Kosta Dosen's functorial programming

Abstract. This article implements Kosta Dosen's functorial programming that 1+2=3 via 3 different methods: the natural numbers category via categories-as-types, 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, 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 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. Article's source: https://github.com/1337777/cartier/blob/master/cartierSolution14.lp

TABLE OF CONTENTS

1 Introduction: Goal.

• The *goal* is to implement a programming language and proof-assistant where the types are categories, and the functions are functors. The surrounding data environment for those categories, instead of being based on sets, could be based on *higher groupoids* (i.e., sets whose elements have intrinsic symmetry) or could be based on *polynomials* (i.e., sets whose elements are container for elements of another parameter set).

• The first subgoal is to reevaluate the meaning of editorial review of scientific articles in the context of proof assistants and AI assistants. A developer writing an AI prompt that orchestrates an interface among various tools/plugins API is akin to an academic author penning a scientific article. The upshot is that 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 is made possible by a new research in AI called "*Dynamic chaining*" of "agents" or external tools (such as search engines, calculators, or weather sensors): Yao (2023) "*ReAct: Synergizing Reasoning and Acting in Language Models*", Wu (2023) "*AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation*". An implementation of this methodology is at: editoReview.com [in OpenAI GPT store and Microsoft Copilot Studio]

• This article closes the open problem of *implementing a dependent-types computer for category theory*, where types are categories and dependent types are fibrations of categories. The basis for this implementation are the ideas and techniques from *Kosta Dosen's book [1] « Cut-elimination in categories » (1999)*, which essentially is about the substructural logic of category theory, in particular about how some *good substructural formulation of the Yoneda lemma* allows for computation and automatic-decidability of categorial equations. An implementation of this methodology is at: https://github.com/1337777/cartier/blob/master/cartierSolution14.lp **[Todo: Live Demo in Microsoft Visual Studio]**

2 Introduction: Motivation 1.

• Composition of functions and associativity normalization: e ◦ f ◦ g ◦ h

CHOICE A? always ((e ◦ f) ◦ g) ◦ h
CHOICE B? always e ◦ (f ◦ (g ◦ h))

• Problem with *computation rules*, for *pairing-projections* or for *case-injections*

projectFirst ◦ pair$\langle g, g' \rangle = g$; projectSecond ◦ pair$\langle g, g' \rangle = g'$

case$[f|f']$ ◦ injectLeft $= f$; case$[f|f']$ ◦ injectRight $= f'$

• Attempt for left-associativity normalization CHOICE A:

$\big((d \circ e) \circ \text{projectFirst}\big) \circ \text{pair}\langle g, g' \rangle) \circ h$ ✗ KO, unless "accumulate/yoneda" trick to control associativity:

$\ldots = (\text{"}(d \circ e) \bullet \text{projectFirst"} \circ \text{pair}\langle g, g' \rangle) \circ h = d \circ e \circ g \circ h.$

$\big((e \circ \text{case}[f|f']) \circ \text{injectLeft}\big) \circ h = \big((\text{case}[e \circ f|e \circ f']) \circ \text{injectLeft}\big) \circ h = e \circ f \circ h$

✓ OK by naturality.

- Attempt for right-associativity normalization CHOICE B:

$e \circ (\text{projectFirst} \circ (\text{pair}\ \langle g, g' \rangle \circ h)) = \ e \circ (\text{projectFirst} \circ (\text{pair}\ \langle g \circ h, g' \circ h \rangle)) = e \circ g \circ h$ OK by naturality.

$e \circ \Big(\text{case}[f|f'] \circ \big(\text{injectLeft} \circ (h \circ i)\big)\Big)$ ✗ KO, unless "accumulate/yoneda" trick to control associativity:

$\ldots = e \circ (\text{case}[f|f'] \circ \text{"injectLeft} \bullet (h \circ i)\text{"}) \ = \ e \circ f \circ h \circ i$.

3 Introduction: Motivation 2.

• How to write the (co)unit transformation ϵ of an adjunction between a left adjoint functor $F: D \to C$ and right adjoint functor $G: C \to D$? Memo: the notion of adjoint functors is a generalization of the notion of inverse functions, and the counit is similar as a projection and the unit is similar as an injection, with similar computation rules as in the preceding section.

CHOICE A: $\epsilon_X: C(FGX, X)$ where X is any *variable*.

