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

Short: The goal of this publication is to remind potential contributors of the existence 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 roadmap, 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 path-induction J-rule for arrows" 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:

$$\vdash catB\colon Cat$$

$$X\colon catB \vdash F\colon catA$$

$$X\colon catB \vdash t\colon (F \underset{catA}{\longrightarrow} G)$$

What happens if we allow contexts to be some arrow-variable in some categorial-hom? Or contexts to be some element-variable of some more general profunctor-hom? Then natural transformations would be special cases of something when the domain is the unit profunctor-hom (that is, the hom of some category). This is the insight that leads Kosta Dosen to say that any ordinary natural transformation

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

can be formulated as an "*antecedental transformation*"

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

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

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

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

For reference, an adjunction with left adjoint functor $F\colon catB \to catA$, right adjoint functor $G\colon catA \to catB$, counit transformation $\phi_A\colon F\ G\ A \to A$, and unit transformation $\gamma_B\colon B \to G\ F\ B$ is formulated as rewrite rules from any redex outer cut on the left-side to the contractum containing some smaller inner cut:

$$f_2 \circ "f_1 \circ \phi" = "(f_2 \circ f_1) \circ \phi"$$

$$"\gamma \circ g_2" \circ g_1 = "\gamma \circ (g_2 \circ g_1)"$$

$$"f \circ \phi" \circ "\phi \circ F("\gamma \circ g")" = f \circ "\phi \circ Fg"$$

$$"G("f \circ \phi") \circ \gamma" \circ "\gamma \circ g" = "Gf \circ \gamma" \circ g$$

together with conversion (or rewrite) rules:

$$"\phi \circ F("Gg \circ \gamma")" = g$$

$$"G("\phi \circ Ff") \circ \gamma" = f$$

where indeed the functions on arrows $F\ -$ and $G\ -$ of those functors are absent from this formulation, but only their application on objects are implicit in the indices; and with this understanding, there are two more rewrite rules corresponding to functoriality, indirectly:

$$"\phi \circ F("\gamma \circ g_2")" \circ "\phi \circ F("\gamma \circ g_1")" = "\phi \circ F("\gamma \circ (g_2 \circ g_1)")"$$

$$"G("f_2 \circ \phi") \circ \gamma" \circ "G("f_1 \circ \phi") \circ \gamma" = "G("(f_2 \circ f_1) \circ \phi") \circ \gamma"$$

together with conversion (or rewrite) rules:

$$"\phi \circ F("\gamma \circ 1")" = 1;\ "G("1 \circ \phi") \circ \gamma" = 1$$

In practice, these cut-elimination techniques are only the kernel for some general contextual proof-assistant programming language which is more expressive. In fact, such antecedental/consequential transformations may be formulated systematically, similarly as the "J-rule" for equality/paths in (homotopy) type theory, if general

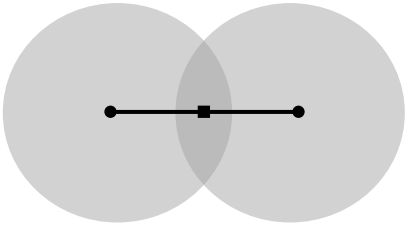
bimodule/profunctor-homs ($HomP : catA^{op} \times catB \to Set$) such as $(F \longrightarrow -)$ or $(- \longrightarrow G -)$ are expressible in the language. *And most importantly, because the general "J-rule" hide some cuts, therefore cut-elimination signify that the general "J-rule" must also be eliminated/admissible/computational.*

Moreover, the coYoneda lemma

$$\frac{A;\ B;\ P(A,B) \vdash Q(HA, KB)}{X;\ B;\ catX(X, H -) \otimes P(-, B) \vdash Q(X, KB)}$$

could be understood as some "J-rule"-in-context (context P), and becomes expressible if general bimodule/profunctor-homs constructions such as the tensor $\otimes$, that is, composable arrows: $\exists B. HomP(A,B) \times HomQ(B,C)$ are expressible. Also, the Yoneda lemma becomes expressible and follows from the coYoneda lemma in the presence of internal-implications, that is, functions on arrows: $\forall B. HomP(A,B) \to HomQ(C,B)$. Such tensor and implication biclosed-logic again constitute some adjunction and is therefore computational via Dosen cut-elimination techniques. Finally, topology/local operators and their sheaves and duality would become expressible in the presence of internal profunctors and comma/slice, that is, square of arrows: $\Sigma A\ B. HomP(A,B)$…

