Kosta Dosen's functorial programming

This article implements Kosta Dosen's functorial programming that 1+2=3 via 3 different methods: the natural numbers category via categories-as-types (and dependent types are fibrations of categories), the natural numbers object inside any fixed category via adjunctions/product/exponential, and the category of finite sets/numbers via colimits inductively computed from coproducts and coequalizers. Such extremely convoluted roundtrip between concrete data structures and the abstract-nonsense of the double category of fibred profunctors is necessary for the goal of the logical specification of algorithms and their theorem proving; for example the usual list/vector tail becomes specified as a fibrational transport or functor over the natural numbers category. This new functorial programming language will now be referred as Dosen's « m— » or « emdash » or « modos ». The basis for this implementation is the ideas and techniques from Kosta Dosen's book « Cut-elimination in categories » (1999), which essentially is about the substructural logic of category theory, in particular how some good substructural formulation of the Yoneda lemma allows for computation and automatic-decidability of categorial equations. This article makes progress on future implementations: how to integrate functorial programming proof-assistants with higher groupoidal symmetry (homotopy types, univalent higher categories) and with polynomial algebra (polynomial monads, databases, effects, and dynamics). Finally, in today's digital landscape, a developer writing an AI prompt that orchestrates an interface among various agents/tools/plugins API is akin to an academic author writing a scientific article: therefore, the ability-or-not of proof-and-AI-assistants to intelligently use or search within an interfacing prompt or article is a new form of editorial review; and is prologue to any eventual (expert) peer "reviewing" (i.e., coauthoring) of a byproduct article that cites the original article (this government-auditable methodology is implemented at: editoReview.com via the OpenAI GPT store and via the Microsoft Copilot Studio and SharePoint). Article's source: https://github.com/1337777/cartier/blob/master/cartierSolution15.lp

• Categories, functors, profunctors, hom-arrows, transformations… organize into a *double category* of (fibred) profunctors, where categories are basic and manipulated from the outside via functors $F : I \to C$ instead of via their usual objects "$F : \mathrm{Ob}(C)$". Then there are the usual compositions/whiskering and their units/identities. Lambdapi code (only loss of generality vs C++ is that 2-ary, 3-ary functors… would be manual):

```
constant symbol cat : TYPE;

constant symbol func : Π (A B : cat), TYPE;

constant symbol mod : Π (A B : cat), TYPE;

constant symbol hom_Set : Π [I A B : cat], func I A → mod A B → func I B → Set;

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 ≔ τ (@transf_Set A' B' A B R' F R G);

symbol ∘> : Π [A B C: cat], func A B → func B C → func A C;

symbol ∘>> : Π [X B C: cat], func C X → mod X B → mod C B;

constant symbol ⊗ : Π [A B X : cat], mod A B → mod B X → mod A X;

symbol ∘↓ : Π [I A B I' : cat] [R : mod A B] [F : func I A] [G : func I B],

 hom F R G → Π (X : func I' I), hom (X ∘> F) R (G <∘ X);

symbol '∘ : Π [A B' B I : cat] [S : mod A B'] [T : mod A B] [X : func I A] [Y : func I B']  [G : func B' B],

 hom X S Y → transf S Id_func T G → hom X T (G <∘ Y);

symbol ''∘ [ B'' B' A B : cat] [R : mod A B''] [S : mod A B'] [T : mod A B] [Y : func B'' B'] [G : func B' B]
:  transf R Id_func S Y → transf S Id_func T G →  transf R Id_func T (G <∘ Y);
```

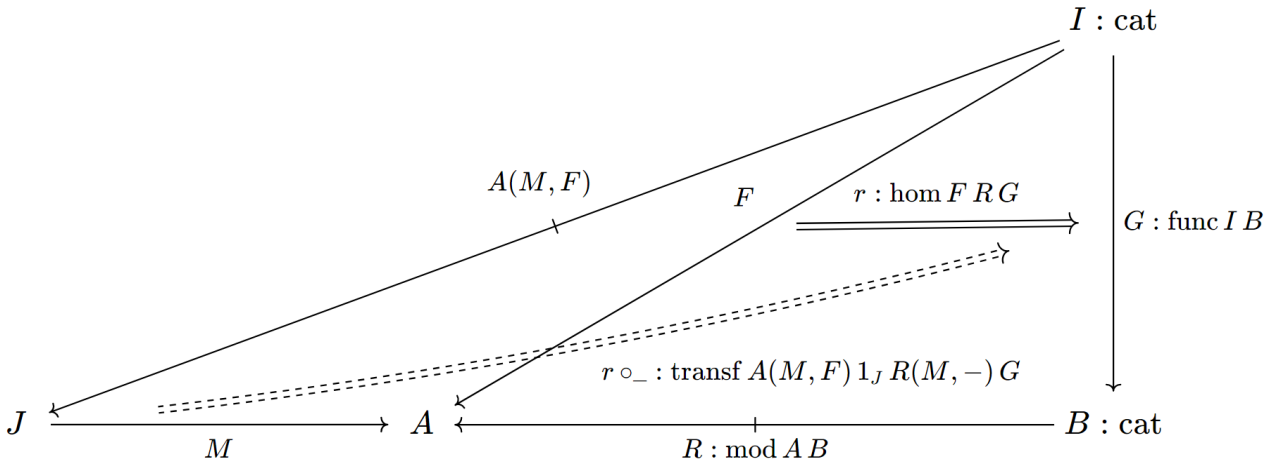
• But the usual inner composition/cut inside categories

