AnthropLOGIC.com WorkSchool 365 for e-commerce, e-learning and e-research with applications to computer proofs in physics

Short: AnthropLOGIC.com WorkSchool 365 is a legal business-university for e-commerce, e-learning, and e-research with published applications in the Microsoft Commercial Marketplace. The SurveyQuiz and EventReview e-commerce and e-learning applications are an integration of many popular business software to enable learners/reviewers to share the transcripts/receipts of their (quizzes/reviews) school/work using password-less user identities. The MODOS e-research application is a library of new-mathematics documents and their new proof-assistant software editor to share and publish documents for e-learning or for e-research in logic and geometry (potentially in physics). Outline:

1. WorkSchool 365 for applications in e-commerce and e-learning (SurveyQuiz transcripts, EventReview receipts)

(1.) What problem is to be solved? From the legal perspective, as prescribed by many legislative assemblies everywhere, a school is defined by its ability to output the shareable transcripts/receipts records of the learning-discovery-engineering-and-teaching/reviewing done by the learners and teachers (reviewers). Links: https://www.ontario.ca/laws/statute/00p36

An ambient legal requirement is that there should be no forced/assault-fool/[intoxicated-by-bad-habits]-and-theft/lie/falsification of those transcripts/receipts records. The initial step is the authentication of the users without requiring excessive personal information (beyond an email address). The second step is the authorization of access to the transcripts/receipts records with guarantee of the integrity of the data.

From the commerce perspective, a business is defined by its ability to account for the direct currency transactions among all the trading parties (learner-reviewer) without requiring excessive financial information (beyond a payout address) and without relying on indirect government/public currencies.

(2.) WorkSchool 365 integrates the Customer Relationship Management (CRM) + Learning Management System (LMS) for your Business or University to engage/qualify/educate prospective users into paying/subscribed/grantee customers/students or paid reviewers/teachers via an integration of Stripe.com e-commerce payment (Card, AliPay, Wechat Pay, PayPal) + Microsoft.com Business Applications MBA (Azure AD, SharePoint Teams, Power Automate). Links: https://azuremarketplace.microsoft.com/en-us/marketplace/apps/anthroplogicworkschool3651593890930054.anthroplogic_workschool_365

(2.1) WorkSchool 365 SurveyQuiz are Word documents of large-scale auto-graded survey/quizzes with shareable transcripts of School, with integration of the Coq 365 proof-assistant Word add-in and examples from the GIAM textbook. Links: http://giam.southernct.edu ; https://surveyquiz.anthroplogic.com

(2.2) WorkSchool 365 EventReview are SharePoint calendars of paid review-assignments for events/documents with shareable receipts of Work. Links: https://eventreview.anthroplogic.com

(2.3) WorkSchool 365 Users are password-less sign-in/sign-up authenticated via Microsoft/Azure or Google or Facebook or Email. The users in Cycle 3 (Reviewers) may create their own thematic instances of the SurveyQuiz and EventReview software for the free users in Cycle 1 (Learners) or the paying non-free users in Cycles 2 (Seminarians). Links: https://anthroplogic.com

2. WorkSchool 365 for applications in e-research (MODOS proof-assistant)

(1.) What problem is to be solved? The discovery of some new mathematics unnoticed for the past half-century, the MODOS proof-assistant founded on homotopical computational logic for geometry (potentially in physics), has applications into the communication-format of mathematics documents itself.

This aspect of the applications should be emphasized by the communication-format in which this library of new-mathematics is published and reviewed inside structured-documents which integrate this proof-assistant editor and the multi-author version-history.

(2.) OCAML/COQ computer is for reading and writing mathematical computations and proofs. Any collection of elements ("datatype") may be presented constructively and inductively, and thereafter any function ("program") may be defined on such datatype by case-analysis on the constructors and by recursion on this function itself. Links: http://coq.inria.fr

Moreover, the COQ computer extends mere computations (contrasted to OCAML) by allowing any datatype to be parameterized by elements from another datatype, therefore such parameterized datatypes become logical propositions and the programs defined thereon become proofs.