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



Approximately, starting with the exact sequence

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

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

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

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

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

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

gives the gluing operation

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

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

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

In practice, the implementation of the topology would be as some categorial site in the form of some local operator $j: \Omega \to \Omega$ where $\Omega(A)$ is the classifier of (sub-)objects (sieves) of the object A, and where $j_A(\mathcal{U})(f) \coloneqq f^*\mathcal{U} \in J(X)$ is the (opaque) set of witnesses that the pullback-sieve $f^*\mathcal{U}$ is covering (remember that the truthness that $\mathcal{U}$ is covering is expressed as $\mathcal{U} \in J(A)$, iff $\forall f.\, j(\mathcal{U})(f)$). But it is better to consider any presheaf in the slice over A, rather than only subfunctors because then everything is expressible as profunctor-homs (of witnesses) over some slice categories. Now the "J-rule" would be formulated to allow as output some sheafification modality and as input any family over some covering sieve, not only the singleton family over the generating identity-arrow of the trivial cover...

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/cartier/blob/master/cartierSolution12.v (Work in Progress)

In Word, search "Insert; Add-ins; WorkSchool 365" to play this Coq script.

C1 / coq

```
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.

Parameter Cat : Set.
Parameter Functor : Cat -> Cat -> Set.
Parameter Object : Cat -> Set.
Parameter Rel : Cat -> Cat -> Set.
Parameter Transf : forall C A B: Cat, Rel A B ->  Functor C A ->  Functor C B -> Set.
Parameter Adjunc : forall C D: Cat, Functor C D -> Functor D C -> Set.

Inductive object: forall (C : Cat), Type :=
| Gen_object : forall {C : Cat}, Object C -> object C
| App_object : forall {C D: Cat}, object C -> functor C D ->  object D

with functor: forall (C D : Cat), Type :=
| Gen_functor : forall {C D : Cat}, Functor C D -> functor C D
| Subst_functor : forall {C D E: Cat}, functor C D -> functor D E ->   functor C E
| Id_functor : forall C : Cat, functor C C

with rel: forall (C D : Cat), Type :=
| Gen_rel : forall (C D : Cat), Rel C D -> rel C D
| Tensor_rel : forall A B C, rel C B -> rel B A -> rel C A
| Id_rel :  forall C : Cat,  forall C' (F' : functor C' C), forall D' (G' : functor D' C),  rel C' D'
| Imply_rel : forall  A C B, rel A C -> rel B C -> rel B A
| ImplyCo_rel : forall  C A B, rel C A -> rel C B -> rel A B .

Inductive adjunc : forall C D: Cat, functor C D -> functor D C -> Type :=
| Gen_adjunc : forall (C D: Cat) (LeftAdjunc_functor : Functor C D) (RightAdjunc_functor : Functor D C),
  adjunc (Gen_functor LeftAdjunc_functor) (Gen_functor RightAdjunc_functor)

with arrow: forall A B: Cat, rel A B -> object A -> object B -> Type :=
| App_arrow : forall C A B: Cat, forall (R : rel A B) (F : functor C A) (G : functor C B),
  forall (X : object C), transf R F G -> arrow R (App_object X F) (App_object X G)

with transf:  forall C A B: Cat, rel A B -> functor C A -> functor C B -> Type :=
| Gen_transf : forall C A B: Cat, forall (F : Functor C A) (R: Rel A B) (G: Functor C B),
  Transf R F G -> transf (Gen_rel R) (Gen_functor F) (Gen_functor G)
| Id_transf : forall E C: Cat, forall (F : functor C E),  transf  (Id_rel F (Id_functor _) ) (Id_functor _)  F
| IdCo_transf : forall E C: Cat, forall (F : functor C E),  transf (Id_rel (Id_functor _) F)  F (Id_functor _)
| UnitAdjunc_transf : forall (C D: Cat) (LeftAdjunc_functor : functor C D) (RightAdjunc_functor : functor D C)
