**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.

There is now sufficient evidence (ref [6], [7]) that Kosta Dosen's ideas and techniques (ref [1], [2], [3], [4], [5]) could be implemented for proof-assistants, sheaves and applications; in particular cut-elimination, rewriting and confluence for various enriched, internal, indexed or double categories with adjunctions, monads, negation, quantifiers or additive biproducts; quantitative/quantum linear algebra semantics; presheaf/profunctor semantics; inductive-sheafification and sheaf semantics; sheaf cohomology and duality...

The problem of “formulations of adjunction”, the problem of “unit objects” and also the problem of “contextual composition/cut” can be understood as the same problem. The question arises when one attempts to write precisely the counit/eval transformation ϵ\_X ∶ catA(F G X, X) where F : catB → catA is the left-adjoint. One could instead write ϵ\_X ∶ catA(F ,1)(G X, X) where the profunctor object/datatype catA(F ,1) is used in lieu of the unit hom-profunctor catA; in other words: F is now some implicit context, and the contextual composition/cut (in contravariant action) of some g:catB(Y, G X) in the unit profunctor against ϵ\_X should produce an element of the same datatype: ϵ\_X∘g : catA(F,1)(Y,X)... Ultimately Dosen-Petric (ref [5]) would be extended in this setting where some dagger compact closed double category, of left-adjoint profunctors across Cauchy-complete categories, is both inner and outer dagger compact closed where the dagger operation on profunctors (as 1-cells) coincides with the negation operation on profunctors (as 0-cells), optionally with sheaf semantics and cohomology. The earlier Coq prototypes (ref [6] for example) show that the core difficulty is in the industrial labor and tooling, which requires a coordinated workshop of workers (not celebrities, lol).

Recall that closed monoidal categories (with conjunction bifunctor ∧ with right adjoint implication functor →) are similar as programming with linear logic and types. Now to be able to express duality, finitely-dimensional/traced/compact closed categories are often used to require the function space (implication →) to be expressible in basis form. But along this attempt to express duality, two (equivalent) pathways of the world of substructural proof theory open up: one route is via Barr’s star-autonomous categories and another route is via Seely’s linearly distributive categories with negation.

For star-autonomous categories, one adds a “dualizing unit” object ⊥ which forces the evaluation arrow A ⊢ (A → ⊥) → ⊥ into an isomorphism. For linearly distributive categories with negation, one adds a “monoidal unit” object ⊥ for another disjunction bifunctor ∨ whose negation A'∨- is right-adjoint to the conjunction A∧- where this adjunction is expressed via the help of some new associativity rule A∧(B∨C) ⊢ (A∧B)∨C called “dissociativity” or “linear distributivity” (used to commute the context A∧ - and the negated context -∨C); and it is this route chosen by Dosen-Petric to prove most of their Gentzen-formulations and cut-elimination lemmas (ref §4.2 of [3] for linear, ref §11.5 of [2] for cartesian, and ref §7.7 of [2] for an introductory example). In summary, those are two routes into some problem of “unit objects” in non-cartesian linear logic.

Some oversight about the problem of “unit objects” is the belief that it should have any definitive once-for-all solution. Instead, this appears to be a collection of substructural problems for each domain-specific language. Really, even the initial key insight of Dosen-Petric, about the cut-elimination formulation in the domain-specific language of categorial adjunctions, can be understood as a problem of “unit profunctor” in the double category of profunctors and the (inner) cut elimination lemma becomes synonymous with elimination of the “J-rule”; and recall that such equality/path-induction J-rule would remain stuck in (non-domain-specific) directed (homotopy) type theory.

Now the many “formulation of adjunctions” are for different purposes. Indeed the outer framework (closed monoidal category) hosting such inner domain-specific language (unit-counit formulation) could itself be in another new formulation of adjunctions (bijection of hom-sets) where the implication bifunctor → is accumulated during computation via dinaturality (in contrast to the traditional Kelly-MacLane formulation).