(3.) The computational logic foundation of OCAML/COQ is "type theory", where there is no real grammatical distinction between elements and types as grammatical terms, and moreover only "singleton" terms can be touched/probed.

Type theory was OK for computer-science applications, but is not OK for mathematics (categorical-algebra). A corollary is that (differential) "homotopy type theory" inherits the same flaws. The algebraic geometry of affine schemes say that "points" (prime ideals) are more than mere singletons: they are morphisms of irreducible closed subschemes into the base scheme.

It is now learned that it was not necessary to retro-grade categorical-algebra into type theory (“categorical-logic” in the sense of Joachim Lambek); but there is instead some alternative reformulation of categorical-algebra as a cut-elimination computational-logic itself (in the sense of Kosta Dosen and Pierre Cartier), where the generalized elements (arrows) remain internalized/accumulated ("point-as-

morphism" / polymorphism) into grammatical-constructors and not become variables/terms as in the usual topos internal-language... Links: http://www.mi.sanu.ac.rs/~kosta ; http://www.ihes.fr/~cartier

(4.) GAP/SINGULAR computer is for computing in permutation groups and polynomial rings, whenever computational generators are possible, such as for the orbit-stabilizer algorithm ("Schreier generators") or for the multiple-variables multiple-divisors division algorithm ("Euclid/Gauss/Groebner basis"). Links: https://www.gap-system.org

In contrast to GAP/SINGULAR which does the inner computational-algebra corresponding to the affine-projective aspects of geometry, the MODOS aims at the outer logical/categorical-algebra corresponding to the parameterized-schematic aspects of geometry; this contrast is similar as the OCAML-COQ contrast. In short: MODOS does the computational-logic of the coherent sheaf modules over some base scheme; dually the relative support/spectrum of such sheaf modules/algebras are schemes parameterized over this base scheme (alternatively, the slice topos over this sheaf is étale over the base topos). Links: https://stacks.math.columbia.edu/tag/01LQ

(5.) MODOS proof-assistant has solved the critical techniques behind those questions, even if the production-grade engineering is still lacking. Some programming techniques ("cut-elimination", "confluence", "dependent-typed functional programming"...) from computer-science (electrical circuits) generalize to the alternative reformulation of categorical-algebra as a cut-elimination computational-logic ("adjunctions", "comonads", "products", "enriched categories", "internal categories", "fibred category with local internal products", "associativity coherence"...). Links: https://github.com/1337777/cartier ; https://github.com/1337777/dosen

2.1. MODOS proof-assistant's potential applications in the computational logic for the geometry of physics

(1.) What problem is to be solved? Quantum Fields is an attempt to upgrade the mathematics of the 19th century's Maxwell equations of electromagnetism, in particular to clarify the duality between matter particles and light waves. However, those differential geometry methods (even post-Sardanashvily) are still "equational algebra"; no upgrade on the «computational-logic formulation» has happened.

(2.) The geometry content of the quantum fields in physics is often in the form of the variational calculus to find the optimal action defined on the jet bundles of the field configurations. This is often formulated in differential, algebraic and even differential "homotopy type theory", of fibered manifolds with equivariance under natural (gauge) symmetries. However, the interdependence between the geometry and the dynamics/momentum data/tensor is still lacking some computational-logic (constructive, mutually-inductive) formulation. Links: https://ncatlab.org/nlab/show/jet+bundle

(3.) The computational content of the quantum field is often in the form of the statistics of the correlation at different localities of some field-configuration and the statistics of the partition function expressed in the field-configurations modes. A corollary: the point in spacetime is not "singleton" (not even some "string" ...); the field configurations are statistical/thermal/quantum and "uncertain" (the derivative/commutator of some observable along another observable is not zero). Hint: The MODOS proof-assistant has potential applications in the computational logic for the geometry of physics.

