title / ereview # Proof-assistants, sheaves and applications title / ereview

short / ereview For the computer, the relevance of "how" is witnessed onto the grammar/syntax and thus cannot be avoided, and Per Martin-Löf "equality of equalities" becomes, when taken seriously, (cubical) homotopy-path types (hott). Then the elimination principle for such path types, which should be some monodromy principle for locally constant sheaves, suggests the mediation via some univalence-axiom. Elsewhere Kosta Dosen cut-elimination confluence techniques in category theory provide some direct-encoding alternative to type theory (internal-logic encoding). Moreover the simplicial methods, instead of the globular/cubical shape of the boundaries, seem to be the better context for (Cech) sheaf cohomology, quasicategories-operad algebras... Concretely this attempt at grammatical sheaf cohomology has shown that $H^1(\Delta^2) = 0$, $H^1(S^1) \neq 0$. short / ereview

reviewers / ereview reviewers / ereview

Intro

Kosta Dosen's programming language for categories and sheaves via cut-elimination.

Short: The goal of this publication is to remind potential contributors of the ongoing project to implement Kosta Dosen's programming language for categories and sheaves via cut-elimination. I will use plain English words to describe the essential insights and the future itinerary, with the understanding that there is already sufficient Coq-code evidence to support these approximations. The summary is that: Kosta Dosen's categorial cut-elimination book had already discovered that natural transformations formulated as operations on arrows is what allows cut-elimination's computation and confluence's decidability of equality of arrows. Therefore, the contributors of the so-called "directed-type-theory arrow-induction" cannot do away with citing Kosta Dosen.

A category is made of objects and arrows. And objects are the same thing as functors from the unit category. Also, arrows are the same things as natural transformations from the unit category. In other words, functors are objects-expressions in the codomain category under the context of an object-variable in some domain category, and natural transformations are arrows-expressions under some object-variable context. What happens if we allow contexts under some arrow-variable? Or contexts under some element-variable of some general profunctor-hom? Then natural transformations would be special cases of something when the domain is the unit profunctor-hom (the hom of some category). This is the insight that leads Kosta Dosen to say that any ordinary natural transformation

$$t_A : F\ G\ A \to H\ K\ A$$

can be formulated as an "*antecedental transformation*"

$$\frac{f : K\ A \to B}{Hf \circ t_A : F\ G\ A \to H\ B}$$

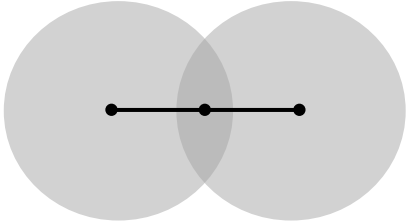
with primitive name "$H - \circ t\_$" in the language, or can be formulated as a "*consequential transformation*"

$$\frac{f : B \to G\ A}{t_A \circ Ff\ : F\ B \to H\ K\ A}$$

with primitive name "$t\_ \circ F -$" in the language. And in the special case when t_A is the counit of an adjunction with the functor F left adjoint to G and with H, K absent ($H = 1, K = 1$), then these various formulations allow for the elimination of the composition $\circ$ (cut-elimination). Of course, this cut elimination is except those (apparent) cuts baked into the primitive language of the antecedental/consequential counit and unit; nevertheless, the decidability of the equality of the arrows still holds via the confluence lemma.

In practice, these cut-elimination techniques are only the kernel for some general contextual proof-assistant programming language which is more expressive. For example, while the surface language would allow expressions in non-empty contexts such as functors or natural elements/transformations or antecedental/consequential transformations and composition/substitutions of those, the target-compilation language is concerned with decisions only on the resulting objects and arrows (with already-applied functors and transformations on them). Another example is that the surface language may allow general profunctor-hom constructions ($Hom_1 : \mathrm{cat}\mathrm{X}^{\mathrm{op}} \times \mathrm{cat}\mathrm{Y} \to \mathrm{Set}$) such as pairs of composable arrows (via the tensor profunctor-hom $\exists B.\, Hom_1(A, B) \times Hom_2(B, C)$), or functions on arrows (via the cotensor profunctor-hom $\forall B.\, Hom_1(A, B) \to Hom_2(C, B)$)), or pairs of parallel arrows (via the product profunctor-hom $Hom_1(A, B) \times Hom_2(A, B)$), or square of arrows (via the comma of the profunctor-hom $\Sigma A\ B.\, Hom_1(A, B)$), or user-opaque profunctor-hom variables. Then the cut-elimination would, at least, still traverse those expressions.

A sheaf is data defined over some topology, and sheaf cohomology is linear algebra with data defined over some topology. The type of this data is unlike the natural numbers, rational numbers, real numbers, or complex numbers data types. Values of this sheaf data type are functions, or more accurately are "*germs"* of functions, that is a germ is any function which is relevant only locally near some point (so that two functions locally-the-same near some point may represent the same germ value). Obviously for the computer, it is out of question to talk directly about points, but rather it is often enough to talk only about covers of the space by open neighborhoods which could be refined until it is fine/good enough to capture all the linear algebra. Now the relation between the former approach (singular cohomology via some fine acyclic resolution by sheaves) and the latter (Cech cohomology of the nerve of some good cover) becomes clear when the space is barycentric subdivided.



Approximately, starting with the exact sequence

$$0 \to \ker j_{\mathcal{U}} \to C^*(M) \xrightarrow{j_{\mathcal{U}}} C^*(\mathcal{U}) \to 0$$

where $\mathcal{U}$ is some cover of the space M and $j_{\mathcal{U}}$ restricts any singular cochain (function) defined on all simplices to only the small simplices contained within any $U \in \mathcal{U}$, then the barycentric subdivision subordinate to $\mathcal{U}$ ensures that j_U is some homotopy equivalence and therefore, at the filtered/inductive colimit over the refinements of $\mathcal{U}$, that the Cech complex (where the refinements are total) is equivalent to the complex of germs (where the refinements of opens are local around each point).

A closer inspection reveals that there is some intermediate formulation which is computationally-better that Cech cohomology: at least for the standard simplexes (line, triangle, etc.), then intersections of opens could be internalized as primitive/generating opens for the cover and become points in the nerve of this cover (as suggested by the barycentric subdivision). This redundant storage space for functions defined over the topology is what allows possibly-incompatible functions to be glued, and to prove the acyclicity for the standard simplex (and to compute how this acyclicity fails in the presence of holes in the nerve). For example, the sheaf data type:

$$F(U0) \coloneqq \text{sum over the slice U0 U01} = \mathbb{Z} \oplus \mathbb{Z};$$

$$F(U1) \coloneqq \mathbb{Z} \oplus \mathbb{Z}; F(U01) \coloneqq \mathbb{Z}$$

$$F(U) = \text{kan extension} = \mathbb{Z} \oplus \mathbb{Z} \oplus \mathbb{Z};$$

gives the gluing operation

$$\text{gluing}: F(U0) \oplus F(U1) \oplus F(U01) \to F(U)$$

$$\big((f0, f01), (g1, g01), (h01)\big) \mapsto (f0, g1, f01 + g01 - h01)$$

where the signed sum generalizes to higher degrees because the Euler characteristic is 1.

In practice, the implementation of the topology would be as some categorial site in the form of some closure operator $j: \Omega \to \Omega$ where $\Omega(A)$ is the classifier of (sub-)objects (sieves) of the object A, and where $j(\mathcal{U})(f) \coloneqq f^*\mathcal{U} \in J(A)$ is the (opaque) set of witnesses that the pullback-sieve $f^*\mathcal{U}$ is covering (remember that the truthness that $\mathcal{U}$ is covering is expressed as $\mathcal{U} \in J(A)$, iff $\forall f.\, j(\mathcal{U})(f)$). But it is better to consider any presheaf in the slice over A, rather than only subfunctors because then everything is expressible as profunctor-homs (of witnesses) over some slice categories.

The kernel of this cut-elimination confluence for adjunctions had already been programmed into the Coq proof-assistant:

https://github.com/1337777/dosen/blob/master/dosenSolution1.v

References:

Kosta Dosen, Cut-elimination in categories.

https://github.com/1337777/dosen/blob/master/dosenSolution1.v

Outline

The augmented cochain complex for the (barycentric subdivision of the) 2-simplex Δ^2 is acyclic: the contracting homotopy is some grammatical (possibly-incompatible) gluing operation of the presheaf. Indeed the existence of semantic models satisfying such postulated grammar operations is demonstrated (by using the barycentric U0 U1 U01 as the good cover instead of U0 U1, ...). Finally the circle S^1 specifies some hole in the nerve of this 2-simplex Δ^2, thus breaking this acyclicity $H^1(S^1) \neq 0$ (for the twisted locally constant sheaf). In full generality, with Kosta Dosen techniques, it should be possible to program the nerve of any topological site with structure (co-)sheaf, its (possibly-incompatible) gluing-differential and (co-)sheaf (co-)homology, upto any (Verdier) duality.