$$\forall A\ B\ C : \mathrm{Ob}(R),\ \mathrm{hom}_R(B,C) \to \mathrm{hom}_R(A,B) \to \mathrm{hom}_R(A,C),$$

instead, is assumed directly as the *Yoneda "lemma", by reordering quantifiers*:

$$\forall B\ C : \mathrm{Ob}(R),\ \mathrm{hom}_R(B,C) \to \big(\forall\ A : \mathrm{Ob}(R), \mathrm{hom}_R(A,B) \to \mathrm{hom}_R(A,C)\big),$$

and using the *unit category-profunctor* so that any *hom-element/arrow* becomes, via this Yoneda "lemma", also a *transformation* from the unit profunctor.




```
constant symbol Unit_mod : Π [X A B : cat], func A X → func B X → mod A B;

injective symbol _'∘> : Π [I A B J : cat] [F : func I A] [R : mod A B] [G : func I B], Π (M : func J A),
  hom F R G → transf (Unit_mod M F) Id_func (M ∘>> R) G;

injective symbol ∘>'_ : Π [I A B J : cat] [F : func I A] [R : mod A B] [G : func I B],
  hom F R G → Π (N: func J B), transf (Unit_mod G N) F (R <<∘ N) Id_func;
```

• Besides implementing (higher) inductive datatypes such as the *join-category* (interval simplex) and *concrete categories* (e.g., the natural-numbers category), with their introduction/elimination/computation rules; this article also implements the natural-numbers object internally to any fixed type/category, and its addition arrow via the product/exponential adjunction:

```
constant symbol inat_func (C : cat) : func (Terminal_cat) C;

constant symbol Zero_inj_inat_hom (C : cat) : hom (itermin_func C) (Unit_mod Id_func (inat_func C)) Id_func;

constant symbol Succ_inj_inat_hom (C : cat) : Π [C0] [X0 : func C0 C] [I] [X: func I C0] [Y : func I _],

 hom X (Unit_mod X0 (inat_func C)) Y → hom X (Unit_mod X0 (inat_func C)) Y;

symbol add_inat_hom C : hom (Product_pair_func (inat_func C) (inat_func C)) (Unit_mod (iprod_func C)
(inat_func C)) Id_func ≔

// ... 1 + 2 ≡ Succ_inj_inat_hom C (Succ_inj_inat_hom C (Succ_inj_inat_hom C (Zero_inj_inat_hom C)))
```

• A fundamental explanation of polynomials comes from the *dualities in the many ways to store the data info of a category*: for example, the codomain object could be understood either as indexing the arrows or as an operation together with the composition operation. All the data of a polynomial functor $p : d\text{-Set} \to c\text{-Set}$, written as $p : c \nleftarrow d$ or $p : c\text{-Set}[d]$,

$$p := C \mapsto \sum_{i \in \mathrm{Ob}(p)(C)} d\text{-Set}\left(p[i], \underline{y}\right), \text{ or shorter } p := \sum_{i \in p} \underline{y}^{p[i]} \text{ where } \underline{y}^S := y \mapsto y^S := \mathrm{Set}(S, \_)$$

is stored inside a single profunctor module between d and the *category of elements* of some $p : c\text{-Set}$:

$$p[-,\_] : \left(\int^{C:c} p(C)\right) \nleftarrow d$$

And the (ongoing) goal of functorial programming here is to express the interface of the polynomial operations (composition/substitution, coclosure, local monoidal closure, etc.) by manipulating only these underlying data of diagrams and profunctors without referring explicitly/directly to any synthetic (semantic) concepts such as the "category of elements". For example, the composition of two polynomials becomes:

```
constant symbol pmod_cov : Π [A : cat] (PA : mod Terminal_cat A) (B : cat), TYPE;

constant symbol ◃pmod_cov : Π [A B C : cat] [PA : mod Terminal_cat A] (R : pmod_cov PA B),

Π [PB : mod Terminal_cat B] (S : pmod_cov PB C), pmod_cov ((PB ⇇pmod_cov R) ×pmod (Proj_pmod_cov PA)) C;
```