =>
((preSieveTransf_unit outer) :>transf_ ff) ).

Lemma transf_of_preSieveTransf_surj (U : vertexGene) (UU : preSieve U) (V : vertexGene)
(VV : sieveFunctor V)
(ff : transf (genSieve UU) VV ) :
forall  G (outer: 'Sieve( G ~> U | (genSieve UU) )),
outer :>transf_ ff == outer :>transf_ transf_of_preSieveTransf
(preSieveTransf_of_transf ff) .
Proof.  intros . unfold preSieveTransf_of_transf. cbn_sieve.
  rewrite -> _natural_transf. apply: _congr_relTransf.  split. cbn_sieve. apply:
identGene_composGene.
Qed.

Definition typeOf_commute_presieveTransfArrow
  (U : vertexGene) (UU : preSieve U) (V : vertexGene) (VV : sieveFunctor V)  (uv :
'Gene( U ~> V))
   (ff : forall G, 'preSieve( G ~> U | UU ) -> 'Sieve( G ~> V | VV) ) : Type :=
  forall (H : vertexGene)  (u : 'preSieve( H ~> U | UU )),
   (ff _ u ) :>sieve_  ==  (u :>preSieve_) o>functor_[functor_ViewOb _] uv .

Record presieveTransfArrow
(U : vertexGene) (UU : preSieve U) (V : vertexGene) (VV : sieveFunctor V)  (uv :
'Gene( U ~> V))  : Type :=
  { _transf_presieveTransfArrow :> forall G, 'preSieve( G ~> U | UU ) -> 'Sieve( G ~> V
| VV);
    _commute_presieveTransfArrow : typeOf_commute_presieveTransfArrow uv
_transf_presieveTransfArrow} .

Definition typeOf_commute_sieveTransfArrow
(G1 : vertexGene) (V1: sieveFunctor G1) (G2 : vertexGene) (V2: sieveFunctor G2)
(g12 : 'Gene( G1 ~> G2))  (vv : transf V1 V2)  : Type :=
  forall (H : vertexGene)  (v : 'Sieve( H ~> G1 | V1 )),
  (v :>transf_ vv) :>sieve_ ==  (v :>sieve_)  o>functor_[functor_ViewOb _] g12.

Record sieveTransfArrow (G1 : vertexGene) (V1: sieveFunctor G1) (G2 : vertexGene) (V2:
sieveFunctor G2)
(g12 : 'Gene( G1 ~> G2)) : Type :=
  { _transf_sieveTransfArrow :> transf V1 V2 ;
    _commute_sieveTransfArrow : typeOf_commute_sieveTransfArrow g12
_transf_sieveTransfArrow} .

Definition sieveTransfArrow_of_preSieveTransf
```

```
(U : vertexGene) (UU : preSieve U)  (V : vertexGene) (VV : sieveFunctor V)  (uv :
'Gene( U ~> V))
(ff : presieveTransfArrow UU VV uv)  : sieveTransfArrow (genSieve UU) VV uv.
Proof. exists (transf_of_preSieveTransf ff).
abstract(move; intros; cbn_sieve; rewrite <- _functorialCompos_functor', <-
_natural_transf;
rewrite -> _commute_presieveTransfArrow; reflexivity).
Defined.

Definition preSieveTransf_of_sieveTransfArrow
(U : vertexGene) (UU : preSieve U)  (V : vertexGene) (VV : sieveFunctor V)  (uv :
'Gene( U ~> V))
(ff : sieveTransfArrow (genSieve UU) VV uv)  : presieveTransfArrow UU VV uv.
Proof. exists (preSieveTransf_of_transf ff).
abstract(move; intros; unfold preSieveTransf_of_transf;
rewrite -> _commute_sieveTransfArrow;  cbn_sieve; rewrite -> _functorialIdent_functor;
reflexivity).
Defined.

Definition sieveTransfArrow_Compos :
forall U U' U'' F F'' F' (u_ : 'Gene( U'' ~> U')) (ff_ : sieveTransfArrow F'' F' u_)
(u' : 'Gene( U' ~> U)) (ff' : sieveTransfArrow F' F u'),
sieveTransfArrow F'' F (u_ o>gene u').
Proof.  intros. unshelve eexists.
- exact: (transf_Compos ff_ ff').
- abstract(move; intros; cbn_transf; do 2 rewrite -> _commute_sieveTransfArrow;
  rewrite <- _functorialCompos_functor'; reflexivity).
Defined.