CHOICE B: $\epsilon_X: C[F, -](GX, X)$ where $C[F, -] : D^{\mathrm{op}} \times C \to \mathrm{Set}$ is profunctor.

CHOICE C: $\epsilon_X: C[FG, -](X, X)$.

CHOICE D: $\epsilon_X^H: C[FG, H](HX, X)$ where H is an extra *parameter*; also, CHOICE B' with extra parameter, etc.

BAD CHOICE: $\epsilon_X: C[FGX, -](-, X)$.

• Kosta Dosen's key insight is that the "accumulated/yoneda" trick version for CHOICE B therefore becomes the usual hom-bijection formulation of adjunction/inverse:

the (contravariant) accumulating operation:

for $f: C[X, -](1, X')$, get "$f \circ \epsilon_X$": $C[F, -](GX, X')$

the (covariant) accumulating operation:

for $g: D[-, GX](Y, 1)$, get "$\epsilon_X \circ g$": $C[F, -](Y, X)$

4 Introduction: Tools, source literature and raw data.

• A *profunctor* (also named *distributor* or *module*) ϕ covariant over a category C and contravariant over a category D, written

$$\phi: C \nrightarrow D \quad \text{or} \quad \phi: D \nleftarrow C,$$

is defined to be a functor in the usual sense from the product category:

$$\phi: D^{\mathrm{op}} \times C \to \mathrm{Set}$$

where D^{op} denotes the *opposite category* of D and Set denotes the *category of sets*. Given morphisms $f: d' \to d, g: c \to c'$ respectively in D, C and an element $x \in \phi(d, c)$, we write $f; x = xf \in \phi(d', c)$ and $x; g = gx \in \phi(d, c')$ to consecutively denote the *contravariant action* and the *covariant action*.

• The *tensor or composite* $\psi \otimes \phi : E \nleftarrow C$ (sometimes written as $\psi \circ \phi$ or $\psi \lhd \phi$) of two profunctors

$$\psi: E \nleftarrow D \text{ and } \phi: D \nleftarrow C$$

is given by a *coend* formula (because the quantified variable d occurs both covariantly and contravariantly), which is equivalently:

$$(\psi \otimes \phi)(e, c) = \int^{d:D} \psi(e, d) \times \phi(d, c) = \left(\coprod_{d \in D} \psi(e, d) \times \phi(d, c) \right) \Big/ \sim$$

