Maclane pentagon is some comonadic descent (Rough Proof) (6 Pages)

Christopher Mary
EGITOR.NET, https://github.com/mozert

Abstract

This Coq text responds to Gross *Coq Categories Experience* and Chlipala *Compositional Computational Reflection* of ITP 2014. “Compositional” is synonymous for functional/functorial; “Computational Reflection” is synonymous for monadic semantics; and this text attempts some comonadic descent along functorial semantics : Dosen semiassociative coherence covers Maclane associative coherence by some comonadic adjunction, embedding : SemiAssoc $\leftrightarrows$ Assoc : flattening reflection.

Keywords : Logical Coherence • Categorical Descent • Coq Dedukti Modulo

1 Contents

This Coq text responds to Gross *Coq Categories Experience* [1] and Chlipala *Compositional Computational Reflection* [2] of ITP 2014. “Compositional” is synonymous for functional/functorial; “Computational Reflection” is synonymous for monadic semantics; and this text attempts some comonadic descent along functorial semantics : Dosen semiassociative coherence covers Maclane associative coherence [4] by some comonadic adjunction, embedding : SemiAssoc $\leftrightarrows$ Assoc : flattening reflection.

Categories [7] study the interaction between reflections and limits. The basic configuration for reflections is :

$$\begin{tikzcd}
\text{CoMod} \arrow[r, "\text{reflection}"] \arrow[d, "\text{identity}"'] \arrow[r, phantom, shift right=6ex, "\overset{\text{unit}}{\Longrightarrow}"] & \text{Mod} \arrow[dl, "\text{coreflection}"] \\
\text{CoMod} \arrow[d, "\text{reification}"'] & \\
\text{Modos} &
\end{tikzcd}$$

where, for all reification functor into any Modos category, the map
$(\_ \star \text{reflection}) \circ (\text{reification} \star \text{unit})$ is bijective, or same, for all object M' in CoMod, the polymorphic in M map
$(\text{coreflection } \_ \,) \circ \text{unit}_{M'} : \text{Mod}(\text{reflection } M', M) \to \text{CoMod}(M', \text{coreflection } M)$ is bijective (and therefore also polymorphic in M' with reverse map $\text{counit}_M \circ (\text{reflection } \_ \,)$ whose reversal is polymorphically determined by $(\text{coreflection} \star \text{counit}) \circ (\text{unit} \star \text{coreflection}) = \text{identity}$ and $(\text{counit} \star \text{reflection}) \circ (\text{reflection} \star \text{unit}) = \text{identity}$); and it is said that the unit natural/polymorphic/commuting transformation is the *unit of the reflection* and the reflective pair $(\text{reification} \circ \text{coreflection}, \text{reification} \star \text{unit})$ is some *coreflective ("Kan") extension functor* of the reification functor along the reflection functor. This text shows some comonadic adjunction, embedding : SemiAssoc $\leftrightarrows$ Assoc : flattening reflection.

Categories [6] [7] converge to the *descent technique*, this convergence is from both the *functorial semantics technique* with the *monadic adjunctions technique*. Now functorial semantics starts when one attempts to *internalize* the common phrasing of the logician model semantics, and this internalization has as consequence some functionalization/functorialization saturation/normalization of the original theory into some more synthetic theory; note that here the congruence saturation is some instance of postfix function composition and the substitution saturation is some instance of prefix function composition. The "Yoneda"/normalization lemma takes its sense here. And among all the relations between synthetic theories, get the *tensor of theories*, which is some *extension of theories*, and which is the coproduct (disjoint union) of all the operations of the component theories quotiented by extra *commutativity* between any two operations from any two distinct component theories; for example the tensor of two rings with units as synthetic theories gives the bimodules as functorial models.