End genSieve.

Definition sumSieve_projOuter :
forall (U : vertexGene) (UU : preSieve U)
(VV_ : forall (H: vertexGene) (outer_: 'preSieve( H ~> U | UU )), sieveFunctor H),
 sieveTransf (sumSieve VV_) (genSieve UU).
Proof. unshelve eexists. unshelve eexists.
- intros K. unshelve eexists.
  + (* _fun_relTransf *) intros wv. eexists. exact (_outer_sumSieve wv). exact
(_inner_sumSieve wv :>sieve_).
  + (* _congr_relTransf *) abstract(move;  intros wv1 wv2 [ inner_wv2  Heq_inner_wv2];
  cbn_transf; split;cbn_transf; rewrite -> Heq_inner_wv2; reflexivity).
- (* _natural_transf *) abstract(move; intros; cbn_sieve; split; cbn_sieve; rewrite  ->
_natural_transf; reflexivity ).
- (* _commute_sieveTransf *) abstract(move; intros; simpl; reflexivity).
Defined.

Definition sumSieve_sectionPull :
forall (U : vertexGene) (UU : preSieve U)
(VV_ : forall (H: vertexGene) (outer_: 'preSieve( H ~> U | UU )), sieveFunctor H)
(H: vertexGene)
(u: 'preSieve( H ~> _ | UU )),
 sieveTransf (VV_ H u)
  (pullSieve (sumSieve VV_) (u:>preSieve_)) .
Proof. unshelve eexists. unshelve eexists.
- intros K. unshelve eexists.
  + (* _fun_relTransf *) intros v.  unshelve eexists.
    * (* _factor_interSieve *)exact: ((v :>sieve_) ).
      (* _whole_interSieve *) unshelve eexists.
    * (* _object_sumSieve *) exact: H.
    * (* _outer_sumSieve *) exact: u.
    * (* _inner_sumSieve *) exact: v.
```

```
    * (* _wholeProp_interSieve *) abstract(simpl; reflexivity).
  + (* _congr_relTransf *) abstract(move;  intros v1 v2 Heq_v; split; autounfold;
simpl;
  first (rewrite -> Heq_v; reflexivity); split; autounfold; simpl;
  rewrite -> Heq_v; reflexivity).
- (* _natural_transf *) abstract(move; intros;  split; cbn_transf; last reflexivity;
cbn_sieve; rewrite -> _natural_transf; reflexivity).
- (* _commute_sieveTransf *) abstract(move; intros; simpl; reflexivity).
Defined.

Definition sumSieve_section:
forall (U : vertexGene) (UU : preSieve U)
(VV_ : forall (H: vertexGene) (outer_: 'preSieve( H ~> U | UU )), sieveFunctor H)
(H: vertexGene)
(u: 'preSieve( H ~> _ | UU )),
 transf (VV_ H u)  (sumSieve VV_) .
Proof. intros. exact: (transf_Compos (sumSieve_sectionPull _ _) (pullSieve_projWhole _
_) ).
Defined.

End sumSieve.

Section sumPreSieve.

Section Section1.
Variables (G : vertexGene) (VV : preSieve G).

Record typeOf_outer_sumPreSieve :=
  { _object_typeOf_outer_sumPreSieve :> vertexGene ;
    _arrow_typeOf_outer_sumPreSieve :> 'preSieve( _object_typeOf_outer_sumPreSieve ~> G
| VV ) }.

(* higher/congruent structure is possible... *)
Variables (WP_ : forall (object_: vertexGene) (outer_: 'preSieve( object_ ~> G | VV )),
 preSieve object_).


Record type_sumPreSieve H :=
  { _object_sumPreSieve : vertexGene ;
    _outer_sumPreSieve : 'preSieve( _object_sumPreSieve ~> G | VV ) ;
    _inner_sumPreSieve : 'preSieve( H ~> _ | WP_ _outer_sumPreSieve ) }.

Inductive rel_sumPreSieve H  (wv : type_sumPreSieve H) : type_sumPreSieve H  -> Type :=
| Rel_sumPreSieve : forall
  (inner': (WP_  (_outer_sumPreSieve wv)) H),
  inner' :>preSieve_  == (_inner_sumPreSieve wv) :>preSieve_ ->
  rel_sumPreSieve wv
  {| _object_sumPreSieve := _ ;
  _outer_sumPreSieve := _ ;
  _inner_sumPreSieve := inner' |}.

