

Dosen functorial programming

Christopher Mary, Camille Nous

Laboratoire Cogitamus
camille.nous@cogitamus.fr

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>

Keywords: Kosta Dosen, proof assistants, AI assistants, functorial programming, polynomial functors, homotopy types

TABLE OF CONTENTS

- | | |
|---|---|
| 1. Introduction: Goal. | 13. Pi-category-of-fibred-functors and Sigma-category. |
| 2. Introduction: Motivation 1. | 14. And why profunctors (of sets)? |
| 3. Introduction: Motivation 2. | 15. What is a fibred profunctor anyway? |
| 4. Introduction: Tools, source literature and raw data. | 16. Higher inductive types, the interval type, concrete categories. |
| 5. Results: Categories, functors, profunctors, hom-arrows, transformations... | 17. Universe, universal fibration. |
| 6. Composition is Yoneda “lemma”. | 18. Weighted limits. |
| 7. Outer cut-elimination or functorial lambda calculus. | 19. Duality Op, covariance vs contravariance. |
| 8. Inner cut-elimination or decidable adjunctions. | 20. Grammatical topology. |
| 9. Synthetic fibred Yoneda. | 21. Applications: datatypes or $1+2=3$ via 3 methods: nat numbers category, nat numbers object and colimits of finite sets. |
| 10. Substructural fibred Yoneda. | 22. Discussion: Whether results conclude goal? |
| 11. Comma elimination (“J-rule arrow induction”). | 23. Discussion: Qualifier for editorial review. |
| 12. Cut-elimination for fibred arrows. | 24. References. |

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 a parameter set). This ultimate goal has some *sub-goals* which are listed below in the form of the achievements of the results in this article.

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, in itself, 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 or user of the article or tooling, as specified by a (personal) editorial of qualifier queries (i.e., “natural/difficult knowledge”, “natural/complex usages”) that should be successfully answered or performed by the proof-AI-search at the interface of the article or tooling in the context of the literature data or market (i.e., other tools) data. In the AI world, the fusion of “reasoning” capabilities and external tools (such as search

engines, calculators, or Wikipedia lookup) has given birth to “agents,” as explained in Yao (2023) “*ReAct: Synergizing Reasoning and Acting in Language Models*” (<https://arxiv.org/abs/2210.03629>). Wu (2023) “*AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation*” (<https://arxiv.org/abs/2308.08155>) explores the concept of multiple “agents”. OpenAI’s “GPTs”, Microsoft’s Bing Copilot plugin, and Google’s Bard Extensions are prime examples of this innovation. This article, about proof assistants based on foundational mathematics, pays attention to this long-term subgoal, and even includes an experimental *Discussion* section to anticipate such novel Editorial Review challenges.

Category theory (<https://www.bing.com/search?q=category+theory+prefer:mathematics>) is a general theory of mathematical structures and their relations. In category theory, a category is formed by two sorts of things: the *objects* of the category, and the *morphisms*, which relate two objects called the *source* and the *target* of the morphism. One often says that a morphism is an *arrow* that maps its source to its target. Morphisms can be *composed* if the target of the first morphism equals the source of the second one, and morphism composition has similar properties as function composition (associativity and existence of identity morphisms).

The second fundamental concept of category theory is the concept of a *functor*, which plays the role of a morphism between two categories. It maps objects of one category to objects of another category and morphisms of one category to morphisms of another category in such a way that sources are mapped to sources and targets are mapped to targets.

A third fundamental concept is a *natural transformation* that may be viewed as a morphism of functors.

Functional programming is a programming paradigm where programs are constructed by applying and composing functions. It is a declarative programming paradigm in which function definitions are trees of expressions that map values to other values, rather than a sequence of imperative statements which update the running state of the program.

In functional programming, functions between data types are treated as primitives, meaning that they can be bound to names (including local identifiers), passed as arguments, and returned from other functions, just as any other data type can. That is, functions become elements of the *function/exponential type*. This allows programs to be written in a declarative and composable style, where small functions are combined in a modular manner.

In functional programming, each data type is understood as a set of elements and each function is understood as a mapping of the elements from one set to the elements of another set.

Imagine a novel programming language where each type becomes a category instead of a mere set, and where each function between types becomes a functor between these *categories-as-types*? This is the goal of Kosta Dosen’s *functorial programming*; and it is more subtle than it sounds...

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.

The core of dependent types/fibrations in category theory is the Lawvere’s comma/slice construction and the corresponding Yoneda lemma for fibrations (<https://stacks.math.columbia.edu/tag/0GWH>), thereby its implementation essentially closes this open problem also investigated by Cisinski’s directed types or Garner’s 2-dimensional types. What qualifies as a solution is subtle and ***the thesis here is that Dosen’s substructural techniques cannot be bypassed***.

In summary, this article implements, using *Blanqui’s LambdaPi metaframework software tool*, an (outer) cut-elimination in the double category of fibred profunctors with (inner) cut-eliminated adjunctions.

This implementation of the outer cut-elimination essentially is a *new functorial lambda calculus via the « dinaturality » of evaluation* and the monoidal bi-closed structure of profunctors, without need for multicategories because (outer) contexts are expressed via dependent types.

This article also implements (*higher*) *inductive datatypes* such as the join-category (interval simplex), with its introduction/elimination/computation rules.

This article also implements *Sigma-categories/types* and categories-of-functors and more generally *Pi-categories-of-functors*, but an alternative more-intrinsic formulation using functors fibred *over spans* or *over Kock’s polynomial-functors* will be investigated.

This article also implements *dualizing Op operations*, and it can computationally-prove that left-adjoint functors preserve profunctor-weighted colimits from the proof that *right-adjoint functors preserve profunctor-weighted limits*.

This article also implements a *grammatical (directed-univalent) universe* and the universal fibration classifying small fibrations, together with the dual universal opfibration.

Finally, there is an experimental implementation of *covering (co)sieves towards grammatical sheaf cohomology* and towards a description of algebraic geometry's schemes in their formulation as *locally affine ringed sites* (structured topos), instead of via their Coquand's formulation as underlying topological space...

2 Introduction: Motivation 1.

Composition of functions and associativity normalization: $e \circ f \circ g \circ h$

CHOICE A?	always	$((e \circ f) \circ g) \circ h$
CHOICE B?	always	$e \circ (f \circ (g \circ h))$

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

$$\begin{aligned} \text{projectFirst} \circ \text{pair}\langle g, g' \rangle &= g; & \text{projectSecond} \circ \text{pair}\langle g, g' \rangle &= g' \\ \text{case}[f|f'] \circ \text{injectLeft} &= f; & \text{case}[f|f'] \circ \text{injectRight} &= f' \end{aligned}$$

Attempt for left-associativity normalization CHOICE A:

$$\begin{aligned} ((d \circ e) \circ \text{projectFirst}) \circ \text{pair}\langle g, g' \rangle \circ h & \quad \times \text{ KO, unless "accumulate/yoneda" trick to control associativity:} \\ \dots = ("(d \circ e) \bullet \text{projectFirst}" \circ \text{pair}\langle g, g' \rangle) \circ h &= d \circ e \circ g \circ h. \\ ((e \circ \text{case}[f|f']) \circ \text{injectLeft}) \circ h &= ((\text{case}[e \circ f|e \circ f']) \circ \text{injectLeft}) \circ h = e \circ f \circ h \quad \checkmark \text{ OK by naturality.} \end{aligned}$$

Attempt for right-associativity normalization CHOICE B:

$$\begin{aligned} e \circ (\text{projectFirst} \circ (\text{pair}\langle g, g' \rangle \circ h)) &= e \circ (\text{projectFirst} \circ (\text{pair}\langle g \circ h, g' \circ h \rangle)) = e \circ g \circ h \quad \text{OK by naturality.} \\ e \circ (\text{case}[f|f'] \circ (\text{injectLeft} \circ (h \circ i))) & \quad \times \text{ KO, unless "accumulate/yoneda" trick to control associativity:} \\ \dots = e \circ (\text{case}[f|f'] \circ \text{injectLeft} \bullet (h \circ i)) &= e \circ f \circ h \circ i. \end{aligned}$$

3 Introduction: Motivation 2.

How to write the (co)unit transformation ϵ of an adjunction between a left adjoint functor $F: D \rightarrow C$ and right adjoint functor $G: C \rightarrow D$? 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 $C[F, -]: D^{\text{op}} \times C \rightarrow \text{Set}$ 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.