Here is the outline for the implementation of the idea (in COQ mathcomp ssralg). The nerve inductive type (where structCoSheafO could come from duality or from extension by 0, Stacks Lemma 00A4) is approximately:

```
| GlueDiff : (_ : structCoSheafO [i_0; ... i_n])
 (coface : forall i, nerve dimCoef [i; i_0; ... i_n] )
 (face : forall j, nerve dimCoef [i_0; ... <i_j>; ... i_n]),
nerve (dimCoef+1) [i_0; ... i_n]  | Diff : ... | Empty : ... .
```

The presheaf record structure is approximately:

```
Structure shfyCoef_struct := { shfyCoef : forall (cell: seq vertexSet), Type;
 restrict : forall j, shfyCoef [i_0; ... <i_j>; ... i_n] -> shfyCoef [i_0; ... i_n];
 glue : (f_ : forall i, shfyCoef [i; i_0; ... i_n]) -> shfyCoef [i_0; ... i_n];
 _ : restrict (glue (fun i => f_ i)) = glue (fun i => restrict (f_ i));
 _ : glue (fun i => @restrict 0 f : shfyCoef [i; i_0; ... i_n]) = f;  ... }
```

Then the definition of the gluing-differential operation is approximately:

```
Definition diffGluing: (ff_ : forall cell, nerve dimCoef cell -> shfyCoef F_sh cell) ->
(forall cell, nerve (dimCoef+1) cell -> shfyCoef F_sh cell).
```

Finally the homotopy to the null-complex when the nerve has no holes is approximately:

```
glue (fun i => restrict 0 (ff_ [i_0; ... i_n]) +
   (\sum_(j < n+1) (-1)^(1 + j) * restrict (1 + j) (ff_ [i; i_0; ... <i_j>; ... i_n])))
+ \sum_(j < n+1) (-1)^j * restrict j
     (glue (fun i => ff_ [i; i_0; ... <i_j>; ... i_n])
      + \sum_(k < n) (-1)^k * restrict k (ff_ [i_0; ... <i_k>; ... <i_j>; ... i_n])) =
ff_ [i_0; ... i_n]
```

This setup is similar as the acyclicity of the mapping cone of some (gluing with shift -1) quasi-isomorphism, so that the shift is now -2. And this acyclicity proof is new, when compared to the Stacks Project Lemma 03AT (exactness of the source chain complex), Lemma 0G6S (exactness of the Cech complex when U = U_i for some i), Lemma 01EM (exactness of the source chain complex), Lemma 01FM (homotopy equivalence with the alternating, ordered, semi-ordered Cech complexes), Lemma 02FU (exactness of the Cech complex of stalks). This setup contains the proof that d ∘ d = 0, which should rely on the usual simplicial face relations δ_j ∘ δ_i = δ_i ∘ δ_(j+1) for i <= j; however grammatically it becomes relevant how those faces-of-faces are specified and oneself may care about relevance up to d ∘ d ≠ 0 with d ∘ d ∘ d = 0 ... As for the Scholze Lean experiment, the first step has been some acyclicity proof.

The generalization to computational logic constructors is by noting that the pullback-sieve and the sum-sieve should be blended as sum-of-pullbacks-sieve in the definition of topological sites, approximately:

```
| GluingDiff : (sievesV_ : for all G with Site( G ~> U | in sieveU ),
for some G' with Site( G ~> G' ),  is-sieve at G' refining-the-fixed-cover)

  (u_ : forall j, Site( G_ j ~> U | in sieveU ))

  (ff_ : forall j, PreSheaves( Nerve structSheafO (sievesV_ (u_j)) [u_0; ... <u_j>; ...
u_n] ~> Sheafified F ))

⊢ PreSheaves( Nerve structSheafO (sum-of-pullbacks sievesV_ over sieveU) [u_0; ... u_n]
~> Sheafified F )
```

The significance of such research programme, prompted from the mathematicians Kosta Dosen and Pierre Cartier, is similar as the significance of homotopy type theory or differential linear logic. Earlier COQ proofs of cut-elimination confluence for Maclane associativity coherence (the pentagon is some recursive square), adjunctions, comonads, cartesian products, enriched categories, internal categories, 2-categories, were published. This is sufficient evidence to "pay" long-term attention into this research programme.

More motivations

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.

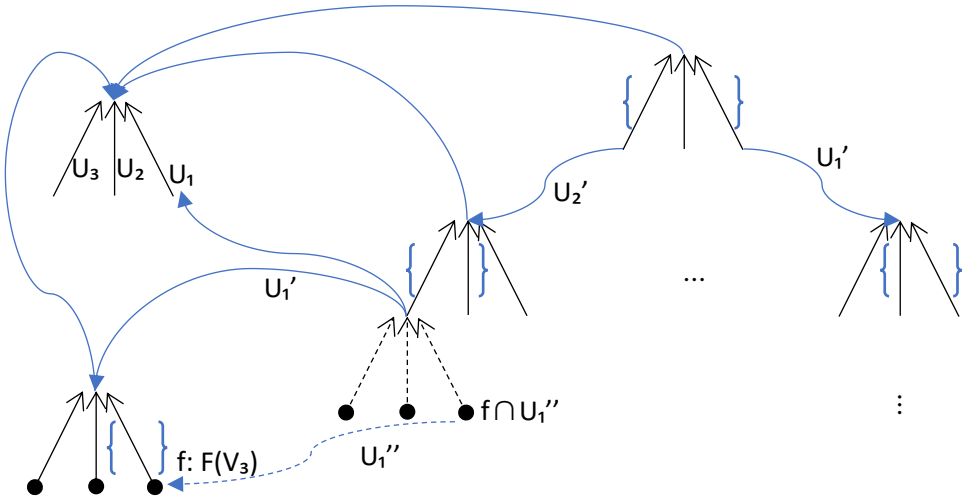


Diagram 1. Each nested basic sieve is some refinement of the fixed cover $\mathcal{U} = \{U_i \to U\}_{i \in I}$. The grammatical total/sum sieve is no longer one-to-one (mono) into the actual arrows of the site.

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 $U_{i_0} \times_U \ldots \times_U U_{i_p}$ exist in $\mathcal{C}$. Consider the chain complex $Z_{U,\cdot}$ of abelian presheaves

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

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

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

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 $\mathcal{F} \in \mathrm{Ob}\left(PAb(\mathcal{C})\right)$ ■

Note that any of the products $U_{i_0} \times_U \ldots \times_U U_{i_p}$ 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

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

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

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 $H^i\left(U_{i_0} \times_U \ldots \times_U U_{i_p}, \mathcal{F}\right) = 0$ for all $i > 0$, all $p \geq 0$ and all $i_0, \ldots, i_p \in I$. 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:

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

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

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.

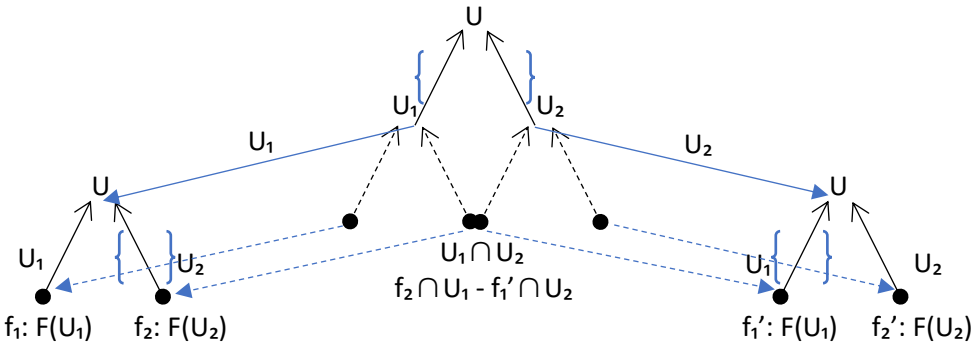


Diagram 2. Instance of *Diagram 1* where all the refinements from the fixed top cover-sieve are identities.

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
```

Applications, COQ script, qualifying reviewer form

Reviewer form

WorkSchool 365 is your education marketplace where you get qualified to review stories by new authors, with shareable transcripts receipts of your qualifications.

Sign-in into this Word online document or via the Microsoft Marketplace, to fill in this quiz form:

https://workschool365.com

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

Q1. The MODOS end-goal is:

(A) proof-assistant for the computational logic of sheaves.

(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

and such research programme would require some new WorkSchool 365 education market for

Coq script

Download this Word document or its Coq script at:

https://github.com/1337777/cartier

Or in Word, search “Insert; Add-ins; WorkSchool 365” to play this Coq script.

```
C5 / coq (** # #
#+TITLE: cartierSolution0.v

Proph

https://github.com/1337777/cartier/blob/master/cartierSolution11.v

Grammatical sheaf cohomology, its MODOS proof-assistant and WorkSchool 365 market for
learning reviewers

-----

#+BEGIN_SRC coq :exports both :results silent # # **)

Module Example.
```

```
From Coq Require Import Arith Lia Program.
From Equations Require Import Equations.
From mathcomp Require Import ssreflect ssrfun ssrbool eqtype ssrnat seq path fintype
tuple finfun bigop ssralg.

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

Set Equations With UIP.  Set Equations Derive Equations. Set Equations Derive
Eliminator.

Import EqNotations. Import GRing. Open Scope ring_scope.

Section nerve.
Variable topSieve : nat.
Variable structCoSheafO : forall (cell: seq ('I_ (S topSieve))), bool.

Global Instance  eq_comparable_eqdec_ord  : EqDec ('I_ (S topSieve)) := @eq_comparable
(ordinal_eqType (S topSieve)).
(* Global Instance eq_comparable_eqdec (T : eqType) : EqDec T := @eq_comparable T. *)

Equations pop {T : Type} (n : nat) (s : seq T) : seq T by struct s :=
pop n  [::] := [::];
pop 0 (x :: s') := s';
pop n'.+1  (x :: s') := x :: pop n' s'.

Transparent pop. Arguments pop _ !_ !_  : simpl nomatch.

Lemma pop_pop T Le Ge (_ : Le <= Ge) (cell : seq T)  :
pop Ge (pop Le cell)  = pop Le (pop (S Ge) cell).
Proof. funelim (pop Le cell); simp pop.
-  reflexivity.
-  reflexivity.
- case : Ge H0 => [// | Ge0 ] H0. simp pop. by  rewrite H.
Qed.

Lemma pop_take_drop  (T : Type) (s : seq T)  (n : nat) : pop n s = take n s ++  drop
( S n) s.
Proof. funelim (pop n s);simp pop; simpl.
reflexivity.
rewrite drop0;  reflexivity.
congr ( _ :: _ ). assumption. Qed.