Instance rel_sumPreSieve_Equivalence H : Equivalence (@rel_sumPreSieve H).
abstract(unshelve eexists;
      [ (intros [object_wv outer_wv inner_wv]; constructor; reflexivity)
      | (* intros wv1 wv2 []. *) (intros [object_wv1 outer_wv1 inner_wv1] [object_wv2
outer_wv2 inner_wv2] [];
       constructor; symmetry; assumption)
      | (intros wv1 wv2 wv3 Heq12 Heq23; destruct Heq23 as [ inner3  Heq23'];
      destruct Heq12 as [ inner2  Heq12']; simpl; constructor; simpl;
       rewrite -> Heq23'; simpl; rewrite -> Heq12'; simpl; reflexivity)]).
Qed.
```

```
(* TODO: sumPreSieve_projOuter : sumPreSieve -> UU  *)
Definition sumPreSieve : preSieve G.
Proof.
unshelve eexists.
  - (* typeOf_objects_functor *) intros H.
    +  exact (type_sumPreSieve H).
  - (* typeOf_arrows_transf *) intros H.
    + (* -> *) simpl; intros wv. exact: ((_inner_sumPreSieve wv :>preSieve_) o>functor_
(_outer_sumPreSieve wv :>preSieve_)).
Defined.

Definition sumPreSieve_projOuter : presieveTransfArrow (sumPreSieve)  (genSieve VV)
(identGene).
Proof.  unshelve eexists.
  -  intros H uv. exists _ (_outer_sumPreSieve uv). exact ((_inner_sumPreSieve
uv) :>preSieve_).
  - abstract(move; intros; cbn_sieve; rewrite <- _functorialCompos_functor';
    apply: _congr_relFunctor; first reflexivity; symmetry; exact:
identGene_composGene).
Defined.

End Section1.

End sumPreSieve.

Section sumPullSieve.

Section Section1.
Variables (G : vertexGene) (VV : preSieve G).

(*  *)
Variables (famVertex_ : forall (object: vertexGene) (outer: 'preSieve( object ~> G |
VV )),
  vertexGene).

Variables (famArrow_ : forall (object: vertexGene) (outer: 'preSieve( object ~> G |
VV )),
 'Gene( object ~> famVertex_  outer )).

Variables (famSieve_ : forall (object: vertexGene) (outer: 'preSieve( object ~> G |
VV )),
 sieveFunctor (famVertex_  outer)).

Variables (famInterPreSieve_ : forall (object: vertexGene) (outer: 'preSieve( object ~>
G | VV )),
 preSieve object).

Definition sumPullSieve := @sumSieve G  VV (fun object outer => interSieve (famSieve_
outer) (famArrow_ outer) (genSieve (famInterPreSieve_ outer)) ).

Definition sumPullSieve_projSumPreSieve :
 sieveTransf sumPullSieve (genSieve (sumPreSieve famInterPreSieve_)).
Proof. unshelve eexists. unshelve eexists.
- intros K. unshelve eexists.
  + (* _fun_relTransf *) intros wv. eexists.
    * { unshelve eexists; cycle 1. exact (_outer_sumSieve wv). exact (_outer_sumSieve
(_factor_interSieve (_inner_sumSieve wv))). }
    simpl. exact (_inner_sumSieve (_factor_interSieve (_inner_sumSieve wv))).
  + (* _congr_relTransf *) abstract (move;  intros wv1 wv2  [ inner_wv2
[[outer_factor_inner_wv2 Heq_inner_factor_inner_wv2_] Heq_whole_inner_wv2]];
```

```
  cbn_transf; split; cbn_transf; rewrite -> Heq_inner_factor_inner_wv2_; reflexivity).
- (* _natural_transf *) abstract(move; intros; cbn_sieve; split; cbn_sieve;
reflexivity).
- (* _commute_sieveTransf *) abstract(move; intros; cbn_sieve; rewrite ->
_functorialCompos_functor'; reflexivity).
Defined.

End Section1.
End sumPullSieve.