3. Appendix: What is the minimal example of sheaf cohomology? Grammatically

Short: Hold the site of the 3-points space with two open sets U and V which have another non-empty intersection W. Hold M be the sheaf generated by two elements f function on U and g function on V, without any assumption of compatibility over W. Hold N be the sheaf generated by two elements f' function on U and g' function on V and generated by one compatibility relation between f' and g' over W. Hold mn be the transformation of sheaves from M to N which maps f to f' and maps g to g'. Then mn has surjective image-sheaf, but is not surjective map at each open. The lemma is that this description can be written grammatically, in some new homotopical parameterized computational logic for geometry: MODOS. In short: MODOS interfaces the COQ categorial logic of sheaves down to the GAP/SINGULAR algebra of modules.

0. What is the end goal?

The end goal is not to verify that the sense is correct; of course, everything here makes sense. The end goal is whether it is possible to formulate some computational logic grammatically. Therefore, this text shall be read first without attention to the sense, then read twice to make sense. Ref: https://github.com/1337777/cartier

1. The generating site.

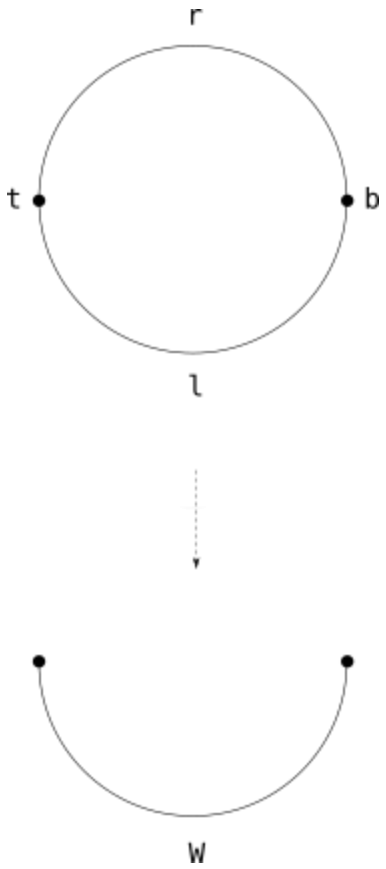
The topos of sheaves is presentable by generators from the site, with pullback/substitution-distributing-over-colimits. The site should have finiteness, such as the site of open subsets of some finite space or finitely generated space. The “points” of such finite space should be thought of as ordered-by-inclusion “simplex faces” (irreducible closed subsets) of another non-finite space. For example, the finite space corresponding to the circle is the “pseudocircle” whose underlying set is a 4-set {l,r,t,b} (for the ‘left side’, ‘right side’, ‘top point’, and ‘bottom point’ of the circle) and whose collection of open subsets, is {{l,r,t,b},{l,r,t},{l,r,b},{l,r},{l},{r},∅}.

Now furthermore there is some open map from this pseudocircle to the 3-points finite space of the introduction text above; therefore, the generating site is parameterized.

coq10 / coq

```
Parameter Parameter0 :
obGenerator -> obParametrizator.


Parameter Parameter1 :
  forall A A' : obGenerator,
'Generator( A ~> A' ) -> 'Parametrizator( Parameter0 A ~> Parameter0 A').
```

coq10 / coq



2. The data-objects, their constructors and algebra.

The elements of some data object sheaf can be either generating constructors or algebraic (restriction, zero, addition, differential). First, the codes for the data objects are collected; for example, the codes of the binary true-false data object are:

coq20 / coq

```
Inductive obData_Param : Type :=
| Binary_Param : obData_Param .


Inductive obData :
  forall  (Param_SubstF : obData_Param), Type :=
| Binary_Form : obData Binary_Param .
```

coq20 / coq

Memo that the formulation of the generating constructors is such that they accumulate restrictions by arrows belonging to some viewing (sieve); this enables to eliminate the algebraic-restrictions from the solution. Also, the grammatical entry for the data-object elements is mutually recursive with the grammatical entry for the general logic-objects codes and the grammatical entry for the pseudo codes of the morphisms.

