Proof-assistants, sheaves and applications

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 H¹(Δ²) = 0, H¹(S¹) ≠ 0.

.

# 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

can be formulated as an “*antecedental transformation*”

with primitive name “” in the language, or can be formulated as a “*consequential transformation*”

with primitive name “” in the language. And in the special case when is the counit of an adjunction with the functor left adjoint to and with absent (), then these various formulations allow for the elimination of the composition (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 () such as pairs of composable arrows (via the tensor profunctor-hom ), or functions on arrows (via the cotensor profunctor-hom )), or pairs of parallel arrows (via the product profunctor-hom ), or square of arrows (via the comma of the profunctor-hom ), 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

where is some cover of the space and restricts any singular cochain (function) defined on all simplices to only the small simplices contained within any , then the barycentric subdivision subordinate to ensures that is some homotopy equivalence and therefore, at the filtered/inductive colimit over the refinements of , 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:

gives the gluing operation

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

In practice, the implementation of the topology would be as some categorial site in the form of some closure operator where is the classifier of (sub-)objects (sieves) of the object , and where is the (opaque) set of witnesses that the pullback-sieve is covering (remember that the truthness that is covering is expressed as , iff ). But it is better to consider any presheaf in the slice over , 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 Δ² 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 S¹ specifies some hole in the nerve of this 2-simplex Δ², thus breaking this acyclicity H¹(S¹) ≠ 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.

U₁

U₂

U₃

U₁’

U₂’

U₁’’

U₁’

f: F(V₃)

f∩U₁’’

⋮

⋯

**Diagram 1.** Each nested basic sieve is some refinement of the fixed cover . 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 be a category. Let be a family of morphisms with fixed target such that all fibre products exist in . Consider the chain complex of abelian presheaves

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

is given by times the canonical map. Then there is an isomorphism

functorial in

Note that any of the products 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 be a presheaf of rings on . The chain complex

is exact in positive degrees

Oneself could dualize any co-sheaf through the complex of the elementary projective sheaves (instead of the generators)

with the boundary maps

Note that this resulting complex would be the same as the linear dual of the 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

where the right-hand side indicates the derived functor

of the left exact functor

**Lemma 03F7.** Let any abelian sheaf . Assume that for all , all and all . Then

And both lemmas rely on the technique of moving into the total complex of some double complex such as or the Cartan-Eilenberg resolution (Lemma 015I) for some injective resolution . Anyway, to sense the idea, remember that for any acyclic resolution , the long exact sequence allows to move through the double complex such as:

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.

U₁

f₂: F(U₂)

f₂∩U₁ - f₁’∩U₂

U₁

U₂

U₂

U₁

f₁: F(U₁)

U

U

U₁∩U₂

f₂’: F(U₂)

U₂

U₁

f₁’: F(U₁)

U

U₂

**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:

**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 \*) : …

# 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.

Click or tap here to enter text.

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.

(\*\* # #

#+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 \*)