Now Galois says that any radical extension of all the *symmetric functions* in some indeterminates, which also contains those indeterminates, is abe to be incrementally/resolvably saturated/"algebra" as some further radical extension whose interesting endomorphisms include all the permutations of the indeterminates. And when there are many indeterminates, then some of those permutations are properly preserved down the resolution ... but the resolution vanish any permutation ! In this context of saturated extensions, one then views any polynomial instead as its quotient/ideal of some ring of polynomials and then pastes such quotients into "algebraic algebras" or "spectrums" or "schemes" .. This is Galois descent along Borceux-Janelidze-Tholen [6]

Esquisse d'un programme : The raw combinatorial ("permutation group") angle converge to Aigner [5]. Another parallel of the raw combinatorial techniques of Galois is the raw proof techniques of Gentzen that inductive recursive arithmetic cannot well-order some ordinal. One question is whether the descent techniques and the proof techniques can converge. The initial item shall be to internalize/functorialize the semantics of Dosen book and do *automation programmation* so to gather data and examples and experiments. The automation programming technique has one common form mixing induction or simplification conversion or logical unification or substitution rewriting or repeated heuristic/attempt destructions or reflective decision procedure; for example the form behind the *crush* of Chlipala CPDT [3] is :

```
[ induction;
(eval beta iota; auto logical unify; auto substitution rewrite);
repeat ( (match goal | context match term => destruct);
(eval beta iota; auto logical unify; auto substitution rewrite) );
congruence; omega; anyreflection ]
```

The next item shall be to memo Borceux books 1 and 2, not only from the simplifying conversion angle or the unification ("logic programming") angle or the substitution rewrite angle, but also from the computational reflection angle. This computational reflection angle shall be far more than decision procedures, but rather shall be descent techniques for existence (fullness) and identification (failfulness); so this would allow for implicit arguments to be resolved after descent or for some arguments to be programmed after descent to some easier terminology .. This is 3 pages/day = 1 year memo reading.

2 Maclane Associative Coherence

This COQ text shows the semiassociativity completeness and coherence internal some encoding where associative coherence is the meta :

MACLANE PENTAGON IS SOME RECURSIVE SQUARE !! This recursive square normalize_map_assoc is the "functorial" parallel to the normalization/flattening of binary trees; and is simply the unit of the reflection for the adjunction.

The associative coherence comes before anything else, before the semiassociative completeness and before the semiassociative coherence.
- The associative coherence, by the recursive square lemma, critically reduce to the classification of the endomorphisms in the semiassociative category.
- This associative coherence do not lack some "Newman-style" diamon lemma. The comonadic adjunction, embedding : **arrows** $\leftrightarrows$ **arrows_assoc** : flattening reflection, which says/subgeres that semiassociative coherence covers (and temporarily comes before) associative coherence is actually done posterior-ly (for want of formality), after it is already known that the simpler **List** subcategory of **arrows** made of endomorphisms is enough to cover (and is equivalent to) **arrows_assoc**.
- The semiassociative coherence do lack some "Newman-style" diamon lemma; and, again posterior-ly, there exists some monadic adjunction, embedding : **List** $\leftrightarrows$ **arrows** : flattening reflection.

The associative category is the meta of the semiassociative category, and the phrasing of semiassociative completeness lemma_completeness shows this actuality very clearly. The semiassociative coherence is done in some internal ("first/second order" ??) encoding relative to associative coherence; may be exists some (yet to be found) "higher-order" / Coq Gallina encoding. The semiassociative coherence do lack some "Newman-style" diamon lemma. This diamon lemma is done in somme two-step process : first the codomain object of the diamon is assumed to be in normal/flattened form and this particularized diamon lemma *lemma_directedness* is proved without holding semiassociative completeness; then this assumption is erased and the full diamon lemma *lemma_coherence0* now necessitate the semiassociative completeness.

Dosen book [4] section 4.2 then section 4.3 is written into some computationally false order/precedence. The source of this falsification is the confusion/mixup between "convertible" (definitionality/meta equal) or "propositional equal" where the things are local variables instead of fully constructor-ed explicit terms.