coq30 / coq

```
Inductive elAlgebra_Param (* VV : forall paramlocal, Vision *):
  forall (Param_SubstF: obData_Param _)
  (G : obGenerator) (paramlocal : Sense00_Param_SumSubstF G)
  (P : obParametrizator) (inFibre_P : inFibre G P), Type :=

with elConstruct_Form (* VV : forall paramlocal, Vision *) :
  forall (Param_SubstF : obData_Param _) (F : obData Param_SubstF)
  param (cons_paramlocal : c.elConstruct_Param (* VV *) Param_SubstF
           (sval Sense1_Param_subst G param) (InFibre G))
  form, Type :=

| True_Binary_Form :
  forall (G' : obGenerator) (g : 'Generator( G' ~> GTop (* | VV true *) )),
elConstruct_Form _

with elAlgebra_Form (* VV : forall paramlocal, Vision *) :
  forall (Param_SubstF : obData_Param _) (F : obData Param_SubstF)
  param (cons_paramlocal : c.elAlgebra_Param (* VV *) Param_SubstF
           (sval Sense1_Param_subst G param) (InFibre G))
  form, Type :=

| ElConstruct_elAlgebra_Form :
  forall _,
elConstruct_Form _  -> elAlgebra_Form _

| Restrict_elAlgebra_Form :
  forall param cons_paramlocal form (cons_form : elAlgebra_Form form),
  forall (G' : obGenerator) (g : 'Generator( G' ~> G (* | VV paramlocal *)
)),
elAlgebra_Form ( g o> form )
```

```
(* | Zero :  forall _ , elAlgebra_Form _
| Addition :  forall _ , elAlgebra_Form _ -> elAlgebra_Form _ ->
elAlgebra_Form _
| Differential :  forall _ , elAlgebra_Form (n+1) _  -> elAlgebra_Form (n) _
*)


with obCoMod_Param :
  forall _, Type :=
```

coq30 / coq

3. The general logic-objects, mutually-inductive with the pseudo codes for the morphisms

Here the clause [ViewOb] is the Yoneda embedding of the generators from the site to the topos. And the clause [Viewing] takes from any data-object together with some viewing (sieve) to produce the general logic-objects. And the clause [ViewedOb] is the sheafification operation. And the clause [FormatOb] is polymorph context-extension which takes any form object to produce some parameter object.

coq40 / coq

```
Inductive elAlgebra_Param : Type :=
with obCoMod_Param :
  forall _, Type :=


| ViewOb_Param :
  forall P : obParametrizator,
obCoMod_Param _


| Viewing_Param_default :
  forall (Param_SubstF : obData_Param _)
  (* VV : forall paramlocal, Vision *),
obCoMod_Param _


| ViewedOb_Param_default :
  forall (Param_F : obCoMod_Param _),
```

```
obCoMod_Param _

| FormatOb :
  forall (F: obCoMod _) (Param_F : obCoMod_Param _),
obCoMod_Param _

with obCoMod :
  forall _, Type :=

| ViewOb :
  forall G : obGenerator,
obCoMod _

| Viewing_Form_default :
  forall (F : obData  Param_SubstF)
  (* VV : forall paramlocal, Vision *),
obCoMod _

| ViewedOb_default :
  forall (F: obCoMod _) (Param_F : obCoMod_Param _),
obCoMod _

| Functions :
  forall (F: obCoMod _) (Param_F : obCoMod_Param _),
  forall (Param_FunctionsF : obCoMod_Param _),
  forall (PParam_SubstF : morCode_PParam Sense1_Param_proj
Sense1_Param_subst),
obCoMod _

with morCode_PParam :
```

```
  forall (Sense1_Param_subst_ff : Sense1_Param_def Sense01_Param_EF
Sense01_Param_F), Type :=


with morCode :

  forall (Sense1_Form_ff : Sense1_Form_def Sense1_FormParam_E
Sense1_FormParam_F

  Sense1_Param_proj_ff Sense1_Param_subst_ff), Type :=  .
```