Finally the problem of “contextual composition/cut” also arises from the problem of the structural coherence of associativity or dissociativity or commutativity, which now enables gluing the codomain/domain of compositions such as A∧f ∶ A∧X → A∧(B∨C) then g∨C ∶ (A∧B)∨C → Y∨C, or such as η\_A ∶ I → A'∧A then ϵ\_A∘f∧A' ∶ A∧A' → I, and which would force the outer framework to explicitly handle compositions under contexts/polycategories modulo associativity (“Gentzen cuts”) or to handle trace functions on arrows loops modulo cyclicity (for compact closed categories)... In other words, the meta framework (such as Blanqui’s LambdaPi and surprisingly not Coq’s CoqMT at present) of the framework should ideally implement those strictification lemmas (ref §3.1 of [2]) to be able to compute modulo structural coherence.

Here is an introduction to the insights. 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 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

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/admissibility of the composition (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 , right adjoint functor , counit transformation , and unit transformation is formulated as rewrite rules from any redex outer cut on the left-side to the contractum containing some smaller inner cut:

together with conversion (or rewrite) rules:

where indeed the functions on arrows and of those functors are not primitive but are themselves the “consequential transformation” formulations “” of the identity natural transformation…

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 () such as or 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

could be understood as some “J-rule”-in-context (context ), and becomes expressible if general bimodule/profunctor-homs constructions such as the tensor , that is, composable arrows: 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: . 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: …

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 local 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. 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**:

[1] Dosen-Petric: Cut Elimination in Categories 1999;

[2] Proof-Theoretical Coherence 2004;

[3] Proof-Net Categories 2005;

[4] Coherence in Linear Predicate Logic 2007;

[5] Coherence for closed categories with biproducts 2022

[6] <https://github.com/1337777/cartier/blob/master/cartierSolution12.v> (Work in Progress)

[7] Pierre Cartier

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

**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**.

**Inductive** functor: **forall** (C D : Cat), **Type** :=

| 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** :=

| Tensor\_antec\_rel' : **forall** [A B C B' : Cat], rel C B -> functor B B' -> rel B' A -> rel C A

| Id\_rel : **forall** [C C' D' : Cat], functor C' C -> functor D' C -> rel C' D'

| Imply\_antec\_rel' : **forall** [A C B C' : Cat], rel A C -> rel B C' -> functor C C' -> rel B A

| Subst\_rel : **forall** [C D C' D': Cat], rel C D -> functor C' C -> functor D' D -> rel C' D' .

**Inductive** adjunc : **forall** [C D: Cat], functor C D -> functor D C -> **Type** :=

with transf: **forall** [C A B: Cat], rel A B -> functor C A -> functor C B -> **Type** :=

| Restr\_transf (\* admissible \*): **forall** C A B: Cat, **forall** (R : rel A B) (F : functor C A) (G : functor C B),

**forall** D (X : functor D C), transf R F G -> transf R (Subst\_functor X F) (Subst\_functor X G)

| Id\_antec\_transf : **forall** E C: Cat, **forall** (F : functor C E), **forall** D (X : functor D C),

transf (Id\_rel F (Id\_functor **\_**) ) X (Subst\_functor X F)

| Id\_conse\_transf : **forall** E C: Cat, **forall** (F : functor C E), **forall** D (X : functor D C),

transf (Id\_rel (Id\_functor **\_**) F) (Subst\_functor X F) X

| UnitAdjunc\_transf : **forall** (C D: Cat) (LeftAdjunc\_functor : functor C D) (RightAdjunc\_functor : functor D C)

(adj: adjunc LeftAdjunc\_functor RightAdjunc\_functor ), **forall** A (X : functor A C),

transf (Id\_rel (Id\_functor **\_**) (RightAdjunc\_functor) ) X (Subst\_functor X (LeftAdjunc\_functor))

| CoUnitAdjunc\_transf : **forall** (C D: Cat) (LeftAdjunc\_functor : functor C D) (RightAdjunc\_functor : functor D C)