Tools: The methodology for these results is a large-scale-integrated concrete implementation and engineering into the computer, of many smaller-scale semantically-well-developed mathematical concepts but understood through Dosen's insights and techniques. This methodology is novel in the sense that the traditional researchers who are expert at implementation tools often lack the knowledge of these Kosta Dosen references; and the traditional researchers who are expert at Kosta Dosen references often lack the knowledge of these implementation tools.

The precise tool is the logical framework [Lambdapi](#) by Frederic Blanqui. It allows to quickly prototype the type systems of new logic-or-programming languages, without worrying about reimplementing the low-level tasks such as syntax with binders and substitution, implicit arguments and metavariables, or unification and constraint. This logical framework approach contrasts from the approach of reimplementing such new type systems from scratch using the language C++ for example; however it has the limitation that automatic full generality is no longer possible: for example, (pro-)functors of 4, or 3, or 2 arguments would each be manually implemented besides functors of 1 argument (which is not a significant problem when the arity is bounded in the particular application of interest).

LambdaPi is a proof assistant based on the $\lambda\Pi$ -calculus modulo theory. It is an interactive proof system that features dependent types like in Martin-Löf's type theory, but allows defining elements and types using *oriented equations*, aka *rewriting rules*, and reasoning modulo those equations. One methodology with this LambdaPi tool consists in attempting any concrete computation such as $1+2=3$, investigating where the computation gets stuck, and then *reverse engineering* to discover the lacking rewrite rules (for the non-trivial interactions which happen at the large-scale integration of the many smaller-scale micro-theories).

Another tool used is the *Visual Studio Code* IDE which makes very readable the Lambdapi source code of this implementation, even more readable than (vertical-aligned) Latex formulas; therefore, it is strongly-understood that this full source code is an appendix which is an integral part of this ongoing article document, no less than the usual mathematical Latex code.

Source literature: Most traditional source literature (<https://www.bing.com/search?q=category+theory+prefer:mathematics>) about category theory are either about *categorical algebra* or *categorical logic*. For example, categorical algebra would understand a category similarly as an abstract algebra's *ring* and understand a "profunctor" as a *module of this ring*. Another example: categorical logic would understand functional programming as happening within a fixed category such that the types are the objects and the functions are the morphisms, within this category.

Many alternative themes and debates, such as Dosen's approach to categorical logic [1] « *Cut-elimination in categories* » (1999), are either overlooked or refused attribution. Kosta Dosen's and Zoran Petric's work in this area culminates with the article [5] « *Coherence for closed categories with biproducts* » (2022), which is an attempt at a *substructural logic* within a dagger compact closed (double-)category (of left-adjoint profunctors across Cauchy-complete categories).

Recall that closed monoidal (double-)categories (with conjunction bifunctor \wedge with right adjoint implication functor \rightarrow) 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 \rightarrow) 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 \perp which forces the evaluation arrow $A \vdash (A \rightarrow \perp) \rightarrow \perp$ into an isomorphism. For linearly distributive categories with negation, one adds a "monoidal unit" object \perp for another disjunction bifunctor \vee whose negation $A' \vee -$ is right-adjoint to the conjunction $A \wedge -$ where this adjunction is expressed via the help of some new associativity rule $A \wedge (B \vee C) \vdash (A \wedge B) \vee C$ called "dissociativity" or "linear distributivity" (used to commute the context $A \wedge -$ and the negated context $- \vee C$); 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] « *Proof-Net Categories* » (2005) for linear; ref §11.5 of [2] « *Proof-Theoretical Coherence* » (2004) 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. And 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.

An earlier attempt at categorical abstract machines (1986) and categorical datatypes (1987) by Curien-Hagino contains a subtle bug... Another attempt by Cisinski at categorical/directed homotopy types lacks the ability to compute $1+2=3$... Nevertheless their (non-constructive) higher groupoidal/homotopical layer about univalence and symmetry is in the progress of being copied here. Finally, the attempt by Spivak via polynomial functors (<https://www.bing.com/search?q=spivak+polynomial+type+theory+prefer:mathematics>) raises the open question of how the hybrid polynomial-profunctorial programming should be.

Raw data: The raw data collection process consists in *paying attention and serendipitously discovering* obscure references such as Kosta Dosen monographs books, and connecting them with mathematical and non-mathematical knowledge from diversified sources using a search engine such as Microsoft Bing (<https://www.bing.com/search?q=prefer:mathematics>). Some samples from this raw data are copied as-is in the next few paragraphs:

A *profunctor* (also named *distributor* by the French school and *module* by the Sydney school) ϕ from a category C to a category D , written

$$\phi: C \rightrightarrows D,$$

is defined to be a functor

$$\phi: D^{\text{op}} \times C \rightarrow \text{Set}$$

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

The *tensor or composite* $\psi\phi$ of two profunctors

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

is given by a *coend* formula, which is equivalently:

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

where \sim is the least equivalence relation such that $(y', x') \sim (y, x)$ whenever there exists a morphism v in D such that

$$y' = vy \in \psi(e, d') \text{ and } x'v = x \in \phi(d, c).$$

There is a bicategory or *double-category* **Prof** whose 0-cells are small categories, (horizontal) 1-cells between two small categories are the profunctors between those categories, (vertical) 1-cells between two small categories are the functors between those categories, and 2-cells between two profunctors are the natural transformations between those profunctors. And the theory of adjunctions, monads, weighted limits carry over to this setting.

In type theory, a system has *inductive types* if it has facilities for creating a new type from constants and functions (*constructors*) that create terms of that type. In particular, *W-types* are well-founded inductive types in *intuitionistic type theory*. They generalize natural numbers, lists, binary trees, and other “tree-shaped” data types. Let U be a *universe of types*. Given a type $A : U$ and a *dependent family* (i.e., *fibration*) $B : A \rightarrow U$, one can form a W-type $W_{a:A}B(a)$. The type A may be thought of as “labels” (or parameters) for the (potentially infinitely many) constructors of the inductive type being defined, whereas B indicates the (potentially infinite) inductive *arity* of each constructor. For example, one may define *lists* over a type $A : U$ as

$$\begin{aligned} \text{List}(A) &:= W_{(x:1+A)}B(x), \\ B(\text{inl}(1_1)) &= 0, \\ \text{forall } a, \quad B(\text{inr}(a)) &= 1, \end{aligned}$$

where 1_1 is the sole inhabitant of $\mathbf{1}$. The value of $B(\text{inl}(1_1))$ corresponds to the 0-ary constructor *nil* for the empty list, whereas the value of $B(\text{inr}(a))$ corresponds to the 1-ary constructor *cons* that appends a to the beginning of another (argument) list.

