Kosta Dosen simplicial-cubical functorial programming for ω-categories and schemes, implemented in Lambdapi

This is the continuation of an ongoing research programme of discovering a truly computational logic (Lambdapi type theory) for categories, profunctors, fibred categories, univalence, polynomial functors, sites, sheaves and schemes, and for the simplicial-cubical homotopy of omega-categories, in the style of Kosta Dosen's book « Cut-elimination in categories » (1999).

How do you program a proof-assistant language without using variables names? Answer: you use a "context-extension" operation (ref. "categories with families", "comprehension categories", etc.) which is internalized (reflected/computational/strictified) into the language itself. So that syntactically there is only "one" (compound/telescope) variable in the context.

How do you program a higher-dimensional omega-categories proof-assistant language without using arbitrary-long (infinite) number of meta-grammar entities for 1-arrows, 2-arrows, 3-arrows, etc.? Answer: you use a "comma/arrow-category" operation, together with the "context-extension" (now rebranded as "total category") operation, to internalize/reflect back the comma/arrow dependent-category (fibration) as a base-category. So that syntactically a 2-arrow is simply an arrow in the comma/arrow category.

There is a "dependent comma/arrow" dependent-category (fibration) construction F if given a dependent-category (fibration) $E \to B$ as input; besides the usual comma dependent-category. Now such F would be dependent/fibred over two bases: as $F \to E$ over the total category E of the dependent-category $E \to B$, and as $F \to \text{arrow}(B)$ over the comma/arrow category $\text{arrow}(B)$ of the base-category B.

For the sheaves and schemes, a glue operation for any sheaf `S` over the sheafification-closure modality `mod_smod` is declared.

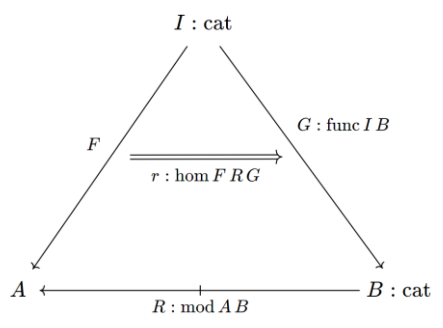
arrow f ∈ (sheafification-closure of sieve S) ⟷ (pullback sieve of S along f) ∈ site-topology

```
constant symbol glue : Π [A B : cat] (A_site : site A)  [S : smod A_site B] [I : cat] [G : func I B] (L : mod A
I), transf L Id_func (smod_mod S) G  →  transf (smod_mod (mod_smod A_site L)) Id_func (smod_mod S) G;
```

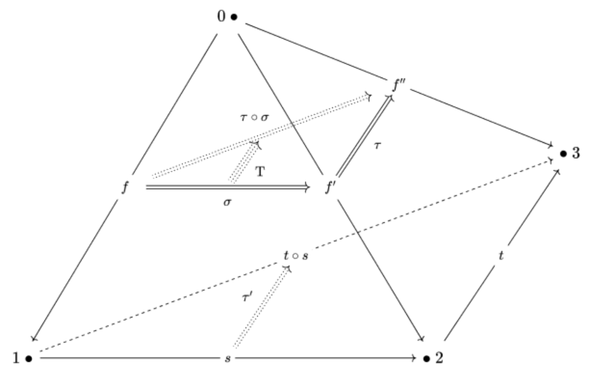
Then, there is an implementation of locally ringed sites and schemes, whose slice-categories satisfy the (computational) interface/specification of an affine scheme. But the affine-scheme interface is coinductive (self-reference), meaning that its slice-categories are also required to satisfy the affine-scheme interface. Moreover, the invertibility-support D(f) for a locally ringed site is defined as the sieve of opens where the function f is invertible; and this sieve becomes generated by a singleton when restricted to a slice affine-scheme.

Also, this implementation is already able to compute that 1+2=3 via 3 different methods: the category of natural-numbers as a higher inductive type; the natural-numbers object inside any fixed category; and the colimits inside the category of finite sets/numbers. Indeed, this new functorial programming language, also referred as Dosen's « m— » or « emdash » or « modos », is able to express the usual logic such as the tensor and internal-hom of profunctors, the sigma-sum and pi-product of fibred categories/profunctors; but is also able to express the concrete and inductively-constructed categories/profunctors, to express the adjunctions such as the product-and-exponential or the constant-diagram-and-weighted-limit adjunctions within any fixed category, to express contravariance and duality such as a computational-proof that right-adjoints preserve limits from the dual statement, to express groupoids and univalent universes, to express polynomial functors as bicomodules in the double category of categories, functors, cofunctors and profunctors, etc. Article's source: https://github.com/1337777/cartier/blob/master/cartierSolution17.lp

```
constant symbol cat : TYPE;  constant symbol func : Π (A B : cat), TYPE;
constant symbol mod : Π (A B : cat), TYPE;
constant symbol hom : Π [I A B : cat], func I A → mod A B → func I B → TYPE;
injective symbol transf [A' B' A B: cat] (R' : mod A' B') (F : func A' A) (R : mod A B) (G : func B' B) : TYPE;
```



I : cat
F
G : func I B
r : hom F R G
A
B : cat
R : mod A B



0
f''
τ ∘ σ
τ
3
T
f
f'
σ
t ∘ s
t
τ'
1
s
2