(adj: adjunc LeftAdjunc\_functor RightAdjunc\_functor ), **forall** A (X : functor A D),

transf (Id\_rel (LeftAdjunc\_functor) (Id\_functor D)) (Subst\_functor X (RightAdjunc\_functor)) X

| App\_transf : **forall** [C D C' D' A: Cat], **forall** [R : rel C D] [F : functor C C'] [S : rel C' D'] [G : functor D D'] [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 C' D': Cat], rel C D -> rel C' D' -> functor C C' -> functor D D' -> **Type** :=

| Subst\_funcTransf (\* admissible \*) : **forall** [C D C' D' C'' D'': Cat], **forall** [R : rel C D] [S : rel C' D'] [F : functor C C'] [G : functor D D']

[T : rel C'' D''] [F' : functor C C''] [G' : functor D D''],

funcTransf R S F G -> funcTransf (Subst\_rel S F G) T F' G' -> funcTransf R T F' G'

| Id\_funcTransf : **forall** C D: Cat, **forall** R : rel C D,

**forall** (C' D': Cat) (F : functor C' C) (G : functor D' D),

funcTransf (Subst\_rel R F G) R F G

| Restr\_funcTransf (\* admissible \*): **forall** C D: Cat, **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 S T F' G' -> funcTransf (Subst\_rel S F G) T (Subst\_functor F F') (Subst\_functor G G')

| Comp\_antec\_funcTransf' : **forall** C A B: Cat, **forall** (F : functor C A) (R: rel A B) (G: functor C B) ,

transf R F G -> **forall** B' (K: functor B' B), funcTransf (Id\_rel G K) R F K

| Comp\_conse\_funcTransf' : **forall** C A B: Cat, **forall** (F : functor C A) (R: rel A B) (G: functor C B) ,

transf R F G -> **forall** A' (K: functor A' A), funcTransf (Id\_rel K F) R K G

| CoYoneda\_antec\_funcTransf'' : **forall** (C D D' : Cat) (H : functor D D') (R : rel C D)

(C' : Cat) (F : functor C C') (T : rel C' D'),

funcTransf R T F H ->

**forall** (D0 : Cat) (K : functor D0 D'),

funcTransf (Tensor\_antec\_rel' R H (Id\_rel (Id\_functor D') K)) T F K

| CoYoneda\_antec\_appId\_funcTransf'' (\* OK version to derive ?? \*):

**forall** C D: Cat, **forall** D0 (H : functor D D0), **forall** R : rel C D,

**forall** C' (F : functor C C') D' (G : functor D D') (T : rel C' D'),

funcTransf (Tensor\_antec\_rel' R H (Id\_rel (Id\_functor **\_**) H)) T F G ->

funcTransf R T F G

| Imply\_antec\_app\_funcTransf'' : **forall** (C E D : Cat)

(C' : Cat) (F : functor C C') (S : rel C' D)

E' (E'E : functor E' E) (P : rel C E')

D' (D'D : functor D' D) (R : rel E D'),

funcTransf P (Imply\_antec\_rel' R S D'D) F E'E ->

funcTransf (Tensor\_antec\_rel' P E'E R) S F D'D

| Imply\_antec\_lambda\_funcTransf' (\* OK version for skew bif \*) : **forall** (C E D : Cat)

(C' : Cat) (F : functor C C') (S : rel C' D)

E' (E'E : functor E' E) (P : rel C E')

D' (D'D : functor D' D) (R : rel E D'),

funcTransf (Tensor\_antec\_rel' P E'E R) S F D'D ->

funcTransf P (Imply\_antec\_rel' R S D'D) F E'E

| Tensor\_antec\_funcTransf'' (\* OK version for skew bif \*) : **forall** (D A : Cat) (S : rel D A) C' (H : functor C' D) C (F : functor C A) (S' : rel C' C)