The constructor for elements of a generic W-type $W_{x:A}B(x)$ has the default name *sup* and the form

$$\frac{a:A \quad f:B(a) \rightarrow W_{x:A}B(x)}{\text{sup}(a, f): W_{x:A}B(x)}$$

The *elimination rule* for W-types works similarly to *structural induction* on trees. If, whenever a property (under the *propositions-as-types* interpretation) $C: W_{x:A}B(x) \rightarrow U$ holds for all subtrees of a given tree it also holds for that tree, then it holds for all trees. This elimination rule, in the style of a *natural deduction* proof, is written as:

$$\frac{w: W_{a:A} B(a), \quad a: A, \quad f: B(a) \rightarrow W_{x:A} B(x), \quad c: \prod_{b: B(a)} C(f(b)) \vdash h(a, f, c): C(\sup(a, f))}{\text{elim}(w, h): C(w)}$$

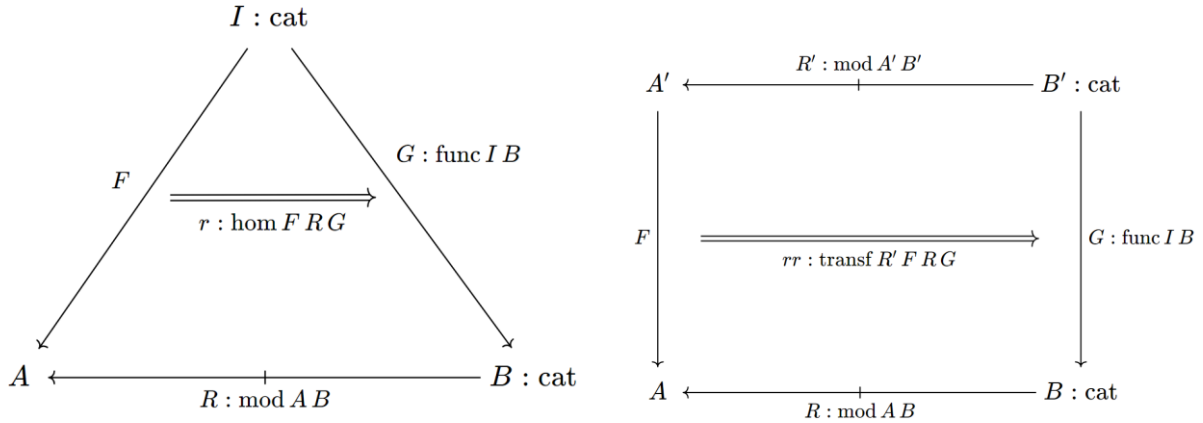
Higher inductive types not only define a new type with constants and functions that create elements of the type, but also new instances of the *identity type* ($_ = _$) that relate those elements. A simple example is the *circle type*, which is defined with two constructors (a basepoint and a loop):

```
base : circle
loop : base =circle base
```

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 $F: I \rightarrow C$ instead of via their usual objects " $F:\text{Ob}(C)$ ". 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 : Π [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);
```



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);

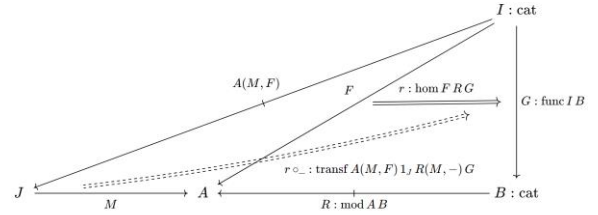
```

But the usual inner composition/cut inside categories

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

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

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



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;

```

7 Outer cut-elimination or functorial lambda calculus.

This implementation of the outer cut-elimination essentially is a *new functorial lambda calculus via the « dinaturality » of evaluation* and the monoidal bi-closed structure of profunctors. The conjunction bifunctor $_ \otimes _$ has right adjoint implication bifunctor $_ \Rightarrow _$ via the lambda/eval bijection of hom-sets. Then dinaturality is used to accumulate the argument-component of the eval operation instead on its function-component:

$$\begin{aligned}
 & \text{“eval}_{B,0} \circ B \otimes (g)” \circ (x \otimes f) \\
 &= \text{“eval}_{A,0} \circ A \otimes ((x \Rightarrow 0) \circ (g \circ f))”, \quad x : A \rightarrow B
 \end{aligned}$$

```

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

```

```

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

```

```

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

```

```

injective symbol Eval_cov_transf : Π [A B X A' : cat] [P : mod A B] [Q : mod B X] [O : mod A' X] [F : func A A'] ,

```

```

  transf P                                     F (O ⇐ Q) Id_func →

```

```

  transf (P ⊗ Q) F O                               Id_func;

```

```

constant symbol Tensor_cov_transf : Π [A' I I' X' A X : cat] [P' : mod A' I'] [Q' : mod I' X'] [P : mod A I] [Q : mod I X] [F : func A' A] [G : func X' X] , Π (M : func I' I),

```

```

  transf P' F (P <<◦ M) Id_func → transf (Q') Id_func (M ◦>> Q) G →

```

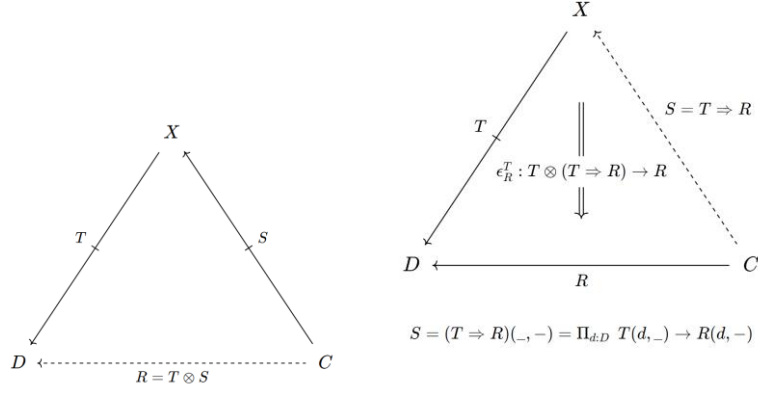
```