where $\sim$ is the least equivalence relation such that

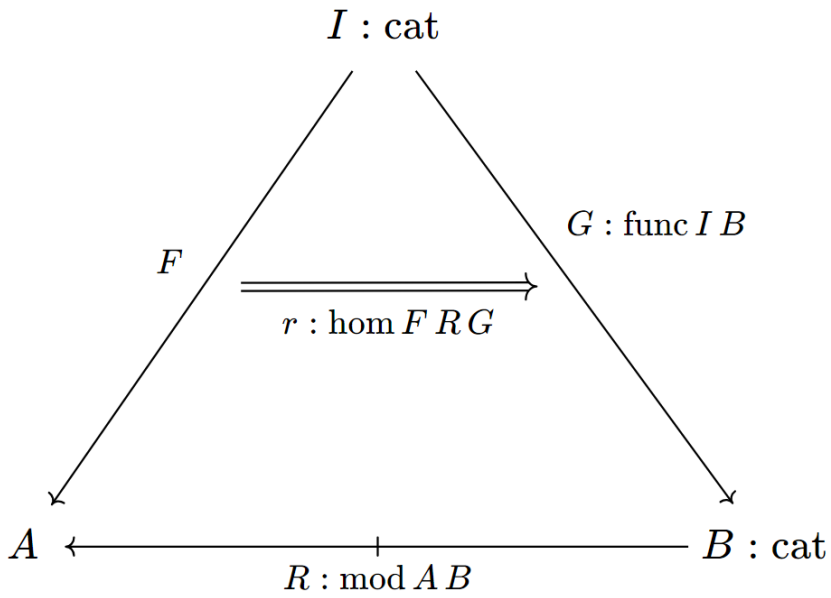
$$(y; f, x) \sim (y, f; x)$$

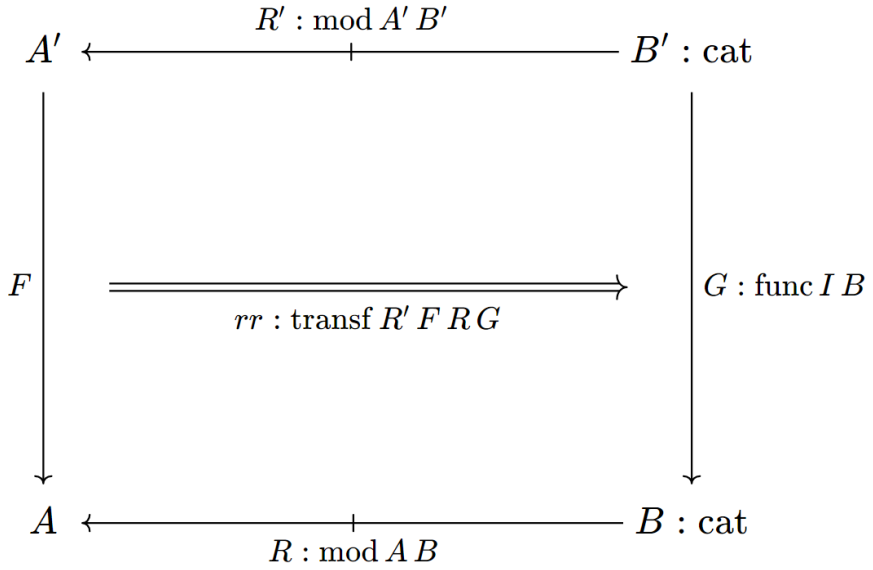
for any $f: d \to d'$ in D, and $y \in \psi(e, d)$, and $x \in \phi(d', c)$.

5 Results: Categories, functors, profunctors, hom-arrows, transformations…

These 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:Ob(C)". This is outlined in the following specially formatted *emdash* m—functorial programming source code:

```
constant symbol cat : TYPE;

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

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

constant symbol hom_Set : ∏ [I A B : cat], func I A ⟶ mod A B ⟶
func I B ⟶ Set;

injective symbol hom [I A B : cat] (F : func I A) (R : mod A B)
(G : func I B): TYPE
  ≔ τ (@hom_Set I A B F R G );

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);
```


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


R' : mod A' B'
A'
B' : cat
F
G : func I B
rr : transf R' F R G
A
B : cat
R : mod A B

6 Composition is Yoneda “lemma”.

• There are the usual compositions/whiskering and their units/identities.

```
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);
```

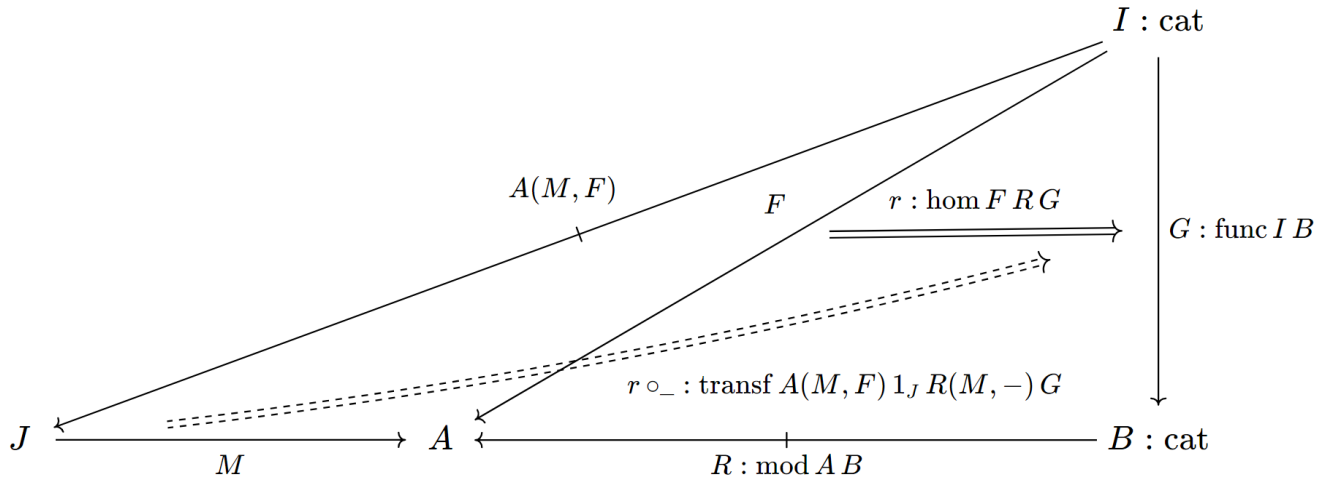
• But the usual inner composition/cut inside categories