Definition typeOf_commute_preSieveTransf
(G : vertexGene) (V1 V2 : preSieve G) (vv : forall G : vertexGene, V1 G -> V2 G)  :
Type :=
  forall (H : vertexGene)  (v : 'preSieve( H ~> G | V1 )),
   (vv _ v ) :>preSieve_   ==  v :>preSieve_ .

Record preSieveTransf G (V1 V2 : preSieve G) : Type :=
  { _transf_preSieveTransf :> forall G : vertexGene, V1 G -> V2 G ;
    _commute_preSieveTransf : typeOf_commute_preSieveTransf _transf_preSieveTransf} .

Notation "f :>preSieveTransf_ ee" := (@_transf_preSieveTransf _ _ _ ee _ f)
  (at level 40, ee at next level) : poly_scope.

Lemma sumSieve_congrTransf (G : vertexGene) (UU1  : preSieve G)
G' ( UU2 : preSieve G')
(uu : forall G : vertexGene, UU1 G -> UU2 G)
(VV1_ : forall H : vertexGene, 'preSieve( H ~> _ | UU1 ) -> sieveFunctor H)
(VV2_ : forall H : vertexGene, 'preSieve( H ~> _ | UU2 ) -> sieveFunctor H)
(vv_ : forall (H: vertexGene) (u1: 'preSieve( H ~> _ | UU1 )),
 sieveTransf  (VV1_ _ u1) (VV2_ _ (uu _ u1 ))) :
transf  (sumSieve VV1_ ) (sumSieve VV2_).
Proof.  unshelve eexists.
- (* _arrows_transf *) intros K. unshelve eexists.
  (* _fun_relTransf *) intros vu. unshelve eexists.
  (* _object_sumSieve *) exact: (_object_sumSieve vu).
  (* _outer_sumSieve *) exact: (uu _ (_outer_sumSieve vu ) ).
  (* _inner_sumSieve *) exact: (_inner_sumSieve vu :>transf_ (vv_ _ _)).
  (* _congr_relTransf  *) abstract(move; intros vu1 vu2 [ inner_vu2  Heq_inner_vu2];
  simpl; constructor; simpl; rewrite -> Heq_inner_vu2; reflexivity).
- (*  _natural_transf  *) abstract(intros K K' k vvu; cbn_sieve;
  constructor; simpl;  rewrite -> _natural_transf; reflexivity).
Defined.

Lemma sumSieve_congr (G : vertexGene) (UU1 UU2 : preSieve G)
(uu : preSieveTransf UU1 UU2)
(VV1_ : forall H : vertexGene, 'preSieve( H ~> _ | UU1 ) -> sieveFunctor H)
(VV2_ : forall H : vertexGene, 'preSieve( H ~> _ | UU2 ) -> sieveFunctor H)
(vv_ : forall (H: vertexGene) (u1: 'preSieve( H ~> _ | UU1 )),
sieveTransf  (VV1_ _ u1) (VV2_ _ (uu _ u1))) :
sieveTransf  (sumSieve VV1_ ) (sumSieve VV2_).
Proof. unshelve eexists. (* _transf_sieveTransf *) exact: sumSieve_congrTransf.
(* _commute_sieveTransf *) abstract(intros K vu; simpl; rewrite ->
_commute_sieveTransf;  rewrite -> _commute_preSieveTransf; reflexivity).
Defined.

Definition typeOf_basePreSieve (U : vertexGene) (UU : preSieve U) :=
  forall (H : vertexGene) (u u' : 'preSieve( H ~> _ | UU )), u :>preSieve_ ==
u' :>preSieve_  -> u = u'.

Parameter basePreSieve : forall (U : vertexGene) (UU : preSieve U)
  (UU_base : typeOf_basePreSieve UU) , Type.
```

```
Inductive isCover : forall (U : vertexGene) (UU_pre : preSieve U) (UU : sieveFunctor
U), sieveTransf UU (genSieve UU_pre)  -> Type :=

| BasePreSieve_isCover : forall (U : vertexGene) (UU : preSieve U) (UU_base :
typeOf_basePreSieve UU),
    basePreSieve UU_base -> @isCover _ UU (genSieve UU) (sieveTransf_Ident _)