  transf ((⊗) P' Q') F ((⊗) P Q) G;

```

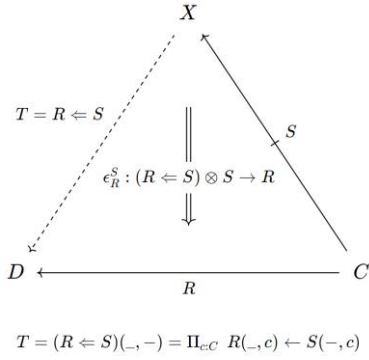
rule (Eval_cov_transf \$pq_o) ◦'' (Tensor_cov_transf \$M \$p'p \$q'q)

Eval_cov_transf ((ImPLY_cov_transf (Id_transf _) \$q'q) ◦'' (((\$pq_o <<=1 \$M) ◦'' \$p'p));



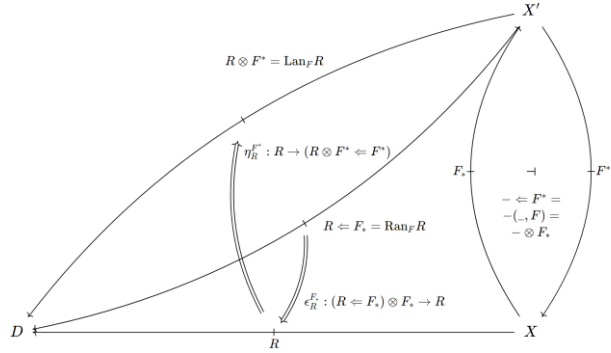
$$R = (T \otimes S)(-, -) = \Sigma_{x: X} T(-, x) \times S(x, -)$$

$T \otimes - \dashv T \Rightarrow -$ contravariant imply ("right lifting")

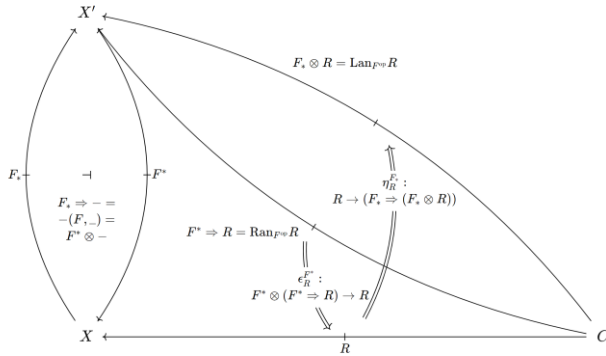


$$T = (R \Leftarrow S)(-, -) = \Pi_{c: C} R(-, c) \Leftarrow S(-, c)$$

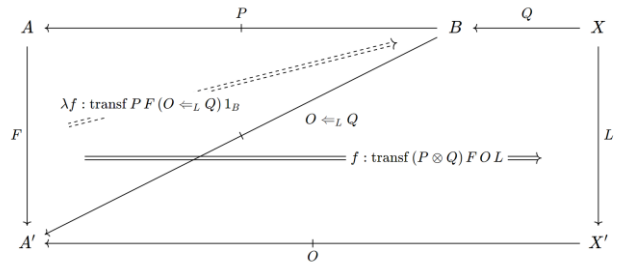
$- \otimes S \dashv - \Leftarrow S$ covariant imply ("right extension")



$$- \otimes F^* \dashv - \Leftarrow F^* = -(F_*, -) = - \otimes F_* \dashv - \Leftarrow F_*$$



$$F_* \otimes - \dashv F_* \Rightarrow - = -(F_*, -) = F^* \otimes - \dashv F^*$$



8 Inner cut-elimination or decidable adjunctions.

The *inner* cuts/compositions/actions within categories must also be eliminated/admissible/computational in a confluent/convergent manner in order to obtain the automatic-decidability of the categorial equations.

Here “cut” is synonymous with either of

- non-fibred composition of arrows (or action by arrows on a profunctor/module), or
- fibred composition of fibred arrows (or action by fibred arrows on a fibred profunctor), or
- fibred transport (action by non-fibred arrows on a fibred profunctor).

For reference, § 4.1.5 in Kosta Dosen’s book says that an adjunction with left adjoint functor $F: \text{catB} \rightarrow \text{catA}$, right adjoint functor $G: \text{catA} \rightarrow \text{catB}$, counit transformation $\phi_A: F G A \rightarrow A$ (where among many formulations, A could be seen as a *parameter* functor exposing a *variable* X with $\phi_X^A: F (G A X) \rightarrow A (X)$, that is $\phi_X^A: \text{hom}_{\text{catA}(F,A)} (G A X) (X)$), and unit transformation $\gamma_B: B \rightarrow G F B$, is formulated as rewrite rules from any redex outer cut on the left-side to the contractum containing some *smaller* inner cut, where f_1, g_1 are the (fixed) *parameters* and f_2, g_2 are the *natural variables*. Those rules are classified as:

- *Accumulation rules* (those accumulation equations can be formulated once generically for all such transformations):

$$f_2 \circ (f_1) "A \circ \phi \circ F" = (f_2 \circ f_1) "A \circ \phi \circ F"$$

$$"A \circ \phi \circ F" (g_1) " \circ F" \circ g_2 = "A \circ \phi \circ F" (g_1 \circ g_2)$$

- *Naturality rules*:

$$(f_1) "A \circ \phi \circ F" \circ ((f_2) "GA \circ 1") = ((f_1) "1 \circ 1" \circ f_2) "A \circ \phi \circ F"$$

$$f_2 \circ "A \circ \phi \circ F" (g_1) = "A \circ \phi \circ F" ((f_2) "GA \circ 1" \circ g_1)$$

- And similarly, for the naturality of the adjunction unit transformation, and for the *functoriality/naturality* (besides the generic accumulation rules) of every functor:

$$("1 \circ FB" (g_2)) \circ "G \circ \gamma \circ B" (g_1) = "G \circ \gamma \circ B" (g_2 \circ "1 \circ 1" (g_1))$$

$$("1 \circ B" (g_2)) \circ "1 \circ B" (g_1) = "1 \circ B" (g_2 \circ "1 \circ 1" (g_1))$$

- *Beta-cancellation conversion (or rewrite) rules* (i.e., $(\lambda -. C[-]) \cdot _ = C[-]$; the other half, *Eta-cancellation* $\lambda -. (g \cdot -) = g$ is similar):

$$(f_1) "A \circ \phi \circ F" ((f_2) "G \circ \gamma \circ B" (g_1)) = (f_1) "A \circ 1" ((f_2) "1 \circ FB" (g_1))$$

With (substructural) variations such as:

$$"A \circ \phi \circ F" ((f_2) "G \circ \gamma \circ B") = f_2$$

$$"1 \circ \phi \circ F" ("G \circ \gamma \circ B" g_2) = "1 \circ FB" g_2$$

where indeed the functions on arrows $F -$ and $G -$ of those functors are not primitive but are themselves the (Yoneda) “*antecedental/consequential transformation*” formulations “ $1 \circ F -$ ” or “ $G - \circ 1$ ” of the identity arrows on applied-functor objects...

```

symbol Func_con_hom :  $\Pi$  [A B A' : cat] (Z : func A A') (F : func B A),
  hom F (Unit_mod Z Id_func) (Z <° F);

constant symbol Adj_con_hom :  $\Pi$  [L R : cat] [LAdj_func : func R L] [RAAdj_func : func L R] (aj : adj
LAdj_func RAAdj_func),  $\Pi$  [I] (Z : func I R) [J] (N : func J I),
  hom N (Unit_mod Z RAAdj_func) (N >° (Z >° LAdj_func));

assert [C D : cat] [F : func D C] [G : func C D] [aj : adj F G] [C'] [N : func C' C] [I] [X : func
I C'] [C''] [N' : func C'' C'] [I'] [X' : func I' I] [Z : func I' C''] (f : hom X' (Unit_mod X N'))
Z) [D'] [M : func D' D] [J] [Y : func J D'] [D''] [M' : func D'' D'] [J'] [Y' : func J' J] [W :
func J' D''] (g : hom W (Unit_mod M' Y) Y')  $\vdash$  eq_refl _ :  $\pi$  (

  ((g '° (M')_>° (Adj_con_hom aj M Y)) >°'_ (N <° X <° X'))

    '° ((M' >° M)_>° ((Adj_cov_hom aj N X) >°'_ (N') >° f))