(* LITTLE ERRATA: TODO: FINALLY SHOULD RE-ALLOW  pop_spec i [] [] *)
Inductive pop_spec: forall (popIndex: nat)  (cell: seq ('I_ (S topSieve)))  (popCell:
seq ('I_ (S topSieve))), Type :=
| PopZero cellHd cell : pop_spec 0 (cellHd :: cell) cell
| PopSucc popPos cell popCell cellHd : pop_spec popPos cell popCell -> pop_spec (S
popPos) (cellHd :: cell) (cellHd :: popCell).

Equations Derive Signature NoConfusion NoConfusionHom for pop_spec.

Lemma pop_spec_prop1 (popIndex: nat) (cell: seq ('I_ (S topSieve))) (popCell: seq ('I_
(S topSieve)))
(ns: pop_spec popIndex cell popCell): popIndex < (size cell).
Proof. by induction ns. Qed.

Lemma pop_spec_prop2 (popIndex: nat) (cell: seq ('I_ (S topSieve))) (popCell: seq ('I_
(S topSieve)))
(ns: pop_spec popIndex cell popCell): (size cell) = S (size popCell).
Proof.  induction ns. reflexivity. simpl. congr (S _). assumption. Qed.
```

```
Lemma popP (popIndex: nat)  (cell: seq ('I_ (S topSieve))) (_ : popIndex < size cell) :
pop_spec popIndex cell (pop popIndex cell).
Proof.  move: cell H. induction popIndex; simpl; intros. destruct cell.
clear -H. abstract(discriminate H).
exact: PopZero. destruct cell. abstract(discriminate H).
simpl. apply: PopSucc. apply: IHpopIndex. assumption.
(* funelim (pop popIndex cell). discriminate H. exact: PopZero.
simpl. simp pop. simpl. apply: PopSucc. apply: X. assumption. *) Defined.
Arguments popP  : clear implicits.

Lemma popP'  (cell: seq ('I_ (S topSieve)))   (popIndex: 'I_(size cell)) :  pop_spec
popIndex cell (pop popIndex cell).
 apply: popP. clear. apply: ltn_ord. (*  case: popIndex. intros; assumption. *)
Defined.
Arguments popP'  : clear implicits.

Lemma pop_spec_function (popIndex: nat)  (cell: seq ('I_ (S topSieve)))  (popCell: seq
('I_ (S topSieve)))
(ps : pop_spec popIndex cell popCell) : forall (popCell': seq ('I_ (S topSieve)))
(ps' : pop_spec popIndex cell popCell'),
popCell = popCell'.
Proof. induction ps; intros popCell'' ps'; dependent elimination ps'.
- reflexivity.
- congr ( _ :: _). apply: IHps; assumption.
Defined.

Lemma pop_spec_UIP (popIndex: nat)  (cell: seq ('I_ (S topSieve)))  (popCell: seq ('I_
(S topSieve)))
(ps ps_ : pop_spec popIndex cell popCell) : ps = ps_.
induction ps; dependent elimination ps_.
Proof.
- reflexivity.
-  congr (PopSucc _ _). apply: IHps.
Defined.

Lemma PopCommute Le Ge (Le_Ge : Le <= Ge) cell popLe_Cell popGe_popLe_Cell popGe_Cell :
pop_spec Le cell popLe_Cell -> pop_spec Ge popLe_Cell popGe_popLe_Cell ->
pop_spec (S Ge) cell popGe_Cell -> pop_spec Le popGe_Cell popGe_popLe_Cell.
Proof.
intros popLe_CellP popGe_popLe_CellP popGe_CellP'. move: Ge Le_Ge popGe_popLe_Cell
popGe_Cell popGe_popLe_CellP popGe_CellP'.
induction popLe_CellP; intros Ge Le_Ge popGe_popLe_Cell popGe_Cell popGe_popLe_CellP
popGe_CellP'.
{ dependent elimination popGe_CellP' as [@PopSucc _ _ popGe_Cell_tl _
popGe_Cell_tl_P'].
rewrite (pop_spec_function popGe_popLe_CellP popGe_Cell_tl_P'). exact: PopZero. }
{ dependent elimination popGe_popLe_CellP as [ PopZero _ _ | PopSucc _
popGe_popLe_Cell_tl_P].
  { clear -Le_Ge. abstract ( done).  }
  { dependent elimination popGe_CellP' as [PopSucc _ popGe_Cell_tl_P'].  apply:
PopSucc.
  apply: IHpopLe_CellP; eassumption. } }
Defined.

Inductive nerve: forall (dimCoef: nat)  (cell: seq ('I_ (S topSieve))), Type :=

| Diff_nerve: forall (dimCoef: nat)  (cell: seq ('I_ (S topSieve)))
  (self_structCoSheafO: structCoSheafO cell),
 forall (face: forall (popPos: nat) (popCell: seq ('I_ (S topSieve)))
          (popCellP: pop_spec popPos cell popCell),
```

```
          nerve dimCoef popCell ),
 nerve (S dimCoef) cell

| GlueDiff_nerve: forall (dimCoef: nat) (cell: seq ('I_ (S topSieve)))
  (self_structCoSheafO: structCoSheafO cell),
forall (coface: forall (pushVal: 'I_ (S topSieve)),
        nerve (dimCoef) (pushVal :: cell) ),
forall (face:  forall (popPos: nat)  (popCell: seq ('I_ (S topSieve)))
          (popCellP: pop_spec popPos  cell  popCell),
        nerve (dimCoef)  popCell ) ,
nerve (S dimCoef) cell

| Empty_nerve:
  nerve 0 [:: ].

Equations Derive Signature NoConfusion NoConfusionHom for nerve.

Lemma nerve_prop1 (dimCoef: nat) (cell: seq ('I_ (S topSieve)))
(ns: nerve dimCoef cell): size cell <= dimCoef.
Proof.  induction ns. destruct cell as [| cellHd cell]. reflexivity.
  move: (H _ _ (@popP 0 (cellHd::cell) is_true_true) ). simp pop.
 destruct cell as [| cellHd cell]. reflexivity. move: (H0 _ _ (@popP 0 (cellHd::cell)
is_true_true) ). simp pop. done.
Qed.

Definition dfinfun : forall (aT : finType) (rT : aT -> Type),
(forall x : aT, rT x) -> {dffun forall x : aT, rT x} := @FinfunDef.finfun.

Notation "[ 'dffun' x : aT => E ]" := (dfinfun (fun x : aT => E))
 (at level 0, x name) : fun_scope.

Section FinFunZmod.
Variable (aT : finType) (rT : aT -> zmodType).
Implicit Types f g : {dffun forall x : aT,  rT x}.

Definition ffun_zero := [dffun a : aT => (0 : rT a)].
Definition ffun_opp f := [dffun a : aT => - f a].
Definition ffun_add f g := [dffun a : aT => f a + g a].

Fact ffun_addA : associative ffun_add.
Proof. by move=> f1 f2 f3; apply/ffunP=> a; rewrite !ffunE addrA. Qed.
Fact ffun_addC : commutative ffun_add.
Proof. by move=> f1 f2; apply/ffunP=> a; rewrite !ffunE addrC. Qed.
Fact ffun_add0 : left_id ffun_zero ffun_add.
Proof. by move=> f; apply/ffunP=> a; rewrite !ffunE add0r. Qed.
Fact ffun_addN : left_inverse ffun_zero ffun_opp ffun_add.
Proof. by move=> f; apply/ffunP=> a; rewrite !ffunE addNr. Qed.

Definition ffun_zmodMixin :=
 Zmodule.Mixin ffun_addA ffun_addC ffun_add0 ffun_addN.
Canonical ffun_zmodType := Eval hnf in ZmodType {dffun forall x : aT, rT x}
ffun_zmodMixin.

Section Sum.
Variables (I : Type) (r : seq I) (P : pred I) (F : I -> {dffun forall x : aT,  rT x}).

Lemma sum_ffunE x : (\sum_(i <- r | P i) F i) x = \sum_(i <- r | P i) F i x.
Proof. by elim/big_rec2: _ => // [|i _ y _ <-]; rewrite !ffunE. Qed.

Lemma sum_ffun :
 \sum_(i <- r | P i) F i = [dffun x : _ => \sum_(i <- r | P i) F i x].
```

```
Proof. by apply/ffunP=> i; rewrite sum_ffunE ffunE. Qed.
End Sum.

Lemma ffunMnE f n x : (f *+ n) x = f x *+ n.
Proof. by rewrite -[n]card_ord -!sumr_const sum_ffunE. Qed.
End FinFunZmod.

Section FinFunLmod.
Variable (R : ringType) (aT : finType) (rT :  aT -> lmodType R).
Implicit Types f g : {dffun forall x : aT, rT x}.

Definition ffun_scale k f := [dffun a : aT => k *: f a].

Fact ffun_scaleA k1 k2 f :
 ffun_scale k1 (ffun_scale k2 f) = ffun_scale (k1 * k2) f.
Proof. by apply/ffunP=> a; rewrite !ffunE scalerA. Qed.
Fact ffun_scale1 : left_id 1 ffun_scale.
Proof. by move=> f; apply/ffunP=> a; rewrite !ffunE scale1r. Qed.
Fact ffun_scale_addr k : {morph (ffun_scale k) : x y / x + y}.
Proof. by move=> f g; apply/ffunP=> a; rewrite !ffunE scalerDr. Qed.
Fact ffun_scale_addl u : {morph (ffun_scale)^~ u : k1 k2 / k1 + k2}.
Proof. by move=> k1 k2; apply/ffunP=> a; rewrite !ffunE scalerDl. Qed.