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

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

$$\forall B\ C : \mathrm{Ob}(R),\ \hom_R(B,C) \to \left(\forall\ A : \mathrm{Ob}(R), \hom_R(A,B) \to \hom_R(A,C)\right),$$

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;
```

21 Applications: datatypes or 1+2=3 via 3 methods: nat numbers category, nat numbers object and colimits of finite sets.

• The goal of this section is to demo that it is possible to do a *roundtrip* between the concrete data structures and the abstract prover grammar; this is very subtle. The concrete application of these datatypes is the computation with the addition function of two variables that 1+2=3 via 3 different methods: the natural numbers category via intrinsic types, the natural numbers object via adjunctions/product/exponential, and the category of finite sets/numbers via limits/colimits/coproducts.

• This article also implements (higher) inductive datatypes such as the *join-category* (interval simplex) and *concrete categories*, with their introduction/elimination/computation rules. (In the updated file, https://github.com/1337777/cartier/blob/master/cartierSolution14.lp)

• *Concrete categories datatypes*, such as the *category of finite sets* or the *category of natural numbers*, are presented within the *abstract prover grammar* via *datatypes*. Now datatypes are higher types because they allow constructors for arrows, besides constructors for objects. Besides there are also *Concrete functors/objects datatypes* such as the *natural numbers object* internal to any particular category.

• The key to discover the correct formulation is to understand the terminal category also as a datatype, and thereafter use this *terminal datatype*'s primitives to formulate the other more-complex datatypes.

• The natural numbers category, and addition functor via intrinsic types:

```
constant symbol nat_cat : cat;

constant symbol Zero_inj_nat_func : func Terminal_cat nat_cat;

constant symbol Succ_inj_nat_func :  ∏ [I], func I nat_cat → func I nat_cat;

symbol add_nat_func : func (Product_cat nat_cat nat_cat) nat_cat
≔

compute ((Product_pair_func (Succ_inj_nat_func (Succ_inj_nat_func Zero_inj_nat_func)) (Succ_inj_nat_func Zero_inj_nat_func)) ◦> add_nat_func);

// ≡ Succ_inj_nat_func (Succ_inj_nat_func (Succ_inj_nat_func Zero_inj_nat_func))
```

• The natural numbers object, and addition arrow via 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)))
```

• The category of finite sets/numbers, and addition cocone via coproducts/colimits/limits: The goal of this section is to demo that it is possible to do a *roundtrip* between the concrete data structures and the abstract prover grammar; this is very subtle. This new approach allows, not only to compute with concrete data, but also to do so via a *grammatical interface* which is more strongly-specified/typed and which enables the theorem proving/programming of the correctness-by-construction of the algorithms, such as the usual *algorithm to inductively compute general finite limits/colimits* from the equalizer limits, product limits and terminal limits.

• This new approach is to be contrasted for example from the AlgebraicJulia library package, which is an attempt to add "functional language" features to the Julia numerical computing language, via category theory. This applied category theory on concrete data structures allows to achieve some amount of compositionality (function-based) features onto ordinary numerical computing. The AlgebraicJulia implementation essentially hacks and reimplements some pseudo-dependent-types domain-specific-language embedded within Julia.

• This demo now successfully works generically, including on this silly example: the limit/equalizer of a (inductive) diagram when the (inductive-hypothesis) product cone **[12;11]×[22;21]×[33;32;31]** now is given an extra constant arrow **[22;21]** → **[33;32;31]** **onto 31**, besides its old discrete base diagram.