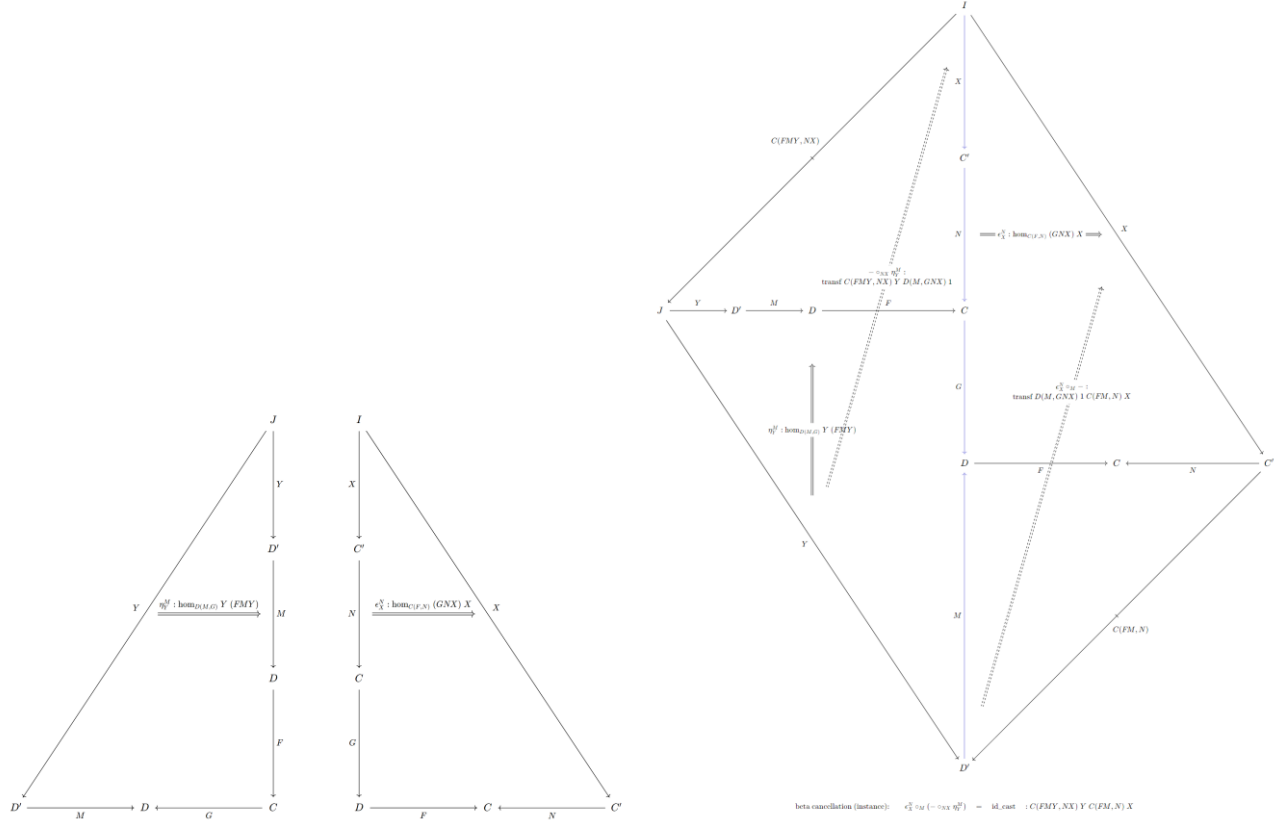
  = ((g '° (M')_>° (Func_con_hom (M >° F) Y)) >°'_ (N <° X <° X'))

```

```

    '◦ ((M' ◦> M ◦> F)'◦> ((Func_cov_hom N X) ◦>'(N') ◦' f)) );
// : transf (Unit_mod (Y' ◦> (Y ◦> (M ◦> F))) (N <◦ X <◦ X'))
    W (Unit_mod (M' ◦> M ◦> F) (N <◦ N')) Z

```



9 Synthetic fibred Yoneda.

Lemma 4.41.1 (2-Yoneda lemma for fibred categories). Let \mathcal{C} be a category. Let $S \rightarrow \mathcal{C}$ be a fibred category over \mathcal{C} . Let $U \in \text{Ob}(\mathcal{C})$. The functor

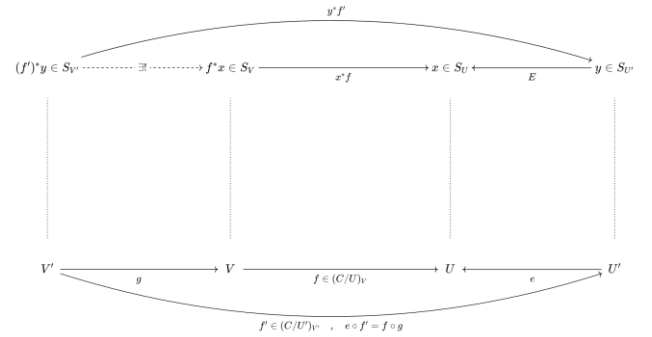
$$\text{Mor}_{\text{Fib}/\mathcal{C}}(\mathcal{C}/U, S) \rightarrow S_U$$

given by $G \mapsto G(\text{id}_U)$ is an equivalence, whose inverse:

$$S_U \rightarrow \text{Mor}_{\text{Fib}/\mathcal{C}}(\mathcal{C}/U, S)$$

is such that given $x \in \text{Ob}(S_U)$ the associated functor is:

- on objects: $(f: V \rightarrow U) \mapsto f^*x$, and
- on morphisms: the arrow $(g: V'/U \rightarrow V/U)$ maps to the universality morphism $(f \circ g)^*x \rightarrow f^*x$ fibred over the arrow g .



10 Substructural fibred Yoneda.

Expressed as a blend of the *generalized (above-any-arrow) universality/introduction rule* of the transported/pulledback objects, via the (fibred) Yoneda formulation, inside fibred categories:

```

symbol Fibration_con_funcd :  $\Pi$  [I X X' : cat] [x'x : func X' X] [G : func X I] [JJ : catd X'] [F : func X' I] [II : catd I] (II_isf : isFibration_con II),

```

```

 $\Pi$  (FF : funcd JJ F II) (f : hom x'x (Unit_mod G Id_func) F),
funcd JJ x'x (Fibre_catd II G);

```

```

constant symbol Fibration_con_intro_homd :  $\Pi$  [I X X' : cat] [x'x : func X' X] [G : func X I] [JJ : catd X'] [F : func X' I] [II : catd I] (II_isf : isFibration_con II) (FF : funcd JJ F II) (f : hom x'x (Unit_mod G Id_func) F) [X'0 : cat] [x'0x : func X'0 X] [X'' : cat] [x''x' : func X'' X'] [x''x'0 : func X'' X'0] (x'0x' : hom x''x'0 (Unit_mod x'0x x'x) x''x') [KK : catd X'0] (GG : funcd KK x'0x (Fibre_catd II G)) [HH : funcd (Fibre_catd JJ x''x') x''x'0 KK],

```

```

homd ((x'0x' '0 ((x'0x)'0> f))) HH (Unit_modd (GG 0>d (Fibre_elim_funcd II G)) Id_funcd)
((Fibre_elim_funcd JJ (x''x')) 0>d FF) 0>
homd x'0x' HH (Unit_modd GG (Fibration_con_funcd II_isf FF f)) (Fibre_elim_funcd JJ (x''x'));

```

blended together with the (*covariant*) *composition operation*, via the (indexed) Yoneda formulation, inside fibred categories:

```

constant symbol 0>d'_ :  $\Pi$  [X Y I : cat] [F : func I X] [R : mod X Y] [G : func I Y] [r : hom F R G] [A : catd X] [B : catd Y] [II] [FF : funcd II F A] [RR : modd A R B] [GG : funcd II G B],

```

```

homd r FF RR GG 0>
 $\Pi$  [J : cat] [M : func J Y] [JJ : catd J] (MM : funcd JJ M B),
transfd ( r 0>'_ (M) ) (Unit_modd GG MM) FF (RR d<<0 MM) Id_funcd;

```

Thereby this blend allows to express the outer (first) functorial-action by S_U or the inner functorial-action by C/U (or both simultaneously) in the mapping:

$$S_U \rightarrow Mor_{Fib/C}(C/U, S)$$

11 Comma elimination (“J-rule arrow induction”).

The above intrinsic/structural universality formulation comes with a corresponding *reflected/internalized algebra formulation*, which is the comma category where the J-rule elimination (“equality/path/arrow induction”) occurs.