These Coq texts do not necessitate impredicativity and do not necessitate limitation of the flow of information from proof to data and do not necessitate large inductive types with polymorphic constructors and do not necessitate universe polymorphism; therefore no distinction between `Prop` or `Set` or `Type` is made in these Coq texts. Moreover this Coq texts do not exhitate to use do not hesitate to use the very sensible/fragile library *Program.Equality* `dependent destruction` which may introduce extra non-necessary *eq_rect_eq* axioms. It is most possible to erase this *eq_rect_eq* (coherence !! ..) axiom from this Coq texts, otherwise this would be some coherence problem as deep/basic as associative coherence or semiassociative coherence.

Noson Yanofsky talks about some Catalan categories to solve associative coherence in hes doctor text, may be the “functorial” normalization/flattening here is related to hes Catalan categories. Also the *AAC tactics* or *CoqMT* which already come with COQ may be related, but the ultimate motivation is different ...

```
Infix "/\0" := (up_0) (at level 59, right associativity).
Print objects.


Inductive objects : Set :=
    letter : letters → objects | up_0 : objects → objects → objects.

Infix "~a" := same_assoc (at level 69).
Print same_assoc.

Inductive same_assoc
                  : ∀ A B : objects,
                    arrows_assoc A B → arrows_assoc A B → Set :=
     same_assoc_refl : ∀ (A B : objects) (f : arrows_assoc A B), f ~a f
   | same_assoc_trans : ∀ (A B : objects) (f g h : arrows_assoc A B),
                            f ~a g → g ~a h → f ~a h
   | same_assoc_sym : ∀ (A B : objects) (f g : arrows_assoc A B),
                          f ~a g → g ~a f
   | same_assoc_cong_com : ∀ (A B C : objects) (f f0 : arrows_assoc A B)
                                   (g g0 : arrows_assoc B C),
                                 f ~a f0 → g ~a g0 → g <oa f ~a g0 <oa f0
   | same_assoc_cong_up_1 : ∀ (A B A0 B0 : objects)
                                   (f f0 : arrows_assoc A B)
                                   (g g0 : arrows_assoc A0 B0),
                                 f ~a f0 → g ~a g0 → f /\1a g ~a f0 /\1a g0
   | same_assoc_cat_left : ∀ (A B : objects) (f : arrows_assoc A B),
                                 unitt_assoc B <oa f ~a f
   | same_assoc_cat_right : ∀ (A B : objects) (f : arrows_assoc A B),
                                 f <oa unitt_assoc A ~a f
   | same_assoc_cat_assoc : ∀ (A B C D : objects) (f : arrows_assoc A B)
                                   (g : arrows_assoc B C) (h : arrows_assoc C D),
                                 h <oa g <oa f ~a (h <oa g) <oa f
   | same_assoc_bif_up_unit : ∀ A B : objects,
                                   unitt_assoc A /\1a unitt_assoc B ~a
                                   unitt_assoc (A /\0 B)
   | same_assoc_bif_up_com : ∀ (A B C A0 B0 C0 : objects)
                                    (f : arrows_assoc A B) (g : arrows_assoc B C)
                                    (f0 : arrows_assoc A0 B0)
                                    (g0 : arrows_assoc B0 C0),
                                          (g <oa f) /\1a (g0 <oa f0) ~a
                                          g /\1a g0 <oa f /\1a f0
  | same_assoc_bracket_left_5 : ∀ A B C D : objects,
                                              bracket_left_assoc (A /\0 B) C D <oa
                                              bracket_left_assoc A B (C /\0 D) ~a
                                              bracket_left_assoc A B C /\1a unitt_assoc D <oa
                                              bracket_left_assoc A (B /\0 C) D <oa
                                              unitt_assoc A /\1a bracket_left_assoc B C D
  | same_assoc_nat : ∀ (A A' : objects) (f : arrows_assoc A' A)
                                   (B B' : objects) (g : arrows_assoc B' B)
                                   (C C' : objects) (h : arrows_assoc C' C),
                                 bracket_left_assoc A B C <oa f /\1a g /\1a h ~a
                                 (f /\1a g) /\1a h <oa bracket_left_assoc A' B' C'
  | same_assoc_bracket_right_bracket_left : ∀ A B C : objects,
                                                          bracket_right_assoc A B C <oa
                                                          bracket_left_assoc A B C ~a
                                                          unitt_assoc (A /\0 B /\0 C)
  | same_assoc_bracket_left_bracket_right : ∀ A B C : objects,
                                                          bracket_left_assoc A B C <oa
                                                          bracket_right_assoc A B C ~a
                                                          unitt_assoc ((A /\0 B) /\0 C)

Infix "~s" := same' (at level 69).
About same'.

Print normal.


Inductive normal : objects → Set :=
     normal_cons1 : ∀ l : letters, normal (letter l)
  | normal_cons2 : ∀ (A : objects) (l : letters),
                              normal A → normal (A /\0 letter l).

Print normalize_aux.


fix normalize_aux (Z A : objects) {struct A} : objects :=
  match A with
  | letter l ⇒ Z /\0 letter l
  | A1 /\0 A2 ⇒ normalize_aux (normalize_aux Z A1) A2
  end
     : objects → objects → objects

Print normalize.


fix normalize (A : objects) : objects :=
  match A with
  | letter l ⇒ letter l
  | A1 /\0 A2 ⇒ normalize A1 </\0 A2
  end
```