The output limit cone's apex object:

```
compute obj_category_Obj ((category_Obj_obj One) ◦>o (sigma_Fst
(construct_inductively_limit_instance_liset _ example_graph_isf
example_diagram)));
// (0,13,21,31) :: (0,13,22,31) :: (0,12,21,31) :: (0,12,22,31)
:: (0,11,21,31) :: (0,11,22,31) :: nil
//Pair_natUniv (Pair_natUniv (Pair_natUniv (Base_natUniv 0)
(Base_natUniv 13)) (Base_natUniv 21)) (Base_natUniv 31) :: …
```

The output limit cone's side arrows:

```
compute arr_category_Arr ((Eval_cov_hom_transf ((sigma_Snd
(construct_inductively_limit_instance_liset _ example_graph_isf
example_diagram))₁ )) ◦a' ( (@weightprof_Arr_arr _ _ _
(category_Obj_obj One) (graph_Obj_obj (Some (Some None))) One))
);
// λ x, natUniv_snd (natUniv_fst (natUniv_fst x))
```

The output limit cone's universality operation:

```
compute arr_category_Arr (((((sigma_Snd
(construct_inductively_limit_instance_liset _ example_graph_isf
example_diagram))₂ ) _ _ _ example_cone) ◦>'_ ( _ ) ) ◦a'
(Id_cov_arr (category_Obj_obj One))) liset_terminal_natUniv;
// Pair_natUniv (Pair_natUniv (Pair_natUniv (Base_natUniv 0)
(Base_natUniv 12)) (Base_natUniv 22)) (Base_natUniv 31)
```

22 Polynomial comodules and modules

• A fundamental explanation of polynomials comes from the *dualities in the many ways to store the data info of a category*:

- The domain-codomain indexing of the arrows by the objects could be structural/outer and thereafter augmented by the composition operation (and the identity-arrow operation).
- Or alternatively only the domain indexing is structural/outer and then the codomain becomes an operation together with the composition operation; or dually only the codomain indexing is outer.
- Or one could forget arrows and remember only the *underlying groupoid/set* (of isomorphisms/equalities); or one could forget the codomain and composition operations and remember only the outer domain indexing of the arrows (i.e., spoiler: *underlying polynomial*) …

- Alternative formulations:

$$A \in c^{\mathrm{op}}, B \in c \mapsto c[A,B] \quad : c^{\mathrm{op}} \times c \to \mathrm{Set} \text{ (“profunctor”)}$$

$$A \in c \mapsto y \in c\text{-Set} \mapsto y^{c[A,-]} \quad : c \to c\text{-Set} \to \mathrm{Set} \text{ (“polynomial”)}$$

$$y \in c\text{-Set} \mapsto A \in c \mapsto y^{c[A,-]} \quad : c\text{-Set} \to c\text{-Set} \text{ (“prafunctor”)}$$

$$B \in c^{\mathrm{op}} \mapsto y \in c^{\mathrm{op}}\text{-Set} \mapsto y^{c[-,B]} \quad : c^{\mathrm{op}} \to c^{\mathrm{op}}\text{-Set} \to \mathrm{Set} \text{ (contravariant)}$$

$$B \in c \mapsto y \in c\text{-Set} \mapsto y \otimes c[-,B] \quad : c \to c\text{-Set} \to \mathrm{Set} \text{ (linear, left adjoint)}$$

The notation c-Set denotes $c \to \mathrm{Set}$.

• In general, the *hom* in the superscript $y^{c[A,\_]} := d\text{-Set}(c[A,\_], y)$ could itself be replaced by a profunctor/relation $r\colon c^{\mathrm{op}} \times d \to \mathrm{Set}$ producing $y^{r[A,\_]}$; or even more generally it could be replaced by any profunctor $p\colon \left(\int^{A:c} F(A)\right)^{\mathrm{op}} \times d \to \mathrm{Set}$ over the *category of elements* $\int^{A:c} F(A) = \mathrm{El}(F) = (A\colon c) \times F(A)$ of another diagram $F\colon c\text{-Set}$ (equivalently understood as *discrete* op-fibration $\int^{A:c} F(A) \to c$), and thus producing the only meaningful (discrete) summation:

$$A \in c \mapsto y \in d\text{-Set} \mapsto \sum_{x:F(A)} y^{p[(A,x),\_]}$$

This is precisely the definition of a *polynomial* (and they behave like the usual algebra's polynomials).