```

constant symbol Comma_con_intro_funcd :  $\Pi$  [A B I : cat] [R : mod A B] (BB : catd B) [x : func I A] [y : func I B] (r : hom x R y),

```

```

funcd (Fibre_catd BB y) x (Comma_con_catd R BB);

```

```

constant symbol Comma_con_elim_funcd :  $\Pi$  [I X : cat] [G : func X I] [II : catd I] (II_isf : isFibration_con II) [JJ : catd I] (FF : funcd JJ Id_func II),

```

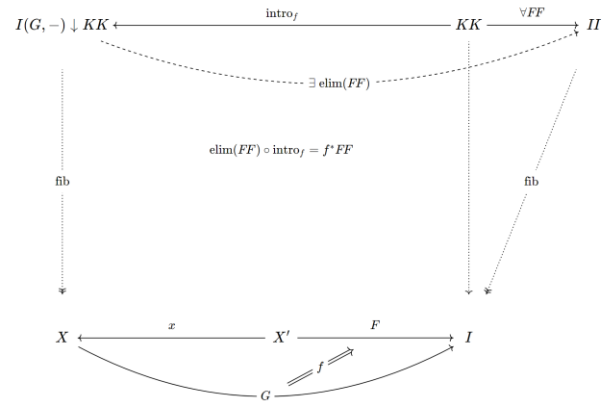
```

funcd (Comma_con_catd (Unit_mod G Id_func) JJ) G II;

```

Similarly, pullbacks have a universal formulation (fibre of fibration), an algebraic formulation (composition of spans), or

mixed (product of fibration-objects in the slice category).



12 Cut-elimination for fibred arrows.

Non-fibred composition cut-elimination only considers pairs of arrows:

$$p: X \rightarrow Y \text{ then } q: Y \rightarrow Z$$

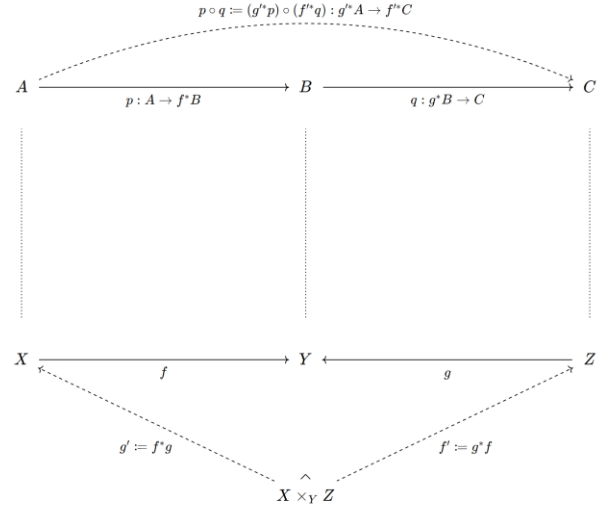
Fibred composition cut-elimination should also consider pairs of arrows:

$$p: A \rightarrow f^*B \text{ then } q: g^*B \rightarrow C$$

Therefore, *grammatically* any fibred arrow should be fibred over a span-of-arrows, instead of over one object (the identity arrow), or more generally should be fibred over a *polynomial-functor* with intrinsic *distributivity* ($\Pi\Sigma = \Sigma\Pi\varepsilon^*$):

$$r: g^*A \rightarrow f^*Z$$

All these *intrinsic* structures are reflected/internalized as an *explicit substitution/pullback-type-former* for any fibration.



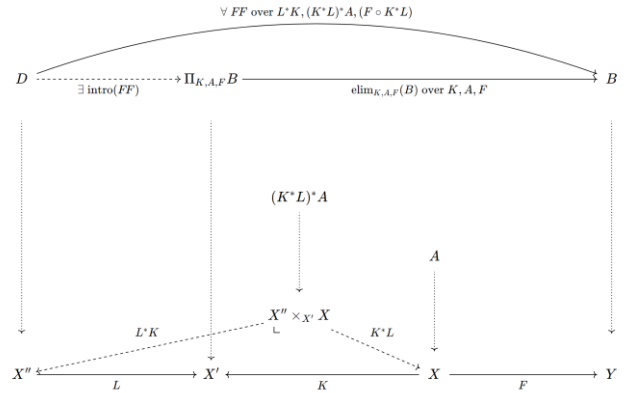
13 Pi-category-of-fibred-functors and Sigma-category.

Ordinary Pi/product-types consider only the sections of some fibration, but Pi-categories should consider all the category-of-fibred-functors to some fibration, this leads to the new construction of *Pi-category-of-functors*.

This preliminary implementation does not yet use the more-intrinsic formulation using *functors fibred-above-a-span of functors*. But *Sigma/sum-categories* can be already intrinsically-implemented using only functors fibred-forward-above a single functor.

```
constant symbol Pi_intro_funcd : Π [X' X Y :
cat] (K : func X X') (A : catd X) (F : func X
Y) (B : catd Y) [X''] (D : catd X'') (L : func
X'' X') (FF : funcd (Productd_catd (Fibre_catd
A (Pullback_snd_func L K)) (Fibre_catd D
(Pullback_fst_func L K))) ((Pullback_snd_func L
K) ∘> F) B),
funcd D L (Pi_catd K A F B);

constant symbol Pi_elim_funcd : Π [X' X Y :
cat] (K : func X X') (A : catd X) (F : func X
Y) (B : catd Y),
funcd (Productd_catd A (Fibre_catd (Pi_catd K A
F B) K)) F B;
```



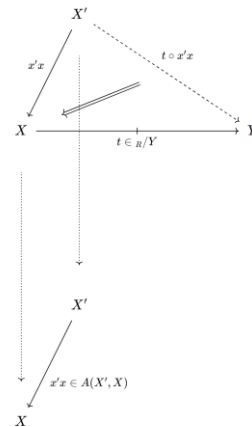
14 And why profunctors (of sets)?

The primary motivation is that they form a monoidal bi-closed double category (*functorial lambda calculus*).

Another motivation is that the subclass of fibrations called *discrete/groupoidal fibrations* can only be computationally-recognized/expressed instead via (indexed) presheaves/profunctors of sets. And the comma construction is how to recover the intended discrete fibration.

Ultimately profunctors enriched in preorders/quantaes instead of mere sets could be investigated.

Visualizing the comma/slice category as a fibred category of *triangles of arrows fibred by their base*:



15 What is a fibred profunctor anyway?

The comma/slice categories are only fibred categories (of triangles of arrows fibred by their base), not really fibred profunctors. One example of fibred profunctor from the coslice category to the slice category is *the set of squares fibred by their diagonal* which witnesses that this square is constructed by pasting two triangles.