`Print` *developed.*

This development or factorization lemma necessitate some deep ('well-founded') induction, using some measure *coherence.length* which shows that this may be related to arithmetic factorization. `Print` *coherence.length.*

```
fix length (A B : objects) (f : arrows A B) {struct f} : nat :=
  match f with
  | unitt _ ⇒ 2
  | bracket_left _ _ _ ⇒ 4
  | up_1 A0 B0 A1 B1 f1 f2 ⇒ length A0 B0 f1 × length A1 B1 f2
  | com A0 B0 C f1 f2 ⇒ length A0 B0 f1 + length B0 C f2
  end
```

```
Check development: ∀ (len : nat) (A B : objects) (f : arrows A B),
       length f ≤ len →
       {f' : arrows A B &
       (developed f' × ((length f' ≤ length f) × (f ~~ f')))%type}.
```

```
Notation normalize_aux_unitrefl_assoc := normalize_aux_arrow_assoc.
Print normalize_aux_arrow_assoc.
```

```
fix normalize_aux_arrow_assoc (Y Z : objects) (y : arrows_assoc Y Z)
                                          (A : objects) {struct A} :
  arrows_assoc (Y /\0 A) (Z </\0 A) :=
  match A as A0 return (arrows_assoc (Y /\0 A0) (Z </\0 A0)) with
  | letter l ⇒ y /\1a unitt_assoc (letter l)
  | A1 /\0 A2 ⇒
      normalize_aux_arrow_assoc (Y /\0 A1) (Z </\0 A1)
        (normalize_aux_arrow_assoc Y Z y A1) A2 <oa
      bracket_left_assoc Y A1 A2
  end
     : ∀ Y Z : objects,
       arrows_assoc Y Z →
       ∀ A : objects, arrows_assoc (Y /\0 A) (Z </\0 A)
```

```
Notation normalize_unitrefl_assoc := normalize_arrow_assoc.
Print normalize_arrow_assoc.
```

```
fix normalize_arrow_assoc (A : objects) : arrows_assoc A (normalize A) :=
  match A as A0 return (arrows_assoc A0 (normalize A0)) with
  | letter l ⇒ unitt_assoc (letter l)
  | A1 /\0 A2 ⇒ normalize_aux_unitrefl_assoc (normalize_arrow_assoc A1) A2
  end
      : ∀ A : objects, arrows_assoc A (normalize A)
```

Check th151 : ∀ A : **objects**, **normal** $A \to$ normalize A = A.

Aborted th270: For local variable A with **normal** A, although there is the propositional equality th151: normalize $A = A$, that normalize A, A are not convertible (definitionally/meta equal); therefore one shall not regard normalize_unitrefl_assoc, *unitt* A as sharing the same domain-codomain indices of **arrows_assoc**

Check th260 : ∀ N P : **objects**, **arrows_assoc** N $P \to$ normalize N = normalize P.