E E' (R : rel E E') (G : functor A E) A' (K : functor A' E') (R' : rel A A') ,

funcTransf S' S H F -> funcTransf R' R G K ->

funcTransf (Tensor\_antec\_rel' S' F R') (Tensor\_antec\_rel' S G R) H K

| Imply\_antec\_funcTransf''\_bif' (\* NOPE because contravariance ? \*): **forall** [A A0 C0 B C' : Cat] (R' : rel A0 C0) (S : rel B C') (K : functor C0 C')

B' (F : functor B B') C'' (G : functor C' C'') (S' : rel B' C'')

C (H : functor C C0) (R : rel A C),

funcTransf S S' F G (\* must G be id ? \*) -> **forall** (L : functor A A0), funcTransf R R' L(\* ?? must be id? more general lambda too; note this is contra now; or assume it is id only for cast\*) H ->

**forall** A1 (M : functor A1 A),

funcTransf (Imply\_antec\_rel' (Subst\_rel R' (Subst\_functor M L) (Id\_functor **\_**)) S K) (Imply\_antec\_rel' R S' (Subst\_functor H (Subst\_functor K G))) F M(\* must be id; or restrict R'? yeo, input M to restrict rr \*)

(\* from sol \*)

| Id\_antec\_Comp\_antec\_funcTransf'' : **forall** E C: Cat, **forall** (F : functor C E), **forall** A (X : functor A **\_**), **forall** A' (Y : functor A' **\_**),

funcTransf (Id\_rel (Subst\_functor X F) Y) (Id\_rel F (Id\_functor **\_**) ) X Y

| Id\_antec\_Comp\_conse\_funcTransf'' : **forall** E C: Cat, **forall** (F : functor C E), **forall** A (X : functor A **\_**), **forall** A' (Y : functor A' **\_**),

funcTransf (Id\_rel Y X) (Id\_rel F (Id\_functor **\_**) ) Y (Subst\_functor X F)

| Id\_conse\_Comp\_conse\_funcTransf'' : **forall** E C: Cat, **forall** (F : functor C E), **forall** A (X : functor A **\_**), **forall** A' (Y : functor A' **\_**),

funcTransf (Id\_rel Y (Subst\_functor X F)) (Id\_rel (Id\_functor **\_**) F) Y X

| UnitAdjunc\_Comp\_antec\_funcTransf'' (\* bad \*): **forall** (C D: Cat) (LeftAdjunc\_functor : functor C D) (RightAdjunc\_functor : functor D C)

(adj: adjunc LeftAdjunc\_functor RightAdjunc\_functor ), **forall** A (X : functor A C), **forall** B (Y : functor B D),

funcTransf (Id\_rel (Subst\_functor X (LeftAdjunc\_functor)) Y) (Id\_rel (Id\_functor **\_**) (RightAdjunc\_functor) ) X Y

| CoUnitAdjunc\_Comp\_antec\_funcTransf'' : **forall** (C D: Cat) (LeftAdjunc\_functor : functor C D) (RightAdjunc\_functor : functor D C) (adj: adjunc LeftAdjunc\_functor RightAdjunc\_functor ),

**forall** A (X : functor A **\_**), **forall** B (Y : functor B **\_**),

funcTransf (Id\_rel X Y ) (Id\_rel (LeftAdjunc\_functor) (Id\_functor **\_**)) (Subst\_functor X (RightAdjunc\_functor)) Y

| CoUnitAdjunc\_Comp\_conse\_funcTransf'' (\* bad \*) : **forall** (C D: Cat) (LeftAdjunc\_functor : functor C D) (RightAdjunc\_functor : functor D C) (adj: adjunc LeftAdjunc\_functor RightAdjunc\_functor ),

**forall** A (X : functor A C), **forall** B (Y : functor B D),

funcTransf (Id\_rel X (Subst\_functor Y (RightAdjunc\_functor)) ) (Id\_rel (LeftAdjunc\_functor) (Id\_functor **\_**)) X Y .