```
constant symbol Comma_homd : Π [A B I : cat] (R
: mod A B) [x : func I A] [y : func I B], Π
[J0] [F : func J0 A] [x' : func I J0] (x'x :
hom x' (Unit_mod F x) Id_func), Π [J1] [G :
func J1 B] [y' : func I J1] (yy' : hom Id_func
(Unit_mod y G) y'), Π (s : hom x' (F >> R) y)
(t : hom x (R << G) y') (r : hom x' (F >> (R
<< G)) y'),
```

```
π (( x'x '◦ ((F)'>> t) ) = r) → π (( (s ◦
>'(G)) ◦' yy' ) = r) →
homd r (Comma_con_intro_funcd (Triv_catd B) s)
((Comma_con_comp_funcd R (Triv_catd B) F) ◦
>>d ((Comma_modd (Triv_catd A) R (Triv_catd B))
d<<◦ (Comma_cov_comp_funcd (Triv_catd A) R G))
(Comma_cov_intro_funcd (Triv_catd A) t);
```

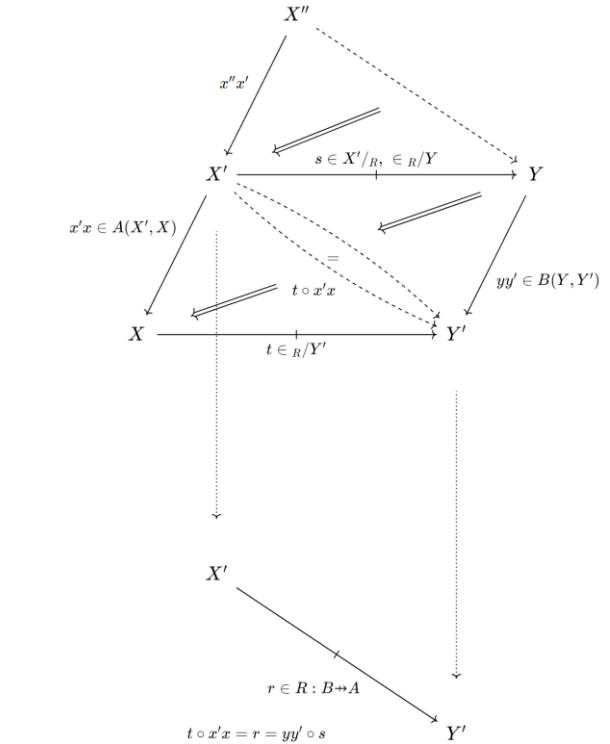
This article implements such fibred profunctor of (cubical) squares (thereby validating the hypothesis that computational-cubes should have connections/diagonals...).

For witnessing the (no-computational-content) pasting along the diagonal, this implementation uses for the first time the *LambdaPi-metaframework's equality predicate* which internally-reflects all the conversion-rules; in particular the implementation uses here *the categorial-associativity equation axiom*, which is a provable metatheorem which must *not* be added as a rewrite rule!

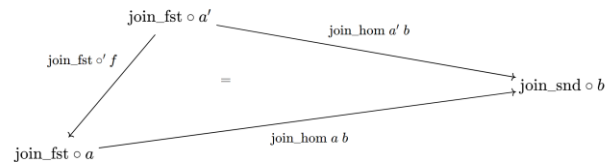
16 Higher inductive types, the interval type, concrete categories.

This article implements (higher) inductive datatypes such as the *join-category* (interval simplex) and *concrete categories* (ref. the section "Applications"), with their introduction/elimination/computation rules. The (3-dimensional) *naturality cone conditions*, which relate the (2-dimensional) arrows introduced by the introduction-rules, are expressed using the LambdaPi metaframework conversion rewrite rules.

```
symbol join_cat : Π (A B : cat), cat;
symbol join_fst_func : Π (A B : cat), func A
(join_cat A B);
symbol join_snd_func : Π (A B : cat), func B
(join_cat A B);
symbol join_hom : Π (A B : cat) [I : cat] (a :
func I A) (b : func I B),
hom a (Unit_mod (join_fst_func A B)
(join_snd_func A B)) b;
```



```
rule @'◦ _ _ _ _ _ $a' Id_func _ _ $r ((
Id_func ) _'◦> (join_hom $A $B $a $b)) ◦
(join_hom $A $B $a' $b);
```



But for the elimination rules, those (3-dimensional) naturality cone conditions are expressed using the LambdaPi equality predicate = (which internally-reflects its rewrites rules), to relate the (2-dimensional) arrows arguments. Note that the naturality cone conditions *carry no computational content* and are only logical consistency checks.

```

symbol join_elim_con_func :  $\Pi$  (A B : cat) [E : cat] (first_func : func A E) (second_func : func B E)
(one_hom :  $\Pi$  (I : cat) (a : func I A) (b : func I B), hom a (Unit_mod (first_func) Id_func)
(second_func <° b))

(natural_eq :  $\Pi$  [I : cat] (a : func I A) (b : func I B) [a'] (r : hom a' (Unit_mod Id_func a)
Id_func),
   $\pi$  (r '° (( _ ) _'°> (one_hom I a b)) = (one_hom I a' b))),

func (join_cat A B) E;

rule join_fst_func $A $B °> (join_elim_con_func $A $B $F0 $F1 $r _)  $\hookrightarrow$  $F0
with join_snd_func $A $B °> (join_elim_con_func $A $B $F0 $F1 $r _)  $\hookrightarrow$  $F1;

rule ((join_hom $A $B $a $b) '° ((join_fst_func $A $B) _'°>
  (Func_con_hom (join_elim_con_func $A $B $first_func $second_func $one_hom _
    (join_snd_func $A $B))))  $\hookrightarrow$  $one_hom _ $a $b ;

```

17 Universe, universal fibration.

This article implements a *grammatical (univalent) universe* and the universal fibration classifying small fibrations, together with the dual universal ofibration.

This universe is made grammatical (univalent) by declaring an inverse to the fibrational-transport inside the universe fibration.

```

constant symbol Universe_con_cat : cat;
constant symbol Universe_con_catd : catd
Universe_con_cat;

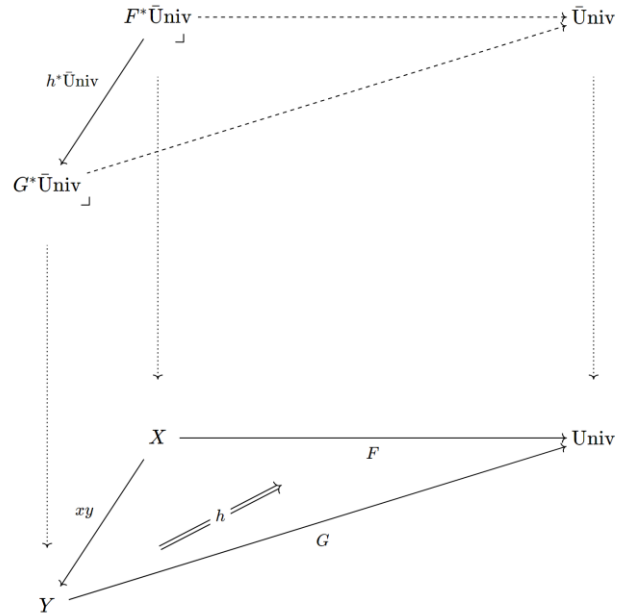
symbol Universe_con_func :  $\Pi$  [X : cat] (A :
catd X) (A_isf : isFibration_con A), func X
Universe_con_cat;

symbol Universe_con_funcd :  $\Pi$  [X : cat] (A :
catd X) (A_isf : isFibration_con A), funcd A
(Universe_con_func A A_isf) Universe_con_catd;