(*TODO | IdentSieve_isCover : forall (G : vertexGene),
 isCover (identSieve G) (identGene G ...) *)

 | InterSieve_isCover : forall (G : vertexGene) (VV_pre : preSieve G) (VV :
sieveFunctor G) (VV_transf : sieveTransf VV (genSieve VV_pre))
    (G' : vertexGene) (g : 'Gene( G' ~> G ))  (UU_pre : preSieve G') (UU : sieveFunctor
G') (UU_transf : sieveTransf UU (genSieve UU_pre)),
     @isCover  _ VV_pre VV VV_transf -> @isCover  _ UU_pre UU UU_transf ->  @isCover _
UU_pre (interSieve VV g UU) (sieveTransf_Compos (interSieve_projFactor _ _ _)
UU_transf)

| SumSieve_isCover : forall (G : vertexGene) (VV : preSieve G) (VV_base :
typeOf_basePreSieve VV)
(VV_base_cover : basePreSieve VV_base),
forall (famVertex_ : forall (object: vertexGene) (outer: 'preSieve( object ~> G |
VV )),
  vertexGene)
  (famPreSieve_ : forall (object: vertexGene) (outer: 'preSieve( object ~> G | VV )),
 preSieve (famVertex_  object outer))
 (famSieve_ : forall (object: vertexGene) (outer: 'preSieve( object ~> G | VV )),
 sieveFunctor (famVertex_  object outer))
 (famSieveTransf_ : forall (object: vertexGene) (outer: 'preSieve( object ~> G | VV )),
    sieveTransf (famSieve_  object outer) (genSieve (famPreSieve_  object outer)))
(famIsCover_ : forall (object: vertexGene) (outer: 'preSieve( object ~> G | VV )),
    @isCover _ (famPreSieve_ object outer) (famSieve_ object outer) (famSieveTransf_
object outer))

 (famPullArrow_ : forall (object: vertexGene) (outer: 'preSieve( object ~> G | VV )),
   'Gene( object ~> famVertex_  object outer ))
 (famPullPreSieve_ : forall (object: vertexGene) (outer: 'preSieve( object ~> G |
VV )),
      preSieve object)
  (famPullPreSieveTransf_ : forall (object: vertexGene) (outer: 'preSieve( object ~> G
| VV )),
      sieveTransfArrow (genSieve (famPullPreSieve_ object outer)) (genSieve
(famPreSieve_ object outer)) (famPullArrow_ object outer)),

  @isCover _ (sumPreSieve famPullPreSieve_) (sumPullSieve famPullArrow_ famSieve_
famPullPreSieve_)
    (sumPullSieve_projSumPreSieve famPullArrow_ famSieve_ famPullPreSieve_).


Section nerveSieve.

Variables (topPreSieveVertexes: vertexGene) (topPreSieve: preSieve topPreSieveVertexes)
(structCoSheaf: typeOf_objects_functor).

Inductive nerveSieve: forall (U : vertexGene) (UU_pre : (preSieve U)) (UU :
sieveFunctor U) (UU_transf: sieveTransf UU (genSieve UU_pre)) (UU_isCover : isCover
UU_transf),
forall (u_arrowTop : 'Gene( U ~> topPreSieveVertexes)) (UU_transfTop :
presieveTransfArrow UU_pre (genSieve topPreSieve) u_arrowTop),
forall (G : vertexGene) (g_sense : 'Gene( G ~> U)),
```

```
forall (dim: nat) (diffPreSieveVertexes: 'I_(dim) ->  vertexGene )
       (diffPreSieve: forall i : 'I_(dim), 'preSieve( (diffPreSieveVertexes i) ~> _ |
topPreSieve )), Type :=

| NerveSieve_Diff  (* at cell dim +1 , at coeffiecients degree +1 *) :
forall (U : vertexGene) (UU_pre : (preSieve U)) (UU_pre_base : typeOf_basePreSieve
UU_pre) (UU_pre_cover : basePreSieve UU_pre_base),

forall (u_arrowTop : 'Gene( U ~> topPreSieveVertexes)) (UU_transfTop :
presieveTransfArrow UU_pre (genSieve topPreSieve) u_arrowTop),

forall (famVertex_ : forall (object: vertexGene) (outer: 'preSieve( object ~> U |
UU_pre )), vertexGene)
(famPreSieve_ : forall (object: vertexGene) (outer: 'preSieve( object ~> U | UU_pre )),
preSieve (famVertex_ object outer))
(famSieve_ : forall (object: vertexGene) (outer: 'preSieve( object ~> U | UU_pre )),
sieveFunctor (famVertex_ object outer))
(famSieveTransf_ : forall (object: vertexGene) (outer: 'preSieve( object ~> U |
UU_pre )),
    sieveTransf (famSieve_  object outer) (genSieve (famPreSieve_  object outer)))
(famIsCover_ : forall (object: vertexGene) (outer: 'preSieve( object ~> U | UU_pre )),
    isCover (famSieveTransf_  object outer))