• Moreover, the Kan extensions $\Sigma \vdash \Delta \vdash \Pi$ semantics of

$$c \xleftarrow{\pi_c} \int^{A:c} F(A) \xleftarrow{\pi_F} \int^{A:c,x:F(A),D:d} p[(A,x),D] \xrightarrow{\pi_d} d$$

produced by the projection functors out of the category of elements of p coincide with the prafunctor semantics of the polynomial (notation: $\int_{D:d}$ … below denotes the *end set* while $\int^{C:c}$… denotes the *coend set* but it sometimes also denotes the *category* of elements):

$$y \in d\text{-Set} \mapsto A \in c \mapsto \Sigma_{\pi_c} \circ \Pi_{\pi_F} \circ \Delta_{\pi_d}(y)(A)$$

$$= \operatorname*{colim}_{x:F(A'),a:A'\to A} \left(\lim_{D:d,P:p[(A'',x'),D],\_:(A',x)\to(A'',x')} y\big(\pi_d(D,P)\big) \right)$$

$$= \sum_{x:F(A)} \lim_{D:d,P:p[(A,x),D]} y(D)$$

$$= \sum_{x:F(A)} \int_{D:d} \mathrm{Set}\big(p[(A,x),D], y(D)\big) = \sum_{x:F(A)} d\text{-Set}(p[(A,x),\_], y) = \sum_{x:F(A)} y^{p[(A,x),\_]}$$

• Recall, from Spivak's (2023) "*Functorial Aggregation*", that given a set S, the corresponding representable functor (the $\underline{y}$oneda of S) is denoted by $\underline{y}^S$ (i.e., *low dash* y):

$$\underline{y}^S := y \mapsto y^S := \mathrm{Set}(S, \_) := x \mapsto \mathrm{Set}(S, x) : \mathrm{Set} \to \mathrm{Set}$$

Most often, the category Set of sets can be generalized to any category c-Set of diagrams (or copresheaves/presheaves) over another category c, and a bicomodule polynomial is any functor $p : d\text{-}\mathrm{Set} \to c\text{-}\mathrm{Set}$, written as $p : c \nleftarrow d$ or $p : c\text{-}\mathrm{Set}[d]$, which is some sum of representables; i.e., there exists

- some set/diagram $\mathrm{Ob}(p) \in c\text{-}\mathrm{Set}$ (covariant over c), and
- some sets/diagrams $p[i] \in d\text{-}\mathrm{Set}$ (each covariant over d), for each $C \in c$ and $i \in \mathrm{Ob}(p)(C)$, which is contravariant in the variable (C, i) over the category of elements of $\mathrm{Ob}(p)$, and producing an assignment which is therefore covariant over c:

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

• In other words, all the data of this polynomial is stored inside a single profunctor module between d and the *category of elements* of some $p : c$-Set:

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

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, and by doing such sub-structurally without referring explicitly/directly to any synthetic (semantic) concepts such as the “category of elements”.

• For example, it would *not be computationally enough* to simply "write" the composition of two polynomials $m : c \nleftarrow d$ and $n : d \nleftarrow e$ as:

$$m \lhd_d n := \sum_{i \in m} \sum_{j \in d\text{-Set}(m[i], n)} \underline{y}^{\operatorname{colim}_{x \in m[i]} n[j(x)]} : c \nleftarrow e$$

or to write the $(\_ \lhd_d n)$-coclosure of two polynomials $m : c \nleftarrow e$ and $n : d \nleftarrow e$ as:

$$\begin{bmatrix} n \\ m \end{bmatrix} := \sum_{i \in m} \underline{y}^{\sum_{j \in n} e\text{-Set}(n[j], m[i])} : c \nleftarrow d$$

Instead, this interface specification must be expressed in the functorial programming style (approximately, the pair (i, j) in $\lhd_d$ above appears via $\times\mathrm{pmod}$ below):