Aborted lemma_coherence_assoc0: For local variables N, P with **arrows_assoc** N P, although there is the propositional equality th260: normalize N = normalize P, that normalize A, normalize B are not convertible (definitionally/meta equal); therefore some transport other than *eq_rect*, some coherent transport is lacked.

```
Check normalize_aux_map_assoc
        : ∀ (X Y : objects) (x : arrows_assoc X Y) (Z : objects)
            (y : arrows_assoc Y Z),
          directed y →
          ∀ (A B : objects) (f : arrows_assoc A B),
          {y_map : arrows_assoc (Y </\0 A) (Z </\0 B) &
          ((y_map <oa normalize_aux_unitrefl_assoc x A ~a
            normalize_aux_unitrefl_assoc y B <oa x /\1a f) × directed y_map)%type}.
Check normalize_map_assoc
      : ∀ (A B : objects) (f : arrows_assoc A B),
          {y_map : arrows_assoc (normalize A) (normalize B) &
          ((y_map <oa normalize_unitrefl_assoc A ~a
            normalize_unitrefl_assoc B <oa f) × directed y_map)%type}.
Print Assumptions normalize_map_assoc.

Closed under the global context


Print Assumptions lemma_coherence_assoc.


JMeq.JMeq_eq
Eqdep.Eq_rect_eq.eq_rect_eq
```

ERRORS OF DOSEN: 1. CONFUSE MIXUP CONVERTIBLE (DEFINITIONALLY/META EQUAL) WITH PROPOSITIONAL EQUAL WHEN THE THINGS ARE VARIABLES INSTEAD OF FULLY CONSTRUCTOR-ED EXPLICIT TERMS, 2. RELATED TO THIS IS THE COMPUTATIONNALLY WRONG ORDER OF HES PRESENTATION

3 Dosen Semiassociative Coherence

```
Print nodes.


Inductive nodes : objects → Set :=
    self : ∀ A : objects, A
  | at_left : ∀ A : objects, A → ∀ B : objects, A /\0 B
  | at_right : ∀ A B : objects, B → A /\0 B.

Infix "<r" := lt_right (at level 70).
Print lt_right.


Inductive lt_right : ∀ A : objects, A → A → Set :=
    lt_right_cons1 : ∀ (B : objects) (z : B) (C : objects),
                        self (C /\0 B) <r at_right C z
  | lt_right_cons2 : ∀ (B C : objects) (x y : B),
                        x <r y → at_left x C <r at_left y C
  | lt_right_cons3 : ∀ (B C : objects) (x y : B),
                        x <r y → at_right C x <r at_right C y.

Notation comparable A B := {f : arrows_assoc A B | True} (only parsing).

Check bracket_left_on_nodes
    : ∀ A B C : objects, nodes (A /\0 (B /\0 C)) → nodes ((A /\0 B) /\0 C).



Definition bracket_left_on_nodes (A B C : objects) ( x : nodes (A ∧ (B ∧ C)) ) : nodes
((A ∧ B) ∧ C).

dependent destruction x.
exact (at_left (self (A ∧ B)) C).
exact (at_left (at_left x B) C).
dependent destruction x.
exact (self ((A ∧ B) ∧ C)).
exact (at_left (at_right A x) C).
exact (at_right (A ∧ B) x).
Defined.

Check arrows_assoc_on_nodes : ∀ A B : objects, arrows_assoc A B → nodes A → nodes
B.
```

Soundness.

```
Check lem033 : ∀ (A B : objects) (f : arrows A B) (x y : A),
        f x <r f y → x <r y.
```

Completeness. Deep ('well-founded') induction on lengthn'', with accumulator/continuation *cumul_letteries*.

Check lemma_completeness : $\forall$ (B A : **objects**) (f : **arrows_assoc** B A)
($H\_cumul\_lt\_right\_B$: $\forall$ x y : **nodes** B, **lt_right** x y $\rightarrow$ **lt_right** (f x) (f y))
, **arrows** A B.