Definition ffun_lmodMixin :=
 LmodMixin ffun_scaleA ffun_scale1 ffun_scale_addr ffun_scale_addl.
Canonical ffun_lmodType :=
 Eval hnf in LmodType R {dffun forall x : aT, rT x} ffun_lmodMixin.
End FinFunLmod.

Lemma ffun_scaleE (R : ringType) (aT : finType) (rT : aT -> lmodType R)
(f : forall x : aT, rT x) (a : R):
 (a *: [dffun x : aT => f x]) = [dffun x : aT => a *: f x].
Proof. apply: eq_dffun => x; rewrite ffunE; reflexivity. Qed.

Lemma sum_ffun_ffun (aT : finType) (rT : aT -> zmodType)
(I : Type) (r : seq I) (P : pred I)
(F : I -> forall x : aT, rT x) :
\sum_(i <- r | P i) [dffun x : aT => F i x] = [dffun x : aT => \sum_(i <- r | P i) F i
x].
Proof. rewrite sum_ffun. apply: eq_dffun => x. apply: eq_bigr => i _. rewrite ffunE.
reflexivity. Qed.

Lemma  big_ord_cast :
forall (R : Type) (idx : R) (op : R -> R -> R)
  (n1 n2 : nat) (F : 'I_n2 -> R) (Heq : n1 = n2),
let w := cast_ord Heq in
\big[op/idx]_(i < n2 ) F i = \big[op/idx]_(i < n1) F (w i).
intros. subst. apply: eq_bigr; simpl. intros i _. subst w.  rewrite cast_ord_id.
reflexivity. Qed.

Variable R : ringType.

Structure shfyCoef_struct := {
 shfyCoef :  forall (cell: seq ('I_ (S topSieve))), lmodType R ;

 glue_shfyCoef : forall (cell: seq ('I_ (S topSieve))),
 {linear {dffun forall (pushVal: 'I_ (S topSieve)), shfyCoef (pushVal :: cell)} ->
(shfyCoef cell)}%R ;

 restrict_shfyCoef : forall (cell: seq ('I_ (S topSieve))) (popPos: nat)
  (popCell: seq ('I_ (S topSieve))) (popCellP : pop_spec popPos  cell  popCell),
```

```
 {linear (shfyCoef (popCell)) -> (shfyCoef  cell) }%R ;

 glue_restrict_shfyCoef : forall (popCell: seq 'I_topSieve.+1)
 (popCell_ZeroP : forall pushVal : 'I_topSieve.+1, pop_spec 0 (pushVal :: popCell)
popCell )
 (ff_ : shfyCoef popCell),
   glue_shfyCoef _ [ffun pushVal : 'I_topSieve.+1 =>
      restrict_shfyCoef (popCell_ZeroP pushVal) ff_] = ff_ ;

 restrict_glue_shfyCoef : forall (cellHd: 'I_topSieve.+1) (cell: seq 'I_topSieve.+1)
 (popPos: nat) (popCell: seq 'I_topSieve.+1) (popCellP : pop_spec popPos (cellHd ::
cell) popCell)
 (popCell_SuccP : forall pushVal : 'I_topSieve.+1, pop_spec popPos.+1 [:: pushVal,
cellHd & cell] (pushVal :: popCell))
 (ff_ : forall pushVal : 'I_topSieve.+1, shfyCoef (pushVal :: popCell)),
   restrict_shfyCoef popCellP
     (glue_shfyCoef _ [ffun pushVal : 'I_topSieve.+1 => (ff_ pushVal)])
   = (glue_shfyCoef _ [ffun pushVal : 'I_topSieve.+1 =>
        restrict_shfyCoef (popCell_SuccP pushVal) (ff_ pushVal)]) ;

  restrict_functor_irrelevant_shfyCoef : forall (cellHd: 'I_topSieve.+1) (cell: seq
'I_topSieve.+1),
  forall (Le : nat) (Ge : nat) (Le_Ge : Le <=  Ge)
  (popLe_Cell:   seq 'I_topSieve.+1)
  (popLe_CellP:  pop_spec Le (cellHd :: cell) (popLe_Cell ))
  (popGe_popLe_Cell:  seq 'I_topSieve.+1)
  (popGe_popLe_CellP:   pop_spec Ge (popLe_Cell )  (popGe_popLe_Cell ))
  (popGe_Cell:  seq 'I_topSieve.+1)
  (popGe_CellP:  pop_spec (S Ge)  (cellHd :: cell)  (popGe_Cell ))
  (popLe_popGe_CellP:  pop_spec Le  (popGe_Cell )  (popGe_popLe_Cell ))
  (ff_: shfyCoef (popGe_popLe_Cell )),

    restrict_shfyCoef (popLe_CellP )
    (restrict_shfyCoef (popGe_popLe_CellP )
      (ff_ ))
  =  restrict_shfyCoef  (popGe_CellP )
      (restrict_shfyCoef (popLe_popGe_CellP )
        (ff_  ))  ;
}.

Arguments restrict_shfyCoef { _ } _  _ {_ } _.  Arguments glue_shfyCoef { _ _}.

Variable F_sh : shfyCoef_struct.

Definition diffGluing: forall (dimCoef: nat),
forall (ff_: (* forall (outerIndex: 'I_ (S topSieve)), *) forall (cell: seq ('I_ (S
topSieve))),
  nerve (dimCoef.-1) cell  -> shfyCoef F_sh  cell),
forall  (cell: seq ('I_ (S topSieve))),
  nerve ( dimCoef)  cell -> shfyCoef F_sh  cell.
Proof. intros ? ? ?  ns.   destruct ns.
{ (* Diff_nerve *)  refine (\sum_( popPos < size cell ) (-1) ^+ popPos *: _)%R.
  unshelve eapply (@restrict_shfyCoef _ _ popPos _ (popP' _ popPos)).
     apply: ff_. apply: (face _ _  (popP' _ popPos)). }
{ (* GlueDiff_nerve *)
  apply: GRing.add.
  { apply glue_shfyCoef.  refine [ffun  pushVal : 'I_topSieve.+1 => _ ].
     apply: ff_ . apply (coface pushVal). }
  { refine (\sum_( popPos < size cell ) (-1) ^+ popPos *: _)%R.
  unshelve eapply (@restrict_shfyCoef _ _ popPos _ (popP' _ popPos)).
    apply: ff_. apply: (face _ _  (popP' _ popPos)). }  }
```

```
{ (* Empty_nerve *) exact (ff_  _  Empty_nerve). }
Defined.

Definition isHomotopyEquivZero_nerve: forall (dimCoef: nat),
forall  (cellHd : 'I_ (S topSieve))  (cell: seq ('I_ (S topSieve)))

(self_structCoSheafO : structCoSheafO  (cellHd :: cell))
(coface_structCoSheafO : forall pushVal : 'I_topSieve.+1, structCoSheafO  (pushVal ::
(cellHd :: cell)))

(selfShifted_nerve: nerve (S dimCoef) (cellHd :: cell))
(cofaceOfFace_nerve: forall pushVal : 'I_topSieve.+1,  forall popPos:nat, forall
popCell,
pop_spec popPos (cellHd :: cell) popCell ->
nerve (S dimCoef) (pushVal :: popCell))

(face_structCoSheafO : forall popPos popCell ,  pop_spec popPos (cellHd :: cell)
popCell -> structCoSheafO popCell)
(faceOfFace_nerve: forall popPos popCell (popCellP : pop_spec popPos (cellHd :: cell)
popCell ),
 forall popPos0 popPopCell (popPopCellP : pop_spec popPos0 popCell popPopCell ) ,
 nerve (S dimCoef)  popPopCell )  ,

nerve (S (S (S dimCoef)))  (cellHd :: cell).
Proof.
intros. apply: GlueDiff_nerve.  exact: self_structCoSheafO.
{ intros. unshelve eapply Diff_nerve.
  exact: coface_structCoSheafO.

  { intros. destruct popCell as [| popCell_hd popCell_tl]. abstract (depelim popCellP).
  depelim popCellP. exact: selfShifted_nerve.
  apply (cofaceOfFace_nerve _ _ _  popCellP). } }

{  intros popPos popCell popCellP. unshelve  eapply GlueDiff_nerve.
exact: (face_structCoSheafO _ _ popCellP).
{  intros pushVal. eapply cofaceOfFace_nerve.  eassumption.    }

 { intros popPos0 popPopCell popPopCellP. exact: (faceOfFace_nerve _ _ popCellP _ _
popPopCellP).  } }
Defined.