coq40 / coq

4. Conversions equations in the algebra of the elements of the data-objects

As an example, it is possible to formulate that two grammatically-distinct constructors [True] and [Yes] with the same sense are in fact convertible (equal) in the algebra. Those equations on the data are the result of the resolution from the conversions in the logic. The final solution to decide these equations is left to some external solver such as GAP/SINGULAR.

coq50 / coq

```
Inductive convElAlgebra_Param :

  forall (cons_paramlocal : elAlgebra_Param Param_SubstF paramlocal),

  forall (cons_paramlocal' : elAlgebra_Param Param_SubstF paramlocal'), Type
:=

with convElAlgebra_Form :

  forall (cons_form : elAlgebra_Form F cons_paramlocal form ),

  forall (cons_form' : elAlgebra_Form F cons_paramlocal' form' ), Type :=


| TrueYes_convElAlgebra_Form :

convElAlgebra_Form (True_Binary_Form unitGenerator)

(Yes_Binary_Form unitGenerator).
```

coq50 / coq

5. Morphisms, whose conversions are to be logically decided or resolved to the algebra of the elements of data-objects

The clause [Constructing] is the embedding of data-objects elements as morphisms. The clause [Destructing] is the elimination from data-objects, which says that the generating constructors (supported on the given viewing/sieve) are sufficient to define some morphism from the data-object. The clauses [Hold] and [Eval] are the usual abstraction/discharge and evaluation/substitution of the

lambda calculus on functions. The clause [FormatMor] is context-extension from form-morphisms to parameter-morphisms. The clause [AdditionMor] is the algebra in the general logic-objects to be reduced into the algebra [Addition] of the data-objects elements.