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

constant symbol pmod_con : Π (A : cat) [B : cat] (PB : mod B
Terminal_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;
```

• An *applied use of polynomials is in the querying of databases*, for example: a polynomial $p : d\text{-Set} \to \text{Set}$, where the category $d := \text{"City} \to \text{Staty} \leftarrow \text{County"}$, might take as input a d-Set diagram $y := \mathit{cities} \to \mathit{states} \leftarrow \mathit{counties}$ (that is, a set of cities, a set of states and a set of counties) and produce this set: $\mathit{cities} \times_{\mathit{states}} \mathit{cities} + \mathit{cities} + \mathit{states}$ (i.e., disjoint union of pairs of cities in the same state, union all the cities, union all the states) via the formula:

$$p := \underline{y}^{p[1]} + \underline{y}^{p[2]} + \underline{y}^{p[3]}$$

where $p[1]: d\text{-Set} := \text{City} \mapsto \{\star, \star\star\}, \text{Staty} \mapsto \{\star\}, \text{County} \mapsto \{\}$, etc.

23 Discussion: Whether results conclude goal?

• Each of the sub-goals listed in the introduction is essentially concluded by the results above. The most significant result is the computation that 1+2=3 via 3 different methods: the natural numbers category via categories-as-types, 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. This result reuses all the other preceding results and concepts. By the nature that the results are computer implementations, they have no absolute-mistake. Now the peer-reviewer may have the opinion for a different specification of the goal, relative to which these results are no longer correct nor complete; therefore the next section will quiz-test whether the reviewer have at least paid attention to learn the author's original specification as summarized there.

• The ultimate goal is not concluded yet, for example the sum Sigma-type and product Pi-type implementation for profunctors (sets), besides those for categories, has been skipped; also some details about the logical-properties content of concrete limits, besides their data content, have been skipped. The future sub-goals should focus on how the surrounding data-environment for categories, instead of being based on sets, could be based on higher groupoids or could be based on polynomial-functors.

• The potential scope and implications of this goal and its results extend to computations in engineering applications such as strongly-specified compositional graph rewriting, or database management, or open dynamical systems, and subsume ordinary functional programming. This goal opens the possibility for the general public to communicate and peer-review such computer-implemented mathematics, in a larger market.

24 Discussion: Qualifier for editorial review.

• As outlined in the *Introduction* section, the ability-or-not of proof-and-AI-assistants to intelligently search within a scientific 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. That is, the proof-AI-search is considered as a qualified reader of the article, as specified by an editorial of qualifier queries (i.e., "natural/difficult knowledge") that should be successfully answered by the proof-AI-search at the interface of the article in the context of the literature data. In other words, the editorial reviewer is responsible to specify/design a judgmental/personal editorial of qualifier queries that should be successfully answered by the proof-AI-search. But prior to being entitled to such an opinionated editorial, the editorial reviewer should themselves qualify against some (more basic) quiz questions which test whether the reviewer have at least paid attention to learn the author's original qualifier-specification as summarized in this section. An implementation of this methodology is at: editoReview.com **[Todo: Short Live Demo in OpenAI GPT store and Microsoft Copilot Studio]**

• **Q1.** This article specifies that which functorial programming operation is more primitive?

(A) The composition of two arrows inside a category.
(B) The Yoneda action of a category's arrows onto a profunctor elements.
(C) The addition functor defined on the natural numbers category.

Q1 ; 30 / quiz Click or tap here to enter text.

25 References

[1] Kosta Dosen, Zoran Petric. *Cut Elimination in Categories*. (1999)
https://www.bing.com/search?q=Kosta+Dosen+Cut+Elimination+in+Categories+1999
[2] Kosta Dosen, Zoran Petric. *Proof-Theoretical Coherence*. (2004)
https://www.bing.com/search?q=Kosta+Dosen+Proof+Theoretical+Coherence+2004
[3] Kosta Dosen, Zoran Petric. *Proof-Net Categories*. (2007)
https://www.bing.com/search?q=Kosta+Dosen+Proof-Net+Categories+2007
[4] Kosta Dosen, Zoran Petric. *Coherence in Linear Predicate Logic*. (2007)
https://www.bing.com/search?q=Kosta+Dosen+Coherence+in+Linear+Predicate+Logic+2007
[5] Kosta Dosen, Zoran Petric. *Coherence for closed categories with biproducts*. (2020)
https://www.bing.com/search?q=Zoran+Petric+Coherence+for+closed+categories+with+biproducts+2020
[6] Christopher Mary. *Cut-elimination in the double category of fibred profunctors with inner cut-eliminated adjunctions*. (2010)
https://github.com/1337777/cartier/blob/master/cartierSolution13.lp
[7] Christopher Mary. *Applications: datatypes or 1+2=3 via 3 methods: natural numbers category via categories-as-types, natural numbers object via adjunctions, and category of finite sets/numbers via colimits*. (2010)
https://github.com/1337777/cartier/blob/master/cartierSolution14.lp
[8] Pierre Cartier