Definition isHomotopyEquivZero:
 forall (dimCoef: nat),
 forall (ff_: forall  (cell: seq ('I_ (S topSieve))),
 nerve (S (S dimCoef)).-1  cell  -> shfyCoef F_sh  cell),

 forall  (cellHd : 'I_ (S topSieve))  (cell: seq ('I_ (S topSieve)))

 (self_structCoSheafO : structCoSheafO  (cellHd :: cell))
 (coface_structCoSheafO : forall pushVal : 'I_topSieve.+1, structCoSheafO  (pushVal ::
(cellHd :: cell)))

 (selfShifted_nerve: nerve (S dimCoef) (cellHd :: cell))
 (cofaceOfFace_nerve: forall pushVal popPos popCell,
      pop_spec popPos (cellHd :: cell) popCell ->
    nerve (S dimCoef) (pushVal :: popCell))

 (face_structCoSheafO : forall popPos popCell ,  pop_spec popPos (cellHd :: cell)
popCell -> structCoSheafO popCell)
```

```
 (faceOfFace_nerve: forall popPos popCell (popCellP : pop_spec popPos (cellHd :: cell)
popCell ),
      forall popPos0 popPopCell (popPopCellP : pop_spec popPos0 popCell popPopCell ) ,
      nerve (S dimCoef)  popPopCell )

  (hyp_nerve_irrelevant:
  forall (Le : nat) (Ge : nat) (Le_Ge : Le <=  Ge)
    (popLe_Cell:   seq 'I_topSieve.+1)
    (popLe_CellP:  pop_spec Le (cellHd :: cell) (popLe_Cell ))
    (popGe_popLe_Cell:  seq 'I_topSieve.+1)
    (popGe_popLe_CellP:   pop_spec Ge (popLe_Cell )  (popGe_popLe_Cell ))
    (popGe_Cell:  seq 'I_topSieve.+1)
    (popGe_CellP:  pop_spec (S Ge)  (cellHd :: cell)  (popGe_Cell ))
    (popLe_popGe_CellP:  pop_spec Le  (popGe_Cell )  (popGe_popLe_Cell )),
        @faceOfFace_nerve _ _ (popLe_CellP ) _ _ (popGe_popLe_CellP )
      =  @faceOfFace_nerve _ _  (popGe_CellP ) _ _ (popLe_popGe_CellP )),

 diffGluing (dimCoef := S (S (S dimCoef))) (diffGluing  ff_)
 (isHomotopyEquivZero_nerve self_structCoSheafO coface_structCoSheafO
selfShifted_nerve cofaceOfFace_nerve face_structCoSheafO  faceOfFace_nerve ) =
 ff_ _ selfShifted_nerve.
Proof. intros. simpl.
under eq_dffun => pushVal. { rewrite (bigD1_ord ord0 (@erefl _ true)). simpl.
unfold apply_noConfusion , DepElim.solution_left, DepElim.solution_right; simpl.
rewrite  scale1r.

under eq_bigr => popPos _. {
rewrite /bump /=. simpl. rewrite [X in (X *: _)%R]exprS -scalerA. over. }
simpl. rewrite -scaler_sumr.  over. }

under eq_dffun => pushVal. { simpl.
pose XR := fun pushVal => -1 *:
              (\sum_(i < (size cell).+1)
                  (-1) ^+ i *:
                  restrict_shfyCoef [:: pushVal, cellHd & cell]
                    (1 + i) (popP' [:: pushVal, cellHd & cell] (lift ord0 i))
                    (ff_ (pushVal :: pop i (cellHd :: cell))
                      (cofaceOfFace_nerve pushVal i (pop i (cellHd :: cell))
                          (popP i (cellHd :: cell) (ltn_ord (lift ord0 i)))))).
rewrite -[X in (_ + X)%R]/(XR pushVal).  rewrite -[XR pushVal]ffunE.
pose XL := fun pushVal => restrict_shfyCoef [:: pushVal, cellHd & cell] 0
               (popP' [:: pushVal, cellHd & cell] ord0)
               (ff_ (cellHd :: cell) selfShifted_nerve) .
rewrite -[X in (X + _)%R]/(XL pushVal). rewrite -[XL pushVal]ffunE.
subst XR XL. over. }
rewrite linearD.

rewrite glue_restrict_shfyCoef. rewrite -[LHS]addrA.
set X := (X in (_ + X = _)%R). suff : X = 0 by move => ->; exact: addr0. subst X.
rewrite -[X in glue_shfyCoef X]ffun_scaleE linearZZ scaleN1r.

under eq_bigr => popPos _.
{ rewrite linearD. rewrite scalerDr.
rewrite (restrict_glue_shfyCoef _ (fun pushVal : 'I_topSieve.+1 => PopSucc pushVal
(@popP' (cellHd :: cell) popPos) )).
rewrite -linearZZ. rewrite ffun_scaleE. over.  } simpl.
rewrite big_split /=. rewrite -linear_sum. rewrite sum_ffun_ffun. rewrite addrA.

set X := (X in (- glue_shfyCoef X + _ )%R). set Y := (Y in (- glue_shfyCoef X  +
glue_shfyCoef Y )%R).
have : X = Y; last (move ->; rewrite addNr add0r); subst X Y.
```

```
  apply: eq_dffun => pushVal. apply: eq_bigr => popPos _. congr (_ *: _).
  unfold popP'; simpl. have -> : (ltn_ord (lift ord0 popPos)) =  (ltn_ord popPos) by
exact: eq_irrelevance.
  reflexivity.

under eq_bigr => Le _. {  have @Heq : size cell = size (pop Le (cellHd :: cell)) by
clear; abstract (rewrite [size (pop _ _)]pred_Sn -(pop_spec_prop2 (popP' (cellHd::cell)
Le)) // ) .
rewrite (big_ord_cast _ _ _  Heq). subst Heq. simpl.

rewrite [(\sum_(_ < _) _)](bigID (fun Ge : 'I_(_) => Le <= Ge)) /=.
rewrite [X in _ + X](eq_bigl (fun i : 'I_(_) => i < Le)); last by intro i; rewrite
ltnNge //.
rewrite linearD scalerDr. do 2 rewrite linear_sum scaler_sumr.
under [X in X + _]eq_bigr => Ge _; first (rewrite linearZZ; over).
under [X in _ + X]eq_bigr => Ge _; first (rewrite linearZZ; over). simpl.
set Y := (Y in _ + \sum_(Ge < _ | _) (-1) ^+ Le *: Y Ge ).
rewrite [X in _ + X](eq_bigr (fun Ge => - ((-1) ^+ Le.-1 *: Y Ge)) ); last first.
intros Ge H.  rewrite -scaleN1r scalerA -exprS (@ltn_predK Ge) //. rewrite sumrN. subst
Y. simpl.  over. }

simpl. rewrite sumrB. clear -hyp_nerve_irrelevant.
set gg_ := (gg_ in \sum_(Le < _) \sum_(Ge < _ | _) _ *: ( _ *: restrict_shfyCoef _ _ _
(restrict_shfyCoef _ _ _ (gg_ Le Ge))  ) -
\sum_(Le < _) \sum_(Ge < _ | _) _ *: ( _ *: restrict_shfyCoef _ _ _ (restrict_shfyCoef
_ _ _ (gg_ Le Ge))  ) ).

(* diff_diff = 0 *)
intros. set LHS := (X in X - _ = _).  set RHS := (X in _ - X = _). suff ->: LHS = RHS
by exact: addrN. subst LHS RHS.
set LF' := (X in (\sum_(i < _) (\sum_(j < _ | _ ) (X i j)))); pose LF := LF'; subst
LF'.
set LP' := (X in (\sum_(i < _) (\sum_(j < _ | X i j ) _))); pose LP := LP'; subst LP'.
rewrite pair_big_dep /=.

under eq_bigl => p. { rewrite -[_ <= _](andb_idl (a:= p.1 < size cell)); cycle 1.
by move: (ltn_ord p.2); intro; move/leq_ltn_trans; exact. over. }

rewrite -(pair_big_dep (fun k : 'I_( (size cell) .+1) => k < size cell) LP LF) /=.
rewrite big_ord_narrow.
rewrite [in RHS](bigD1_ord ord0 (@erefl _ true)) /=. rewrite big_pred0_eq add0r. simpl.
rewrite [RHS](exchange_big_dep xpredT) /=; last reflexivity.
subst LF LP. simpl. apply: eq_bigr; move => /= Le _. apply: eq_bigr; move => /= Ge
Le_Ge.
do 2 rewrite scalerA -exprD. rewrite addnC. congr ( _ *: _)%R.

set popGe_CellP := (popGe_CellP in _ =  restrict_shfyCoef _ _ popGe_CellP
                   (restrict_shfyCoef _ _ _ _) ).
set popLe_popGe_CellP := (popLe_popGe_CellP in _ =  restrict_shfyCoef _ _ _
                   (restrict_shfyCoef _ _ popLe_popGe_CellP _) ).
pose Heq_pop_pop := Logic.eq_sym (pop_pop Le_Ge (cellHd::cell)).

unshelve erewrite restrict_functor_irrelevant_shfyCoef with
(1:=Le_Ge)
(popGe_CellP := popGe_CellP)
(popLe_popGe_CellP :=  rew dependent
  [fun x _ =>  pop_spec Le (cellHd :: pop Ge cell) x] Heq_pop_pop in
popLe_popGe_CellP ).
congr (restrict_shfyCoef _ _ _ _).

suff gg_commute: rew dependent  [fun x _ =>  shfyCoef _  x] Heq_pop_pop in
```

```
  (gg_ (lift ord0 Ge) Le) =  (gg_ (widen_ord (leqnSn (size cell)) Le) Ge) by
  rewrite -{}gg_commute;
  case: _ / Heq_pop_pop popLe_popGe_CellP; reflexivity.
subst gg_. simpl.

unshelve erewrite hyp_nerve_irrelevant  with
(1:=Le_Ge)
(popGe_CellP := popGe_CellP)
(popLe_popGe_CellP :=  rew dependent
  [fun x _ =>  pop_spec Le (cellHd :: pop Ge cell) x] Heq_pop_pop in
popLe_popGe_CellP ).
case: _ / Heq_pop_pop . reflexivity.
Qed.

End nerve.

Module example_1.
Require Import ZArith. Open Scope Z_scope. Section section.
Axiom exc : forall {T : Type}, T.
Let topSieve : nat := 2.
Let structCoSheafO : seq ('I_ (S topSieve)) -> bool := (fun _ => true).
Let U0 : 'I_ (S topSieve) := @Ordinal (S topSieve) (0%N) is_true_true.
Let U1 : 'I_ (S topSieve) := @Ordinal (S topSieve) (1%N) is_true_true.
Let U01 (* U2 *): 'I_ (S topSieve) := @Ordinal (S topSieve) (2%N) is_true_true.

Let ZU0 := Z. Let ZU1 := Z. Let ZU01 := Z. Opaque ZU0 ZU1 ZU01.
Parameter re_U0_U01 : ZU0 -> ZU01. Coercion re_U0_U01 : ZU0 >-> ZU01.
Parameter re_U1_U01 : ZU1 -> ZU01. Coercion re_U1_U01 : ZU1 >-> ZU01.