coq60 / coq
```
Inductive morCoMod :
  forall E F (PParam_EF : morCode_PParam Sense1_Param_proj_ff
Sense1_Param_subst_ff)
  (Form_ff : morCode Sense1_Form_ff ), Type :=

| Compos :
  forall F' (ff' : 'CoMod( F' ~> F @_ PParam_F'F @^ Form_ff' )),
  forall (ff_ : 'CoMod( F'' ~> F' @_ PParam_F''F' @^ Form_ff_ )),
'CoMod( F'' ~> F @_ (Compos_morCode_PParam PParam_F'F PParam_F''F')
                @^ (Compos_morCode  Form_ff'  Form_ff_) )

| Unit :
  forall F,
'CoMod( F ~> F @_ _ @^ _ )

| ViewObMor :
  forall (G H : obGenerator) (g : 'Generator( G ~> H )),
'CoMod( ViewOb G ~> ViewOb H @_ _ @^ _ )

| Constructing_default :
  forall _ (* VV : forall paramlocal, Vision *),
  forall (F : obData Sense1_FormParam_F Sense1_Param_proj Param_SubstF),
  forall (G : obGenerator) (param : Sense00_Param_SubstF G)
  (cons_paramlocal : c.elAlgebra_Param (* VV *) Param_SubstF  (sval
Sense1_Param_subst G param) (InFibre G))
  (form : Sense00_AtParam_ Sense1_FormParam_F Sense1_Param_proj param)
  (cons_form : elAlgebra_Form (* VV *) F  cons_paramlocal form ),
'CoMod( ViewOb G ~> Viewing_Form_default F (* VV *)  @_ _ @^ _ )
```

```
| ViewedMor_default :
  forall  (ff : 'CoMod( E ~> F @_ PParam_EF @^ Form_ff )),
  forall (param_ff : 'CoMod__( Param_E ~> Param_F @_ PParam_EF' )),
'CoMod( ViewedOb_default E Param_E ~> ViewedOb_default F Param_F @_ _  @^ _ )

| UnitViewedOb_default :
  forall (ff : 'CoMod ( E ~> F @_ PParam_EF @^ Form_ff )),
'CoMod ( E ~> ViewedOb_default F Param_F  @_ _  @^ _ )

| Destructing_default :
  forall _ (* VV : forall paramlocal, Vision *).
  forall (param_ee_ :
    forall (G : obGenerator) (paramlocal : Sense00_Param_SumSubstF G)
    (P : obParametrizator) (inFibre_P : inFibre G P)
    (cons_paramlocal : c.elConstruct_Param (* VV *) Param_SubstF paramlocal
inFibre_P ),
    'CoMod__( ViewOb_Param P ~> Param_E @_ _ )),
  forall (ee__ :
    forall (G : obGenerator) (param : Sense00_Param_SubstF G)
    (cons_paramlocal : c.elConstruct_Param (* VV *) Param_SubstF (sval
Sense1_Param_subst G param) (InFibre G))
    (form : Sense00_AtParam_ Sense1_FormParam_F Sense1_Param_proj param)
    (cons_form : elConstruct_Form (* VV *) F cons_paramlocal form ),
    'CoMod( ViewOb G ~> E @_ _  @^ _)),
'CoMod( Viewing_Form_default F (* VV *) ~> ViewedOb_default E Param_E  @_ _
@^ _ )

| Eval :
  forall (param_eval : 'CoMod__( Param_FunctionsF ~> Param_F @_ PParam_SubstF
)),
```

```
  forall E (ee : 'CoMod( F ~> E @_  PParam_FE @^ Form_ee )),
'CoMod( Functions F Param_F Param_FunctionsF PParam_SubstF ~> E  @_ _ @^ _ )

| Hold :
  forall (param_eval : 'CoMod__( Param_FunctionsF ~> Param_F @_ PParam_SubstF
) ),
  forall L (ll : 'CoMod( L ~> F @_ PParam_LF @^ Form_ll )),
'CoMod( L ~> Functions F Param_F Param_FunctionsF PParam_SubstF @_ _ @^ _ )
where "''CoMod' ( E ~> F @_ PParam_EF @^ Form_ff )" := (morCoMod _ )

(* | AdditionMor: _ *)

with morCoMod_PParam :
  forall Param_E Param_F
  (PParam_EF : morCode_PParam Sense1_Param_proj_ff Sense1_Param_subst_ff),
Type :=

| FormatMor :
  forall (ff : 'CoMod( E ~> F @_  PParam_EF @^ Form_ff ))
  (param_ff : 'CoMod__( Param_E ~> Param_F @_ PParam_EF' )),
  forall Param_D (param_ee : 'CoMod__( Param_D ~> Param_E @_ PParam_DE )),
'CoMod__( Param_D ~> FormatOb F Param_F @_ _ )
where "''CoMod__' (  Param_E  ~>  Param_F  @_ PParam_EF )" :=
(morCoMod_PParam _).
```

coq60 / coq

6. Conversions, both logical conversions and resolution to the algebra-conversions

The clause [Constructing_cong] is the exit passage from logic to algebra, whose final decision is by some external solver such as GAP/SINGULAR. The clauses [Constructing_comp_Destructing] and [Hold_comp_Eval] are the usual cancellation conversions. Memo that any conversions in the forms-morphisms above may change the parameter-morphisms below; that is, conversions cause

reparameterization which must be tracked explicitly. Such reparameterization include structural reparameterizations such as the associativity of bracketing and its coherence.

