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

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

March 21, 2015

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.
UPDATE HERE IN FEWHOURS https://github.com/mozert

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

$$\begin{tikzcd}
\mathrm{CoMod} \arrow[r, "\text{reflection}"] \arrow[d, "\text{identity}"'] \arrow[r, phantom, shift right=4ex, "\overset{\text{unit}}{\Longrightarrow}"] & \mathrm{Mod} \arrow[dl, "\text{coreflection}"] \\
\mathrm{CoMod} \arrow[d, "\text{reification}"'] & \\
\mathrm{Modos} &
\end{tikzcd}$$

where, for all reification functor into any Modos category, the map $(\_ \star \mathrm{reflection}) \circ (\mathrm{reification} \star \mathrm{unit})$ is bijective, or same, for all object M' in CoMod, the polymorphic in M map $(\mathrm{coreflection}\ \_\ ) \circ \mathrm{unit}_{M'} : \mathrm{Mod}(\mathrm{reflection}\ M', M) \to \mathrm{CoMod}(M', \mathrm{coreflection}\ M)$ is bijective (and therefore also polymorphic in M' with reverse map $\mathrm{counit}_M \circ (\mathrm{reflection}\ \_\ )$ whose reversal is polymorphically determined by $(\mathrm{coreflection} \star \mathrm{counit}) \circ (\mathrm{unit} \star \mathrm{coreflection}) = \mathrm{identity}$ and $(\mathrm{counit} \star \mathrm{reflection}) \circ (\mathrm{reflection} \star \mathrm{unit}) = \mathrm{identity}$); and it is said that the unit natural/polymorphic/commuting transformation is the unit of the reflection and the reflective pair $(\mathrm{reification} \circ \mathrm{coreflection}, \mathrm{reification} \star \mathrm{unit})$ 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 [5] [6] converge to the descent technique from the functorial semantics technique with the monadic adjunctions technique. Now functorial semantics arise when one attempt to internalize the common phrasing of the logician model theory, and this internalization has as consequence the functionalization/functorialization/saturation ....tensor of theories ... Galois extensions with radical roots symmetric polynomial solvable incompatible with symmetry ... ring extension algebraic functor comonadicity. Gentzen prophecies; Galois-Gentzen-Galthendiek the 3 G's of math, which 4th G for logical recur.

Esquisse d'un programme: Now when try to internalize semantics in Dosen book then get functorialization; therefore writing some Coq text for more from Dosen book is something that shall be done to gather data and examples. Also exploring Borceux books from computational reflection point of view sha be done: 3 pages/day = 1 year reading.

The common auto technique (the one behind CPDT *crush* [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]

In other words, each action is one of: convert, or logify recur, or unify, or substitute rewrite, or reflect. And one attempts some reflection after logic programming.

2 Misc (TO BE UPDATED LATER)

MACLANE PENTAGON IS SOME COMONADIC DESCENT. LOGICAL COHERENCE. CATEGORICAL DESCENT. COQ DEDUKTI MODULO.

1. LOGICAL COHERENCE -GROSS, CHLIPALA, COMONADIC FUNCTORIAL SEMANTIC,~~ GALOIS, GALTHENDIECK, GENTZEN -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 -METAAUTO LOGICAL PROGRAMMING CONTRAST REFLECTION PROGRAMMING -CATEGORICAL LOGICAL COHERENCE AS META REFLECTION PROGRAMMING -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 -convert or logify recur or unify or substitute rewrite or reflect

(... classification more than property specification subset, example: regurlar expression, red-black tree, closed categories, categories ... ... recur on applied subterm or nested subterm ...)

2. CATEGORICAL DESCENT -POLYMORPHIC REFLECTION (ADJUNCTION)~~, MONADIC ADJUNCTION -INTERNALISATION -> FUNCTORIAL SATURATION (YONEDA FREE ALGEBRA) -TENSOR OF THEORIES -COMONADICITY SHALL BE VERY RELATED TO COHERENCE -KAN EXTENSIONS IS CONTEXT FOR PROOFS BY REIFICATION/REFLECTION -BASE FOR KAN EXTENSIONS MAYBE MODOS, MOVE FROM CARTESIAN LIMITS TO ?? -(NORMALIZE_ARROW_ASSOC ON NORMAL IS REVERSIBLE THEREFORE NORMALIZE_UNIT_ASSOC IS UNIT OF REFLECTION)

-SemiAssoc <–> Assoc COMONADIC , NO LACK NEWMAN CONFLUENCE

List <–> SemiAssoc MONADIC , LACK NEWMAN CONFLUENCE, MAYBE "ITERCOVER RESOLUTION" ? -MORE BY INTERNALISATION OF COMMON COHERENCES -NOT YET FULL CATEGORICAL DESCENT FORM, PROGRAMME LOGICAL DOSEN AND CATEGORICAL BORCEUX 1 2 AUTO DESCENT VIEW, RELATE CONVERTIBILITY TYPES BY AUTORESOLUTION OR TACTIC, RELATE CANONICAL-STRUCTURE-AND-COERCION RESOLUTION

3. COQ DEDUKTI MODULO -COQTEXT ASSOC RECURSIVE SQUARE, NO ASSOC PREORDER -COQTEXT SEMIASSOC COMPLETENESS, NO SEMIASSOC CONFLUENCE, NO SEMIASSOC PREORDER

3 Maclane Associative Coherence

```
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),
```

$$normal\ A \rightarrow normal\ (A\ /\backslash 0\ letter\ l).$$

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 → 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 → 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.
```

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

4 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
```

: $\forall\ A\ B\ C$: **objects**, **nodes** (A /\0 (B /\0 C)) $\to$ **nodes** ((A /\0 B) /\0 C).

`Definition` *bracket_left_on_nodes* ($A\ B\ C$: *objects*) (x : *nodes* ($A \wedge (B \wedge C)$)) : *nodes* ($(A \wedge B) \wedge C$).

`dependent destruction` x.
`exact` (*at_left* (*self* ($A \wedge B$)) C).
`exact` (*at_left* (*at_left* x B) C).
`dependent destruction` x.
`exact` (*self* (($A \wedge B) \wedge C$)).
`exact` (*at_left* (*at_right* A x) C).
`exact` (*at_right* ($A \wedge B$) x).
`Defined.`

`Check` arrows_assoc_on_nodes : $\forall\ A\ B$: **objects**, **arrows_assoc** $A\ B \to$ **nodes** $A \to$ **nodes** B.

Soundness.

`Check` lem033 : $\forall$ ($A\ B$: **objects**) (f : **arrows** $A\ B$) ($x\ y$: A),
$f\ x$ `<r` $f\ y \to x$ `<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** $x\ y \to$ **lt_right** ($f\ x$) ($f\ y$))
, **arrows** $A\ B$.

`Check` lem005700: $\forall$ (B : **objects**) (*len* : **nat**),
$\forall$ (*cumul_letteries* : **nodes** $B \to$ **bool**)
(*H_cumul_letteries_wellform* : cumul_letteries_wellform' B *cumul_letteries*)
(*H_cumul_letteries_satur* : $\forall\ y$: **nodes** B, *cumul_letteries* y = true
$\to \forall\ z$: **nodes** B, lt_leftorright_eq $y\ z \to$ *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 $\to$ 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' $\rightarrow$
$\forall$ B B' : *objects*,
objects_same B B' $\rightarrow$
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] Francis Borceux, George Janelidze. “Galois Theories”. Cambridge University Press, 2001.

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