Check lem005700: $\forall$ (B : **objects**) (len : **nat**),
$\forall$ (*cumul_letteries* : **nodes** B $\rightarrow$ **bool**)
(*H_cumul_letteries_wellform* : cumul_letteries_wellform' B *cumul_letteries*)
(*H_cumul_letteries_satur* : $\forall$ y : **nodes** B, *cumul_letteries* y = true
$\rightarrow$ $\forall$ z : **nodes** B, lt_leftorright_eq y z $\rightarrow$ *cumul_letteries* z = true)
(*H_len* : lengthn'' *cumul_letteries* *H_cumul_letteries_wellform* $\leq$ *len*),
$\forall$ (A : **objects**) (f : **arrows_assoc** B A)
(*H_node_is_lettery* : $\forall$ x : **nodes** B, *cumul_letteries* x = true $\rightarrow$ node_is_lettery f x)
(*H_object_at_node* : $\forall$ x : **nodes** B, *cumul_letteries* x = true $\rightarrow$ object_at_node x = object_at_node (f x))
(*H_cumul_B* : $\forall$ x y : **nodes** B, **lt_right** x y $\rightarrow$ **lt_right** (f x) (f y))
, **arrows** A B.

Print Assumptions lem005700.

Get two equivalent axioms.

JMeq.JMeq_eq : $\forall$ (A : Type) (x y : A), *JMeq.JMeq* x y $\rightarrow$ $x = y$
Eqdep.Eq_rect_eq.eq_rect_eq : $\forall$ (U : Type) (p : U) (Q : U $\rightarrow$ Type)
(x : Q p) (h : $p = p$), x = *eq_rect* p Q x p h

Infix "<l" := **lt_left** (at level 70).
Print *lt_left*.

Maybe some betterement revision/egition by using *objects_same* is necessary here. Contrast this eq with *objects_same* Print *lt_leftorright_eq*.

Notation lt_leftorright_eq x y :=
(*sum* (*eq* x y) (*sum* (**lt_left** x y) (**lt_right** x y))).

nodal_multi_bracket_left_full below and later really lack this constructive equality *objects_same*, so that we get transport map which are coherent, transport map other than *eq_rect* Print *objects_same*.

Inductive *objects_same* : **objects** $\rightarrow$ **objects** $\rightarrow$ Set :=
objects_same_cons1 : $\forall$ l : *letters*,
objects_same (*letter* l) (*letter* l)
| *objects_same_cons2* : $\forall$ A A' : **objects**,

```
objects_same A A' →
∀ B B' : objects,
objects_same B B' →
objects_same (A /\0 B) (A' /\0 B').
```

nodal_multi_bracket_left_full is one of the most complicated/multifolded construction in this coq text. nodal_multi_bracket_left_full below and later really lack this constructive equality *objects_same*, so that we get transport map which are coherent, transport map other than *eq_rect*

```
Print "/\\".

 fix foldright (A : objects) (Dlist : list objects) {struct Dlist} :
  objects :=
  match Dlist with
  | nil ⇒ A
  | (D0 :: Dlist0)%list ⇒ foldright A Dlist0 /\0 D0
```

```
Check multi_bracket_left : ∀ (A B C : objects) (Dlist : list objects),
        arrows (A /\0 (B /\0 C /\\ Dlist)) ((A /\0 B) /\0 C /\\ Dlist).
```

```
Check (fun A (x : nodes A) (A2 B2 C2 : objects) (Dlist2 : list objects) ⇒
     @nodal_multi_bracket_left_full A x A2 B2 C2 Dlist2).
```

```
Print object_at_node.

object_at_node =
fix object_at_node (A : objects) (x : A) {struct x} : objects :=
  match x with
  | self A0 ⇒ A0
  | at_left A0 x0 _ ⇒ object_at_node A0 x0
  | at_right _ B x0 ⇒ object_at_node B x0
  end
```