(famTopArrow_ : forall (object: vertexGene) (outer: 'preSieve( object ~> U | UU_pre )),
'Gene( (famVertex_ object outer) ~> topPreSieveVertexes ) )
(famTransfTop_ : forall (object: vertexGene) (outer: 'preSieve( object ~> U |
UU_pre )), presieveTransfArrow (famPreSieve_ object outer) (genSieve topPreSieve)
(famTopArrow_ object outer))

(famPullArrow_ : forall (object: vertexGene) (outer: 'preSieve( object ~> U |
UU_pre )), 'Gene( object ~> famVertex_ object outer ))
(famPullPreSieve_ : forall (object: vertexGene) (outer: 'preSieve( object ~> U |
UU_pre )),
      preSieve object)
(famPullPreSieveTransf_ : forall (object: vertexGene) (outer: 'preSieve( object ~> U |
UU_pre )),
      sieveTransfArrow (genSieve (famPullPreSieve_ object outer)) (genSieve
(famPreSieve_ object outer)) (famPullArrow_ object outer))
(famHeqArrow_ : forall (object: vertexGene) (outer: 'preSieve( object ~> U | UU_pre )),
   (*  (outer :>sieve_)  o>functor_[functor_ViewOb _]  u_arrowTop :=: *)  (UU_transfTop
_ outer) :>sieve_
  == (famPullArrow_ object outer) o>functor_[functor_ViewOb _] (famTopArrow_ object
outer) ),

forall (G : vertexGene) (g_sense : 'Gene( G ~> U)),
forall (dim: nat) (object: 'I_(S dim) -> vertexGene),
forall (outer: forall i : 'I_(S dim), 'preSieve( object i ~> U | UU_pre )),
forall (inner: forall i : 'I_(S dim),
               'Sieve( G ~> _ | interSieve (famSieve_ (object i) (outer i))
(famPullArrow_ (object i) (outer i))
                                            (genSieve (famPullPreSieve_ (object i)
(outer i))) ) ),
forall (inner_nerveSieve: forall i : 'I_(S dim),
  nerveSieve (famIsCover_ (object i) (outer i))
     (famTransfTop_ (object i) (outer i))
(*      ((_outer_sumSieve (((inner i) :>transf_ (interSieve_projWhole _ _ _)) :>transf_
(famSieveTransf_ (object i) (outer i)) )) :>preSieve_ ) *)
     ((_outer_sumSieve ( (famPullPreSieveTransf_ (object i) (outer i)) _ ((inner
i) :>transf_ (interSieve_projFactor _ _ _))  )) :>preSieve_ )
     (fun j : 'I_(dim) => _outer_sumSieve (UU_transfTop _ (outer (lift i j))))),
```

```
forall (inner_senseCompat : forall i : 'I_(S dim), ((inner i) :>sieve_)
o>functor_[functor_ViewOb _] ((outer i) :>preSieve_) == g_sense ),

forall (G_weight : structCoSheaf G),

nerveSieve (SumSieve_isCover UU_pre_cover famIsCover_ famPullPreSieveTransf_)
  (preSieveTransf_of_sieveTransfArrow (sieveTransfArrow_Compos
(sieveTransfArrow_of_preSieveTransf (sumPreSieve_projOuter famPullPreSieve_))
        (sieveTransfArrow_of_preSieveTransf UU_transfTop)))  g_sense
   (fun i : 'I_(S dim) => _outer_sumSieve (UU_transfTop _ (outer i)))

| NerveSieve_Gluing (* at same cell dim >= 0, at coefficients degree +1 *) :
forall (U : vertexGene) (UU_pre : (preSieve U)) (UU_pre_base : typeOf_basePreSieve
UU_pre) (UU_pre_cover : basePreSieve UU_pre_base),

forall (u_arrowTop : 'Gene( U ~> topPreSieveVertexes)) (UU_transfTop :
presieveTransfArrow UU_pre (genSieve topPreSieve) u_arrowTop),