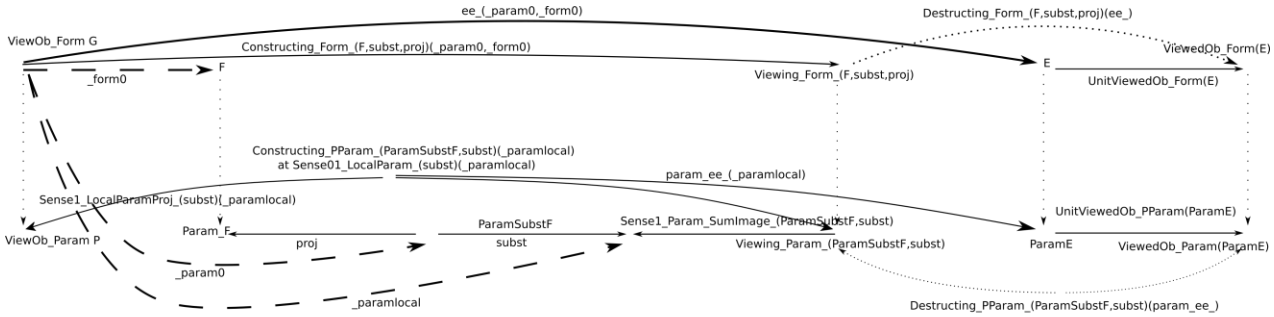
coq70 / coq

```
Inductive convCoMod :
  forall (PParam_EF : morCode_PParam _) (Form_ff : morCode _)
  (ff : 'CoMod( E ~> F @_ PParam_EF @^ Form_ff ))
  (ff0 : 'CoMod( E ~> F @_ PParam_EF0 @^ Form_ff0 )), Prop :=


| Constructing_default_cong :
  forall G param form  (cons_form : elConstruct_Form F param form),
  forall param' form'  (cons_form' : elConstruct_Form F param' form'),
  convElAlgebra_Form cons_form cons_form' ->
(Constructing_default cons_form') [ _ @^ _  ]<~~ (Constructing_default
cons_form)


| Constructing_default_comp_Destructing_default :
forall G param cons_paramlocal form cons_form ,


(UnitViewedOb_default Param_E ( ee__ G param cons_paramlocal form cons_form
))
  [ (Assoc_reparam _ _ _)   o>_$ _
  @^ (Assoc_congrMorCode _ _ _)  o>$ _ ]<~~
((Constructing_default (ElConstruct_elAlgebra_Form cons_form))
      o>CoMod ( Destructing_default param_ee_  ee__  ))


| Hold_comp_Eval :
forall (F : @obCoMod Sense00_Form_F Sense01_Form_F Sense00_Param_F
            Sense01_Param_F Sense1_FormParam_F)
Param_F Param_FunctionsF,
forall (param_eval : 'CoMod__( Param_FunctionsF ~> Param_F @_ PParam_SubstF )
),
forall L (ll : 'CoMod( L ~> F @_ PParam_LF @^ Form_ll )),
```

```
forall E (ee : 'CoMod( F ~> E @_  PParam_FE @^ Form_ee )),
   ( ll o>CoMod ee )
     [  (Assoc_reparam _ _ _) o>_$ _
         @^ (Assoc_congrMorCode  _  _  _)  o>$ _ ]<~~
   ( (Hold param_eval ll) o>CoMod (Eval param_eval' ee) )
where "ff0 [  reparam_rr @^ congr_ff  ]<~~ ff" := (convCoMod _)

with convCoMod_PParam :
  forall (PParam_EF : morCode_PParam _)
  (param_ff : 'CoMod__( Param_E ~> Param_F @_ PParam_EF ))
  (param_ff0 : 'CoMod__( Param_E ~> Param_F @_  PParam_EF0 )), Prop :=
| Constructing_PParam_default_comp_Destructing_PParam_default : _
where "param_ff0  [  reparam_rr  ]<~~__ param_ff" := (convCoMod_PParam _).
```

coq70 / coq

Another angle of view:



((UnitViewedOb_Form(E)) "<o" (ee_(_param0,_form0))) [isos]<~~
((Destructing_Form_(F,subst,proj)(ee_)) <o (Constructing_Form_(F,subst,proj)(_param0,_form0)))