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

Short: The goal of this publication is to remind potential contributors 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 itinerary, 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 arrow-induction" 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. What happens if we allow contexts under some arrow-variable? Or contexts under some element-variable of some general profunctor-hom? Then natural transformations would be special cases of something when the domain is the unit profunctor-hom (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*"

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

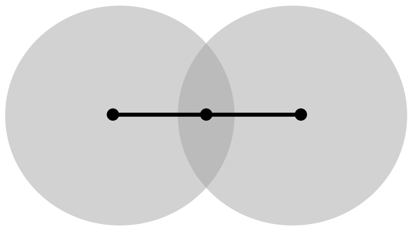
with primitive name "$H - \circ t\_$" in the language, or can be formulated as a "*consequential transformation*"

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

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 of the composition $\circ$ (cut-elimination). Of course, this cut elimination is except those (apparent) cuts baked into the primitive language of the antecedental/consequential counit and unit; nevertheless, the decidability of the equality of the arrows still holds via the confluence lemma.

In practice, these cut-elimination techniques are only the kernel for some general contextual proof-assistant programming language which is more expressive. For example, while the surface language would allow expressions in non-empty contexts such as functors or natural elements/transformations or antecedental/consequential transformations and composition/substitutions of those, the target-compilation language is concerned with decisions only on the resulting objects and arrows (with already-applied functors and transformations on them). Another example is that the surface language may allow general profunctor-hom constructions ($Hom_1\colon \mathrm{catX}^{\mathrm{op}} \times \mathrm{catY} \to \mathrm{Set}$) such as pairs of composable arrows (via the tensor profunctor-hom $\exists B.\, Hom_1(A, B) \times Hom_2(B, C)$), or functions on arrows (via the cotensor profunctor-hom $\forall B.\, Hom_1(A, B) \to Hom_2(C, B)$)), or pairs of parallel arrows (via the product profunctor-hom $Hom_1(A, B) \times Hom_2(A, B)$), or square of arrows (via the comma of the profunctor-hom $\Sigma A\ B.\, Hom_1(A, B)$), or user-opaque profunctor-hom variables. Then the cut-elimination would, at least, still traverse those expressions.

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

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

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 closure operator $j: \Omega \to \Omega$ where $\Omega(A)$ is the classifier of (sub-)objects (sieves) of the object A, and where $j(\mathcal{U})(f) \coloneqq f^*\mathcal{U} \in J(A)$ 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.

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/dosen/blob/master/dosenSolution1.v