forall (famVertex_ : forall (object: vertexGene) (outer: 'preSieve( object ~> U |
UU_pre )), vertexGene)
(famPreSieve_ : forall (object: vertexGene) (outer: 'preSieve( object ~> U | UU_pre )),
preSieve (famVertex_ object outer))
(famSieve_ : forall (object: vertexGene) (outer: 'preSieve( object ~> U | UU_pre )),
sieveFunctor (famVertex_ object outer))
(famSieveTransf_ : forall (object: vertexGene) (outer: 'preSieve( object ~> U |
UU_pre )),
    sieveTransf (famSieve_  object outer) (genSieve (famPreSieve_  object outer)))
(famIsCover_ : forall (object: vertexGene) (outer: 'preSieve( object ~> U | UU_pre )),
    isCover (famSieveTransf_  object outer))

(famTopArrow_ : forall (object: vertexGene) (outer: 'preSieve( object ~> U | UU_pre )),
'Gene( (famVertex_ object outer) ~> topPreSieveVertexes ) )
(famTransfTop_ : forall (object: vertexGene) (outer: 'preSieve( object ~> U |
UU_pre )), presieveTransfArrow (famPreSieve_ object outer) (genSieve topPreSieve)
(famTopArrow_ object outer))

(famPullArrow_ : forall (object: vertexGene) (outer: 'preSieve( object ~> U |
UU_pre )), 'Gene( object ~> famVertex_ object outer ))
(famPullPreSieve_ : forall (object: vertexGene) (outer: 'preSieve( object ~> U |
UU_pre )),
      preSieve object)
(famPullPreSieveTransf_ : forall (object: vertexGene) (outer: 'preSieve( object ~> U |
UU_pre )),
      sieveTransfArrow (genSieve (famPullPreSieve_ object outer)) (genSieve
(famPreSieve_ object outer)) (famPullArrow_ object outer))
(famHeqArrow_ : forall (object: vertexGene) (outer: 'preSieve( object ~> U | UU_pre )),
   (*  (outer :>sieve_)  o>functor_[functor_ViewOb _]  u_arrowTop :=: *)  (UU_transfTop
_ outer) :>sieve_
  == (famPullArrow_ object outer) o>functor_[functor_ViewOb _] (famTopArrow_ object
outer) ),

forall (G : vertexGene) (g_sense : 'Gene( G ~> U)),
forall (dim: nat) (diffPreSieveVertexes: 'I_(dim) ->  vertexGene )
       (diffPreSieve: forall i : 'I_(dim), topPreSieve (diffPreSieveVertexes i)),

forall (fam_nerveSieve: forall (object: vertexGene) (outer: 'preSieve( object ~> U |
UU_pre )),
      nerveSieve (famIsCover_ object outer)
      (famTransfTop_ object outer)
      (famPullArrow_ object outer)
      diffPreSieve),
```

```
forall (G_weight : structCoSheaf G),

nerveSieve (SumSieve_isCover UU_pre_cover famIsCover_ famPullPreSieveTransf_)
(preSieveTransf_of_sieveTransfArrow (sieveTransfArrow_Compos
(sieveTransfArrow_of_preSieveTransf (sumPreSieve_projOuter famPullPreSieve_))
      (sieveTransfArrow_of_preSieveTransf UU_transfTop)))  g_sense diffPreSieve

| NerveSieve_Base (* at cell dim = 0, at coeffiecients degree = 0 *) :
forall (U : vertexGene) (UU_pre : preSieve U) (UU_pre_base : typeOf_basePreSieve
UU_pre) (UU_pre_isBase: basePreSieve UU_pre_base ),
forall (u_arrowTop : 'Gene( U ~> topPreSieveVertexes)) (UU_transfTop :
presieveTransfArrow UU_pre (genSieve topPreSieve) u_arrowTop),
forall (G : vertexGene) (g_sense : 'preSieve( G ~> _ | UU_pre)),
nerveSieve (BasePreSieve_isCover UU_pre_isBase)  UU_transfTop (g_sense :>preSieve_ )
 (fun i : 'I_0 => (ffun0 (card_ord 0) : forall i : 'I_(0), 'preSieve( ((ffun0 (card_ord
0)) i) ~> _ | topPreSieve )) i).

End nerveSieve.
End COMOD.

(** # #
#+END_SRC

Voila.

# # **) C6 / coq
```