object_is_letter is some particularised sigma type so to do convertibility (definitinal/meta equality) instantiatiations instead and avoid propositional equalities. `Print` *object_is_letter*.

```
Inductive object_is_letter : objects → Set :=
     object_is_letter_cons : ∀ l : letters, object_is_letter (letter l).
```

```
Print object_is_tensor.
```

```
Print node_is_letter.

Notation node_is_letter x := (object_is_letter (object_at_node x)).
```

```
Print node_is_tensor.
```

```
Notation node_is_tensor x := (object_is_tensor (object_at_node x)).

Print node_is_lettery.


Notation node_is_lettery f w :=
  (prod
      ( ∀ (x : nodes _), lt_leftorright_eq w x → lt_leftorright_eq (f w) (f x) )
      ( ∀ (x : nodes _), lt_leftorright_eq (f w) (f x) → lt_leftorright_eq ((rev f) (f w)) ((rev
f) (f x)) )).

Print cumul_letteries_wellform'.


Notation cumul_letteries_wellform' B cumul_letteries :=
  (∀ x : B,
    object_is_letter (object_at_node x) → eq (cumul_letteries x) true).

Print lengthn'' .


lengthn'' =
fix lengthn'' (A : objects) (cumul_letteries : A → bool)
                (H_cumul_letteries_wellform : cumul_letteries_wellform' A
                                                    cumul_letteries) {struct A} :
  nat :=
  match
    A as o
    return
      (∀ cumul_letteries0 : o → bool,
       cumul_letteries_wellform' o cumul_letteries0 → nat)
  with
  | letter l ⇒
      fun (cumul_letteries0 : letter l → bool)
        (_ : cumul_letteries_wellform' (letter l) cumul_letteries0) ⇒ 1
  | A1 /\0 A2 ⇒
      fun (cumul_letteries0 : A1 /\0 A2 → bool)
        (H_cumul_letteries_wellform0 : cumul_letteries_wellform' (A1 /\0 A2)
                                               cumul_letteries0) ⇒
      let s :=
        Sumbool.sumbool_of_bool (cumul_letteries0 (self (A1 /\0 A2))) in
      if s
      then 1
      else
       let IHA1 :=
    lengthn'' A1 (restr_left cumul_letteries0)
      (restr_left_wellform cumul_letteries0 H_cumul_letteries_wellform0) in
  let IHA2 :=
    lengthn'' A2 (restr_right cumul_letteries0)
      (restr_right_wellform cumul_letteries0 H_cumul_letteries_wellform0) in
  IHA1 + IHA2
end cumul_letteries H_cumul_letteries_wellform

Check restr_left : ∀ B1 B2 : objects, (B1 /\0 B2 → bool) → B1 → bool.
Check restr_left_wellform : ∀ (B1 B2 : objects) (cumul_letteries : B1 /\0 B2 → bool),
    cumul_letteries_wellform' (B1 /\0 B2) cumul_letteries →
    cumul_letteries_wellform' B1 (restr_left cumul_letteries).
```

More at https://github.com/mozert/ .

References

[1] Jason Gross, Adam Chlipala, David I. Spivak. “Experience Implementing a Performant Category-Theory Library in Coq” In: Interactive Theorem Proving. Springer, 2014.

[2] Gregory Malecha, Adam Chlipala, Thomas Braibant. “Compositional Computational Reflection” In: Interactive Theorem Proving. Springer, 2014.

[3] Adam Chlipala. “Certified Programming with Dependent Types” http://adam.chlipala.net/cpdt/

[4] Kosta Dosen, Zoran Petric. “Proof-Theoretical Coherence” http://www.mi.sanu.ac.rs/~kosta/coh.pdf , 2007.

[5] Martin Aigner. “Combinatorial Theory” Springer, 1997

[6] Francis Borceux, George Janelidze. “Galois Theories” Cambridge University Press, 2001.

[7] Francis Borceux. “Handbook of categorical algebra. Volumes 1 2 3” Cambridge University Press, 1994.