Let shfyCoef_F (cell: seq ('I_ (S topSieve))) : Type:=
  if perm_eq cell  [:: ] then
       (ZU0 * ZU1 * ZU01)%type
  else if perm_eq cell [:: U0] || perm_eq cell [:: U0; U0]  then
       (ZU0 * ZU01)%type
  else if perm_eq cell [:: U1] ||  perm_eq cell  [:: U1; U1] then
       (ZU1 * ZU01)%type
  else if perm_eq cell [:: U01 ]
         || perm_eq cell [:: U0; U1 ] || perm_eq cell [:: U0; U01 ]
         || perm_eq cell [:: U1; U01 ]
         || perm_eq cell [:: U01; U01 ]  then
      (ZU01)%type
  else exc .
Eval compute in  (shfyCoef_F [:: U0]).

Hypothesis restrict_shfyCoef_F : forall (cell: seq ('I_ (S topSieve))) (popPos: nat)
  (popCell: seq ('I_ (S topSieve))) (popCellP : pop_spec popPos  cell  popCell),
 (shfyCoef_F (popCell)) -> (shfyCoef_F  cell) .

Hypothesis rEq_u_u0 : forall x (f_u0 : ZU0) (f_u1 : ZU1) (f_u01 : ZU01),
@restrict_shfyCoef_F [:: U0 ] 0 [:: ] x (f_u0, f_u1, f_u01) = (f_u0, re_U1_U01 f_u1 +
f_u01 ).
Hypothesis rEq_u_u1 : forall x f_u0 f_u1 f_u01, @restrict_shfyCoef_F [:: U1 ] 0 [:: ] x
(f_u0, f_u1, f_u01) = (f_u1, re_U0_U01 f_u0 + f_u01 ).
Hypothesis rEq_u_u01 : forall x f_u0 f_u1 f_u01, @restrict_shfyCoef_F [:: U01 ] 0 [:: ]
x (f_u0, f_u1, f_u01) = (re_U0_U01 f_u0 + re_U1_U01 f_u1 + f_u01 ).

Hypothesis rEq_u0_u0u0 : forall x f_u0 f_u01, @restrict_shfyCoef_F [:: U0; U0 ] 0 [::
U0] x (f_u0, f_u01) = (f_u0, f_u01 ).
Hypothesis rEq_u0_u1u0 : forall x f_u0 f_u01, @restrict_shfyCoef_F [:: U1; U0 ] 0 [::
U0] x (f_u0, f_u01) = (re_U0_U01 f_u0 + f_u01 ).
```

```
Hypothesis rEq_u0_u01u0 : forall x f_u0 f_u01, @restrict_shfyCoef_F [:: U01; U0 ] 0 [::
U0] x (f_u0, f_u01) = (re_U0_U01 f_u0 + f_u01 ).

Hypothesis rEq_u0_u0u0' : forall x f_u0 f_u01, @restrict_shfyCoef_F [:: U0; U0 ] 1 [::
U0] x (f_u0, f_u01) = (f_u0, f_u01 ).
Hypothesis rEq_u1_u1u0 : forall x f_u1 f_u01, @restrict_shfyCoef_F [:: U1; U0 ] 1 [::
U1] x (f_u1, f_u01) = (re_U1_U01 f_u1 + f_u01 ).
Hypothesis rEq_u01_u01u0 : forall x f_u01, @restrict_shfyCoef_F [:: U01; U0 ] 1 [::
U01] x (f_u01) = (f_u01 ).

Hypothesis glue_shfyCoef_F : forall (cell: seq ('I_ (S topSieve))),
 (forall (pushVal: 'I_ (S topSieve)), shfyCoef_F (pushVal :: cell)) -> (shfyCoef_F
cell).

Hypothesis gEq_u : forall f_, @glue_shfyCoef_F [:: ] f_ = ( (f_ U0).1 , (f_ U1).1 , (f_
U0).2 + (f_ U1).2 - (f_ U01) ).

Hypothesis gEq_u0 : forall f_, @glue_shfyCoef_F [:: U0] f_ = ( (f_ U0).1 , (f_ U0).2 +
(f_ U1) - (f_ U01) ).

Arguments restrict_shfyCoef_F _  _ { _ } _. Arguments glue_shfyCoef_F : clear
implicits.

Lemma gr_1
(popCell_ZeroP : forall pushVal : 'I_topSieve.+1, pop_spec 0 (pushVal :: [:: ]) [:: ] )
(f_ : shfyCoef_F [:: ]) :
glue_shfyCoef_F [::]   (fun pushVal : 'I_topSieve.+1 =>
  restrict_shfyCoef_F [:: pushVal] 0 (popCell_ZeroP pushVal) f_) = f_ .
rewrite gEq_u. destruct f_ as [[f_u0 f_u1] f_u01]. rewrite rEq_u_u0 rEq_u_u1 rEq_u_u01.
simpl.
congr ( _ , _ , _).  rewrite -?[ZU0]/Z -?[ZU1]/Z -?[ZU01]/Z.  ring. Qed.

Lemma gr_2
(popCell_ZeroP : forall pushVal : 'I_topSieve.+1, pop_spec 0 (pushVal :: [:: U0]) [::
U0] )
(f_ : shfyCoef_F [:: U0]) :
glue_shfyCoef_F [:: U0] (fun pushVal : 'I_topSieve.+1 =>
  restrict_shfyCoef_F [:: pushVal; U0] 0 (popCell_ZeroP pushVal) f_) = f_ .
rewrite gEq_u0. destruct f_ as [f_u0 f_u01]. simpl.
rewrite ?rEq_u0_u0u0 ?rEq_u0_u1u0 ?rEq_u0_u01u0. simpl.
congr ( _ , _). rewrite -?[ZU0]/Z -?[ZU1]/Z -?[ZU01]/Z.  ring. Qed.

Lemma rg_1  (popCellP : pop_spec (0)%N (U0 :: [:: ]) [:: ])
(popCell_SuccP : forall pushVal : 'I_topSieve.+1, pop_spec (0.+1)%N [:: pushVal, U0 &
[:: ]] (pushVal :: [:: ]))
(ff_ : forall pushVal : 'I_topSieve.+1, shfyCoef_F (pushVal :: [:: ])):
restrict_shfyCoef_F [:: U0] 0 popCellP
  (glue_shfyCoef_F [::] (fun pushVal : 'I_topSieve.+1 => ff_ pushVal)) =
glue_shfyCoef_F [:: U0] (fun pushVal : 'I_topSieve.+1 =>
  restrict_shfyCoef_F [:: pushVal; U0] 1 (popCell_SuccP pushVal)
    (ff_ pushVal)).
rewrite gEq_u0.  rewrite gEq_u.  rewrite ?rEq_u_u0 ?rEq_u_u1 ?rEq_u_u01.
rewrite ?rEq_u0_u0u0 ?rEq_u0_u1u0 ?rEq_u0_u01u0.
destruct (ff_ U0) as [ffU0_u0 ffU0_u01]. destruct (ff_ U1) as [ffU1_u1 ffU1_u01].
rewrite ?rEq_u0_u0u0' ?rEq_u1_u1u0 ?rEq_u01_u01u0. simpl.
congr ( _, _). rewrite -?[ZU0]/Z -?[ZU1]/Z -?[ZU01]/Z.  ring. Qed.

Lemma rr_1 (popLe_CellP:  pop_spec 0 (U1 :: [:: U0]) [:: U0])
  (popGe_popLe_CellP:   pop_spec 0 [:: U0]  [:: ])
  (popGe_CellP:  pop_spec (S 0)  (U1 :: [:: U0]) [:: U1 ])
  (popLe_popGe_CellP:  pop_spec 0  [:: U1 ]  [:: ])
```

```
  (f_: shfyCoef_F [:: ]):
  restrict_shfyCoef_F [:: U1; U0] 0 popLe_CellP
  (restrict_shfyCoef_F [:: U0] 0 popGe_popLe_CellP f_) =
restrict_shfyCoef_F [:: U1; U0] 1 popGe_CellP
  (restrict_shfyCoef_F [:: U1] 0 popLe_popGe_CellP f_).
destruct f_ as [[f_u0 f_u1] f_u01]. simpl. rewrite ?rEq_u_u0 ?rEq_u0_u1u0.
rewrite ?rEq_u_u1 ?rEq_u1_u1u0.
rewrite -[shfyCoef_F _]/Z.
 rewrite -?[ZU0]/Z -?[ZU1]/Z -?[ZU01]/Z.  ring. Qed.
End section. End example_1.


Module todo_circle_couterexample_2.
Require Import ZArith. Open Scope Z_scope. Section section.
Axiom exc : forall {T : Type}, T.
Let topSieve : nat := 6.
Let structCoSheafO : seq ('I_ (S topSieve)) -> bool := (fun _ => true (* circle: U012
=> false *)).
Let U0 : 'I_ (S topSieve) := @Ordinal (S topSieve) (0%N) is_true_true.
Let U1 : 'I_ (S topSieve) := @Ordinal (S topSieve) (1%N) is_true_true.
Let U2 : 'I_ (S topSieve) := @Ordinal (S topSieve) (2%N) is_true_true.
Let U01 (* U3 *): 'I_ (S topSieve) := @Ordinal (S topSieve) (3%N) is_true_true.
Let U02 (* U4 *): 'I_ (S topSieve) := @Ordinal (S topSieve) (4%N) is_true_true.
Let U12 (* U5 *): 'I_ (S topSieve) := @Ordinal (S topSieve) (5%N) is_true_true.
Let U012 (* U6 *): 'I_ (S topSieve) := @Ordinal (S topSieve) (6%N) is_true_true.