      (adj: adjunc LeftAdjunc_functor RightAdjunc_functor ),
```

```
    transf   (Id_rel (Id_functor _)
(RightAdjunc_functor) ) (Id_functor C)
(LeftAdjunc_functor)
| CoUnitAdjunc_transf : forall (C D:
Cat) (LeftAdjunc_functor : functor C D)
(RightAdjunc_functor : functor D C)
(adj: adjunc LeftAdjunc_functor
RightAdjunc_functor ),
    transf  (Id_rel (LeftAdjunc_functor)
(Id_functor _)) (RightAdjunc_functor)
(Id_functor _)
| App_transf : forall C D: Cat, forall
R : rel C D, forall  C' D' (F : functor
C C'), forall S : rel C' D', forall (G :
functor D D'),
            forall  A (M : functor A C)
(N : functor A D),
    transf R M  N -> funcTransf R S F G
-> transf S (Subst_functor M F)
(Subst_functor N G)

with funcTransf: forall C D: Cat, rel C
D -> forall C' D', rel C' D' -> forall
(F : functor C C') (G : functor D D'),
Type :=

| Subst_funcTransf : forall C D: Cat,
forall R : rel C D, forall  C' D' (F :
functor C C'), forall S : rel C' D',
forall (G : functor D D'),
  forall  C'' D'' (F' : functor C' C''),
forall T : rel C'' D'', forall (G' :
functor D' D''),
  funcTransf R S F G -> funcTransf S T
F' G' -> funcTransf R T (Subst_functor F
F') (Subst_functor G G')
| Id_funcTransf : forall C D: Cat,
forall R : rel C D,
    funcTransf R R (Id_functor _)
(Id_functor _)
| AnteceComp_funcTransf : forall C A B:
Cat, forall (F : functor C A)  (R: rel A
B)  (G: functor C B) ,
    transf R F G -> funcTransf (Id_rel G
(Id_functor B)) R F (Id_functor B)
| ConseqComp_funcTransf : forall C A B:
Cat, forall (F : functor C A)  (R: rel A
B)  (G: functor C B) ,
    transf R F G -> funcTransf (Id_rel
(Id_functor A) F) R (Id_functor A) G
| CoYoneda_anteceComp_funcTransf :
forall C D: Cat, forall D' (H : functor
D  D'), forall R : rel C D,
forall  C' (F : functor C C')  (T : rel
C' D'),
  funcTransf R T F H ->
  funcTransf (Tensor_rel R (Id_rel H
(Id_functor _))) T F (Id_functor _)
| CoYoneda_conseqComp_funcTransf :
forall C D: Cat, forall C' (H : functor
C C'), forall R : rel C D,
forall  D' (G : functor D D') (T : rel
C' D'),
  funcTransf R T H G ->
  funcTransf (Tensor_rel (Id_rel
(Id_functor _) H) R) T (Id_functor _) G
| Imply_lambda_funcTransf : forall C E
D: Cat, forall R : rel C E, forall (R' :
rel E D), forall  C' (F : functor C C'),
forall S : rel C' D,
  funcTransf (Tensor_rel R R') S F
(Id_functor _) ->
  funcTransf R (Imply_rel R' S) F
(Id_functor _)
| Imply_app_funcTransf : forall C E D:
Cat, forall R : rel C E, forall (R' :
rel E D), forall  C' (F : functor C C'),
forall S : rel C' D,
  funcTransf R (Imply_rel R' S) F
(Id_functor _) ->
  funcTransf (Tensor_rel R R') S F
(Id_functor _)
| Imply_eval_funcTransf : forall C E D:
Cat, forall R : rel E C, forall S : rel
D C, forall D' (F : functor D D') C'
(G : functor C C') (T : rel D' C'),
  funcTransf S T F G ->
  funcTransf (Tensor_rel (Imply_rel R S)
R) T F G
| Imply_pair_funcTransf : forall C E D:
Cat, forall R : rel C E, forall  (R' :
rel D C),
  forall D' (F : functor D' D) C' (G :
functor C' C) (T : rel D' C'),
  funcTransf T R' F G ->
  funcTransf T (Imply_rel R (Tensor_rel
R' R)) F G
| TODO_ImplyCo_lambda_funcTransf :
forall C E D: Cat, forall R : rel C E,
forall (R' : rel E D), forall  D' (G :
functor D D'), forall S : rel C D',
  funcTransf (Tensor_rel R R') S
(Id_functor _) G ->
  funcTransf R' (ImplyCo_rel R S)
(Id_functor _) G
| TensorCo_funcTransf : forall (C A B :
Cat) (R : rel A B) (G : functor C B)
(D : Cat) (S : rel D A) C' (H : functor
C' D) (S' : rel C' C),
  forall (F : functor C A), funcTransf
S' S H F -> funcTransf (Id_rel G
(Id_functor _) ) R F (Id_functor _) ->
  funcTransf S' (Tensor_rel S R) H G
| Tensor_funcTransf : forall (C A B :
Cat)   (R : rel A B)  C'' (G : functor
C'' B)  (R' : rel C C'')
  (D : Cat) (S : rel D A)  (H : functor
C D)  ,
  forall (F (* existential *): functor C
A), funcTransf (Id_rel (Id_functor _) H)
S (Id_functor _) F -> funcTransf R' R F
G ->
```

```
  funcTransf R' (Tensor_rel S R) H
G .
```