Let ZU0 := Z. Let ZU1 := Z. Let ZU2 := Z.
Let ZU01 := Z. Let ZU02 := Z. Let ZU12 := Z.
Let ZU012 := Z. Opaque ZU0 ZU1 ZU2 ZU01 ZU02 ZU12 ZU012.
Parameter re_U0_U01 : ZU0 -> ZU01.
Parameter re_U1_U01 : ZU1 -> ZU01.
Parameter re_U0_U02 : ZU0 -> ZU02.
Parameter re_U2_U02 : ZU2 -> ZU02.
Parameter re_U1_U12 : ZU1 -> ZU12.
Parameter re_U2_U12 : ZU2 -> ZU12.
Parameter re_U01_U012 : ZU01 -> ZU012.
Parameter re_U02_U012 : ZU02 -> ZU012.
Parameter re_U12_U012 : ZU12 -> ZU012.
Coercion re_U0_U01 : ZU0 >-> ZU01.
Coercion re_U1_U01 : ZU1 >-> ZU01.
Coercion re_U0_U02 : ZU0 >-> ZU02.
Coercion re_U2_U02 : ZU2 >-> ZU02.
Coercion re_U1_U12 : ZU1 >-> ZU12.
Coercion re_U2_U12 : ZU2 >-> ZU12.
Coercion re_U01_U012 : ZU01 >-> ZU012.
Coercion re_U02_U012 : ZU02 >-> ZU012.
Coercion re_U12_U012 : ZU12 >-> ZU012.

Parameter morphism_add_re_U12_U012 : forall f g, re_U12_U012 (f + g) = re_U12_U012 f +
re_U12_U012 g.
Parameter morphism_opp_re_U12_U012 : forall f, re_U12_U012 (- f) = - re_U12_U012 f.

Parameter morphism_add_re_U02_U012 : forall f g, re_U02_U012 (f + g) = re_U02_U012 f +
re_U02_U012 g.

Parameter functor_re_U2_U02_U012_v_U2_U12_U012 : forall f, re_U02_U012 (re_U2_U02 f) =
re_U12_U012 (re_U2_U12 f).

Let shfyCoef_F (cell: seq ('I_ (S topSieve))) : Type :=
  if perm_eq cell  [:: ] then
       (ZU0 * ZU1 * ZU2 * ZU01 * ZU02 * ZU12 * ZU012)%type
```

```
  else if perm_eq cell [:: U0] || perm_eq cell [:: U0; U0]  then
       (ZU0 * ZU01 * ZU02 * ZU012)%type
  else if perm_eq cell [:: U1] ||  perm_eq cell [:: U1; U1] then
       (ZU1 * ZU01 * ZU12 * ZU012)%type
  else if perm_eq cell [:: U2] ||  perm_eq cell [:: U2; U2] then
       (ZU2 * ZU02 * ZU12 * ZU012)%type
  else if perm_eq cell [:: U01] ||  perm_eq cell [:: U0 ; U1] ||  perm_eq cell [:: U0 ;
U01] || perm_eq cell [:: U1 ; U01] then
       (ZU01 * ZU012)%type
  else if perm_eq cell [:: U02] ||  perm_eq cell [:: U0 ; U2] ||  perm_eq cell [:: U0 ;
U02] || perm_eq cell [:: U2 ; U02] then
       (ZU02 * ZU012)%type
  else if perm_eq cell [:: U12] ||  perm_eq cell [:: U1 ; U2] ||  perm_eq cell [:: U1 ;
U12] || perm_eq cell [:: U2 ; U12] then
       (ZU12 * ZU012)%type
  else if perm_eq cell [:: U012 ]
         || perm_eq cell [:: U0; U12 ] || perm_eq cell [:: U0; U012 ]
         || perm_eq cell [:: U1; U02 ] || perm_eq cell [:: U1; U012 ]
         || perm_eq cell [:: U2; U01 ] || perm_eq cell [:: U2; U012 ]
         || perm_eq cell [:: U012; U012 ]
         || perm_eq cell [:: U0; U1; U2 ] then
      (ZU012)%type
  else exc .
Eval compute in  (shfyCoef_F [:: U0]).

Hypothesis restrict_shfyCoef_F : forall (cell: seq ('I_ (S topSieve))) (popPos: nat)
  (popCell: seq ('I_ (S topSieve))) (popCellP : pop_spec popPos  cell  popCell),
 (shfyCoef_F (popCell)) -> (shfyCoef_F  cell) .

Hypothesis rEq_u_u0 : forall x f_u0 f_u1 f_u2 f_u01 f_u02 f_u12 f_u012,
@restrict_shfyCoef_F [:: U0 ] 0 [:: ] x (f_u0, f_u1, f_u2, f_u01, f_u02, f_u12, f_u012)
    = (f_u0, re_U1_U01 f_u1 + f_u01, re_U2_U02 f_u2 + f_u02, re_U12_U012 f_u12 +
f_u012).
Hypothesis rEq_u_u1 : forall x f_u0 f_u1 f_u2 f_u01 f_u02 f_u12 f_u012,
@restrict_shfyCoef_F [:: U1 ] 0 [:: ] x (f_u0, f_u1, f_u2, f_u01, f_u02, f_u12, f_u012)
    = (f_u1, re_U0_U01 f_u0 + f_u01, re_U2_U12 f_u2 + f_u12, re_U02_U012 f_u02 +
f_u012).
Hypothesis rEq_u_u2 : forall x f_u0 f_u1 f_u2 f_u01 f_u02 f_u12 f_u012,
@restrict_shfyCoef_F [:: U2 ] 0 [:: ] x (f_u0, f_u1, f_u2, f_u01, f_u02, f_u12, f_u012)
    = (f_u2, re_U0_U02 f_u0 + f_u02, re_U1_U12 f_u1 + f_u12, re_U01_U012 f_u01 +
f_u012).
Hypothesis rEq_u_u01 : forall x f_u0 f_u1 f_u2 f_u01 f_u02 f_u12 f_u012,
@restrict_shfyCoef_F [:: U01 ] 0 [:: ] x (f_u0, f_u1, f_u2, f_u01, f_u02, f_u12,
f_u012)
    = (re_U0_U01 f_u0 + re_U1_U01 f_u1 + f_u01, re_U02_U012 f_u02 + re_U12_U012 f_u12 +
f_u012) .
Hypothesis rEq_u_u02 : forall x f_u0 f_u1 f_u2 f_u01 f_u02 f_u12 f_u012,
@restrict_shfyCoef_F [:: U02 ] 0 [:: ] x (f_u0, f_u1, f_u2, f_u01, f_u02, f_u12,
f_u012)
    = (re_U0_U02 f_u0 + re_U2_U02 f_u2 + f_u02, re_U01_U012 f_u01 + re_U12_U012 f_u12 +
f_u012)  .
Hypothesis rEq_u_u12 : forall x f_u0 f_u1 f_u2 f_u01 f_u02 f_u12 f_u012,
@restrict_shfyCoef_F [:: U12 ] 0 [:: ] x (f_u0, f_u1, f_u2, f_u01, f_u02, f_u12,
f_u012)
    = (re_U1_U12 f_u1 + re_U2_U12 f_u2 + f_u12, re_U01_U012 f_u01 + re_U02_U012 f_u02 +
f_u012)  .
Hypothesis rEq_u_u012 : forall x f_u0 f_u1 f_u2 f_u01 f_u02 f_u12 f_u012,
@restrict_shfyCoef_F [:: U012 ] 0 [:: ] x (f_u0, f_u1, f_u2, f_u01, f_u02, f_u12,
f_u012)
    = (re_U01_U012 f_u01 + re_U02_U012 f_u02 + re_U12_U012 f_u12 + f_u012) .
```

```
Hypothesis rEq_u0_u0u0 : forall x f_u0 f_u01 f_u02 f_u012, @restrict_shfyCoef_F [:: U0;
U0 ] 0 [:: U0 ] x (f_u0, f_u01, f_u02, f_u012)
    = (f_u0, f_u01, f_u02, f_u012).
Hypothesis rEq_u0_u1u0 : forall x f_u0 f_u01 f_u02 f_u012, @restrict_shfyCoef_F [:: U1;
U0 ] 0 [:: U0 ] x (f_u0, f_u01, f_u02, f_u012)
    = (re_U0_U01 f_u0 + f_u01, re_U02_U012 f_u02 + f_u012).
Hypothesis rEq_u0_u2u0 : forall x f_u0 f_u01 f_u02 f_u012, @restrict_shfyCoef_F [:: U2;
U0 ] 0 [:: U0 ] x (f_u0, f_u01, f_u02, f_u012)
    = (re_U0_U02 f_u0 + f_u02, re_U01_U012 f_u01 + f_u012).
Hypothesis rEq_u0_u01u0 : forall x f_u0 f_u01 f_u02 f_u012, @restrict_shfyCoef_F [::
U01; U0 ] 0 [:: U0 ] x (f_u0, f_u01, f_u02, f_u012)
    = (re_U0_U01 f_u0 + f_u01, re_U02_U012 f_u02 + f_u012) .
Hypothesis rEq_u0_u02u0 : forall x f_u0 f_u01 f_u02 f_u012, @restrict_shfyCoef_F [::
U02; U0 ] 0 [:: U0 ] x (f_u0, f_u01, f_u02, f_u012)
    = (re_U0_U02 f_u0 + f_u02, re_U01_U012 f_u01 + f_u012)  .
Hypothesis rEq_u0_u12u0 : forall x f_u0 f_u01 f_u02 f_u012, @restrict_shfyCoef_F [::
U12; U0 ] 0 [:: U0 ] x (f_u0, f_u01, f_u02, f_u012)
    = (re_U01_U012 f_u01 + re_U02_U012 f_u02 + f_u012)  .
Hypothesis rEq_u0_u012u0 : forall x f_u0 f_u01 f_u02 f_u012, @restrict_shfyCoef_F [::
U012; U0 ] 0 [:: U0 ] x (f_u0, f_u01, f_u02, f_u012)
    = (re_U01_U012 f_u01 + re_U02_U012 f_u02 + f_u012) .

Hypothesis rEq_u0_u0u0' : forall x f_u0 f_u01 f_u02 f_u012, @restrict_shfyCoef_F [::
U0; U0 ] 1 [:: U0 ] x (f_u0, f_u01, f_u02, f_u012)
    = (f_u0, f_u01, f_u02, f_u012).
Hypothesis rEq_u1_u1u0 : forall x f_u1 f_u01 f_u12 f_u012, @restrict_shfyCoef_F [:: U1;
U0 ] 1 [:: U1 ] x (f_u1, f_u01, f_u12, f_u012)
    = (re_U1_U01 f_u1 + f_u01, re_U12_U012 f_u12 + f_u012).
Hypothesis rEq_u2_u2u0 : forall x f_u2 f_u02 f_u12 f_u012, @restrict_shfyCoef_F [:: U2;
U0 ] 1 [:: U2 ] x (f_u2, f_u02, f_u12, f_u012)
    = (re_U2_U02 f_u2 + f_u02, re_U12_U012 f_u12 + f_u012).
Hypothesis rEq_u01_u01u0 : forall x f_u01 f_u012, @restrict_shfyCoef_F [:: U01; U0 ] 1
[:: U01 ] x (f_u01, f_u012)
    = (f_u01, f_u012) .
Hypothesis rEq_u02_u02u0 : forall x f_u02 f_u012, @restrict_shfyCoef_F [:: U02; U0 ] 1
[:: U02 ] x (f_u02, f_u012)
    = (f_u02, f_u012)  .
Hypothesis rEq_u12_u12u0 : forall x f_u12 f_u012, @restrict_shfyCoef_F [:: U12; U0 ] 1
[:: U12 ] x (f_u12, f_u012)
    = (re_U12_U012 f_u12 + f_u012)  .
Hypothesis rEq_u012_u012u0 : forall x f_u012, @restrict_shfyCoef_F [:: U012; U0 ] 1 [::
U012 ] x (f_u012)
    = (f_u012) .

Hypothesis glue_shfyCoef_F : forall (cell: seq ('I_ (S topSieve))),
 (forall (pushVal: 'I_ (S topSieve)), shfyCoef_F (pushVal :: cell)) -> (shfyCoef_F
cell).
Local Notation " x ...1" := (fst (fst (fst x))) (at level 2).
Local Notation " x ...2" := (snd (fst (fst x))) (at level 2).
Local Notation " x ...3" := (snd ( (fst x))) (at level 2).
Local Notation " x ...4" := (snd ( ( x)))  (at level 2).
Hypothesis gEq_u : forall f_, @glue_shfyCoef_F [:: ] f_ =
( (f_ U0)...1 , (f_ U1)...1 , (f_ U2)...1 ,
  (f_ U0)...2 + (f_ U1)...2 - (f_ U01).1 , (f_ U0)...3 + (f_ U2)...2 - (f_ U02).1 , (f_
U1)...3 + (f_ U2)...3 - (f_ U12).1 ,
  (f_ U0)...4 + (f_ U1)...4 + (f_ U2)...4 - (f_ U01).2 - (f_ U02).2 - (f_ U12).2 + (f_
U012)   ).

Hypothesis gEq_u0 : forall f_, @glue_shfyCoef_F [:: U0 ] f_ =
( (f_ U0)...1 ,
```

```
  (f_ U0)...2 + (f_ U1).1 - (f_ U01).1 , (f_ U0)...3 + (f_ U2).1 - (f_ U02).1 ,
  (f_ U0)...4 + (f_ U1).2 + (f_ U2).2 - (f_ U01).2 - (f_ U02).2 - (f_ U12) + (f_
U012)   ).

Arguments restrict_shfyCoef_F _  _ { _ } _. Arguments glue_shfyCoef_F : clear
implicits.

Lemma gr_1
(popCell_ZeroP : forall pushVal : 'I_topSieve.+1, pop_spec 0 (pushVal :: [:: ]) [:: ] )
(f_ : shfyCoef_F [:: ]) :
glue_shfyCoef_F [::]   (fun pushVal : 'I_topSieve.+1 =>
  restrict_shfyCoef_F [:: pushVal] 0 (popCell_ZeroP pushVal) f_) = f_ .
rewrite gEq_u. simpl. refine (let: (f_u0, f_u1, f_u2, f_u01, f_u02, f_u12, f_u012) :=
f_ in _).
 rewrite rEq_u_u0 rEq_u_u1 rEq_u_u2 rEq_u_u01 rEq_u_u02 rEq_u_u12 rEq_u_u012. simpl.
 congr ( _ , _ , _ , _ , _ , _ , _); rewrite -?[ZU0]/Z -?[ZU1]/Z -?[ZU2]/Z -?[ZU01]/Z
-?[ZU02]/Z -?[ZU12]/Z -?[ZU012]/Z.
  ring. ring.  ring. Set Printing Coercions. ring. Qed.

Lemma gr_2
(popCell_ZeroP : forall pushVal : 'I_topSieve.+1, pop_spec 0 (pushVal :: [:: U0]) [::
U0] )
(f_ : shfyCoef_F [:: U0]) :
glue_shfyCoef_F [:: U0] (fun pushVal : 'I_topSieve.+1 =>
  restrict_shfyCoef_F [:: pushVal; U0] 0 (popCell_ZeroP pushVal) f_) = f_ .
rewrite gEq_u0. refine (let: (f_u0, f_u01, f_u02, f_u012) := f_ in _).
 rewrite rEq_u0_u0u0 rEq_u0_u1u0 rEq_u0_u2u0 rEq_u0_u01u0 rEq_u0_u02u0 rEq_u0_u12u0
rEq_u0_u012u0. simpl.
congr ( _ , _ , _ , _); rewrite -?[ZU0]/Z -?[ZU1]/Z -?[ZU2]/Z -?[ZU01]/Z -?[ZU02]/Z
-?[ZU12]/Z -?[ZU012]/Z.
 ring. ring.  ring. Qed.

Lemma rg_1  (popCellP : pop_spec (0)%N (U0 :: [:: ]) [:: ])
(popCell_SuccP : forall pushVal : 'I_topSieve.+1, pop_spec (0.+1)%N [:: pushVal, U0 &
[:: ]] (pushVal :: [:: ]))
(ff_ : forall pushVal : 'I_topSieve.+1, shfyCoef_F (pushVal :: [:: ])):
restrict_shfyCoef_F [:: U0] 0 popCellP
  (glue_shfyCoef_F [::] (fun pushVal : 'I_topSieve.+1 => ff_ pushVal)) =
glue_shfyCoef_F [:: U0] (fun pushVal : 'I_topSieve.+1 =>
  restrict_shfyCoef_F [:: pushVal; U0] 1 (popCell_SuccP pushVal)
    (ff_ pushVal)).
rewrite gEq_u0.  rewrite gEq_u. rewrite rEq_u_u0.
refine (let: (fU0_u0, fU0_u01, fU0_u02, fU0_u012) := (ff_ U0) in _).
refine (let: (fU1_u1, fU1_u01, fU1_u12, fU1_u012)  := (ff_ U1) in _).
refine (let: (fU2_u2, fU2_u02, fU2_u12, fU2_u012)  := (ff_ U2) in _).
refine (let: (fU01_u01, fU01_u012)  := (ff_ U01) in _).
refine (let: (fU02_u02, fU02_u012)   := (ff_ U02) in _).
refine (let: (fU12_u12, fU12_u012)   := (ff_ U12) in _).

rewrite rEq_u0_u0u0' rEq_u1_u1u0 rEq_u2_u2u0 rEq_u01_u01u0 rEq_u02_u02u0 rEq_u12_u12u0
rEq_u012_u012u0.
simpl. congr ( _, _, _, _); rewrite -?[ZU0]/Z -?[ZU1]/Z -?[ZU2]/Z -?[ZU01]/Z -?[ZU02]/Z
-?[ZU12]/Z -?[ZU012]/Z.
 ring. ring.
 do 2 rewrite morphism_add_re_U12_U012. rewrite morphism_opp_re_U12_U012. ring. Qed.

Lemma rr_1 (popLe_CellP:  pop_spec 0 (U1 :: [:: U0]) [:: U0])
  (popGe_popLe_CellP:   pop_spec 0 [:: U0]  [:: ])
  (popGe_CellP:  pop_spec (S 0)  (U1 :: [:: U0]) [:: U1 ])
  (popLe_popGe_CellP:  pop_spec 0  [:: U1 ]  [:: ])
  (f_: shfyCoef_F [:: ]):
```

```
  restrict_shfyCoef_F [:: U1; U0] 0 popLe_CellP
  (restrict_shfyCoef_F [:: U0] 0 popGe_popLe_CellP f_) =
restrict_shfyCoef_F [:: U1; U0] 1 popGe_CellP
  (restrict_shfyCoef_F [:: U1] 0 popLe_popGe_CellP f_).
Proof. refine (let: (f_u0, f_u1, f_u2, f_u01, f_u02, f_u12, f_u012) := f_ in _).
rewrite rEq_u_u0 rEq_u0_u1u0. rewrite rEq_u_u1 rEq_u1_u1u0.
congr ( _, _); rewrite -?[ZU0]/Z -?[ZU1]/Z -?[ZU2]/Z -?[ZU01]/Z -?[ZU02]/Z -?[ZU12]/Z
-?[ZU012]/Z.
ring.
rewrite morphism_add_re_U12_U012 morphism_add_re_U02_U012. rewrite
functor_re_U2_U02_U012_v_U2_U12_U012. ring.  Qed.

End section. End todo_circle_couterexample_2.
End Example.

(* Voila *)
```

C5 / coq