injective symbol
Universe_Fibration_con_funcd_inv :  $\Pi$  [X Y : cat]
(F : func X Universe_con_cat) (G : func Y
Universe_con_cat) [xy : func X Y],

funcd (Fibre_catd Universe_con_catd F) xy
(Fibre_catd Universe_con_catd G)  $\rightarrow$ 
hom xy (Unit_mod G Id_func) F;

```

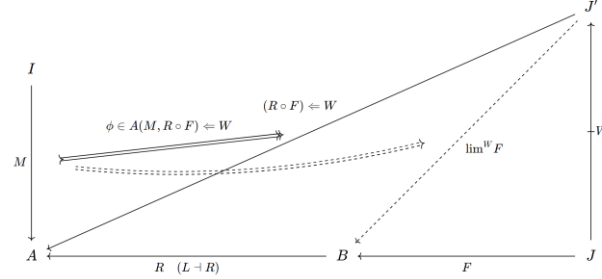


18 Weighted limits.

This article implements *profunctor-weighted limits* (that the *right-extension* $\text{Hom}(-, F) \Leftarrow W$ is representable as $\text{Hom}(-, \lim^W F)$) and *profunctor-weighted colimits* (that the *right-lifting* $W \Rightarrow \text{Hom}(F, -)$ is representable as $\text{Hom}(\text{colim}^W F, -)$).

And it can *computationally-prove* that left-adjoint functors preserve profunctor-weighted colimits from its computational-proof that *right-adjoint functors preserve profunctor-weighted limits*.

A computational-proof is to be contrasted from a logical deduction which uses the reflected/internalized LambdaPi propositional equality.



```
constant symbol limit_cov : Π [B J0 J J' : cat] (K : func J J0) (F : func J0 B) (W : mod J' J) (F_<_W :
: func J' B), TYPE;

injective symbol limit_cov_univ_transf : Π [B J J' : cat] [W : mod J' J] [F : func J B] [F_<_W :
func J' B]

(isl : limit_cov F W F_<_W), Π [I : cat] (M : func I B),
transf (((Unit_mod M F)) <-> W) Id_func (Unit_mod M F_<_W) Id_func;

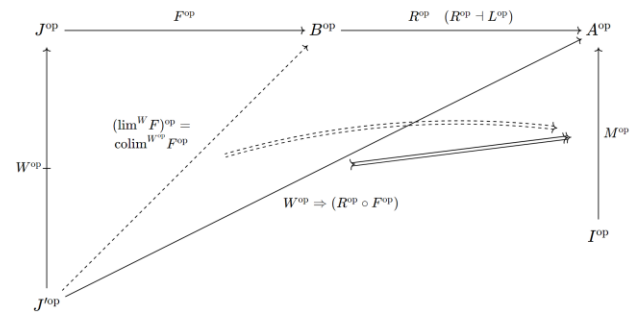
symbol right_adjoint_preserves_limit_cov [B J J' A : cat] [W : mod J' J] [F : func J B] [F_<_W : func
J' B] (isl : limit_cov F W F_<_W) [R : func B A] [L : func A B] (isa : adj L R) [I : cat] (M : func I
A) :

transf ((Unit_mod M (F > R)) <-> W) Id_func (Unit_mod M R << F_<_W) Id_func

:= ((Adj_con_hom isa M Id_func) >' (F_<_W)) >'
((limit_cov_univ_transf isl (M > L)) >'
(Implied_cov_transf ((M)' > Adj_cov_hom isa F Id_func) (Id_transf W)));
```

19 Duality Op, covariance vs contravariance.

This article implements the *dualizing Op* operations for categories, functors, profunctors/modules, hom-arrows, transformations, adjunctions, limits, fibrations... which are used to computationally-prove that left-adjoint functors preserve profunctor-weighted colimits from the proof that right-adjoint functors preserve profunctor-weighted limits.



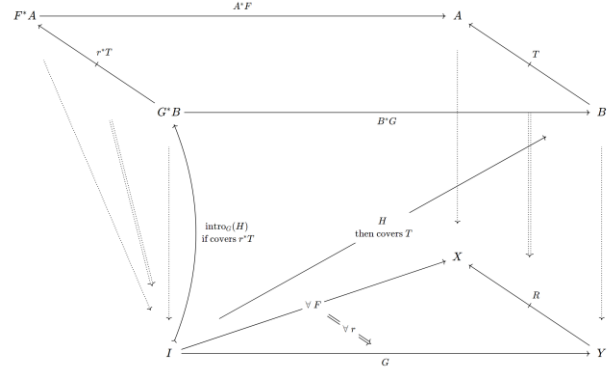
```
symbol left_adjoint_preserves_limit_con [B J J' A : cat] [W : mod J J'] [F : func J B] [W_<_F : func
J' B] (isl : limit_con F W W_<_F) [R : func A B] [L : func B A] (isa : adj L R) [I : cat] (M : func I
A) :

transf (W => (Unit_mod (F > L) M)) Id_func (W_<_F >> Unit_mod L M) Id_func
```

```
:= Op_transf (righ_adjoint_preserves_limit_cov (Op_limit_cov isl) (Op_adj isa) (Op_func M));
```

20 Grammatical topology.

Finally, there is an experimental implementation of *covering (co)sieves towards grammatical sheaf cohomology* and towards a description of algebraic geometry's schemes in their formulation as *locally affine ringed sites* (structured topos)... The implementation of the covering (co)sieve *predicate* uses ideas from the local modifier $j: \Omega \rightarrow \Omega$ where $\Omega(A)$ is the classifier of (sub-)objects (sieves) \mathcal{U} of the object A , and where $j_A(\mathcal{U})(f: X \rightarrow A) := "f^*\mathcal{U} \in J(X)"$ is the (opaque) set of witnesses that the pullback-sieve $f^*\mathcal{U}$ is covering. Then the transitivity axiom: $\mathcal{U} \in J(A)$ if $(\forall f: \text{some cover of } A, j_A(\mathcal{U})(f))$, iff $j_A(j_A(\mathcal{U}))$, becomes expressible.



```
constant symbol covering :  $\Pi$  [X Y I : cat] [A : catd X] [R : mod X Y] [B : catd Y] (RR : modd A R B)
[y : func I Y], funcd (Triv_catd I) y B  $\rightarrow$  TYPE ;

constant symbol coveringu :  $\Pi$  [I : cat] [A : catd I] [B : catd I] (RR : moddu A B), funcd (Triv_catd I)
Id_func B  $\rightarrow$  TYPE ;

constant symbol Total_covering :  $\Pi$  [X Y I : cat] [A : catd X] [R : mod X Y] [B : catd Y] [RR : modd A
R B] [G : func I Y] [H : funcd (Triv_catd I) G B],
( $\Pi$  [F : func I X] (r : hom F R G), coveringu (Fibre_hom_moddu RR r) (Fibre_intro_funcd _ G Id_func
H))  $\rightarrow$ 
covering RR H ;

constant symbol Glue_transfd :  $\Pi$  [X Y X' Y' : cat] [A' : catd X'] [A : catd X] [B' : catd Y'] [B : catd
Y] [xx' : func X X'] [yy' : func Y Y'] [R' : mod X' Y'] [R : mod X Y] [rr' : transf R xx' R' yy'] [RR
: modd A R B] [FF : funcd A xx' A'] [RR' : modd A' R' B'] [GG : funcd B yy' B'],  $\Pi$  [I : cat] [G :
func I Y] [H : funcd (Triv_catd I) G B], covering RR H  $\rightarrow$ 
( $\Pi$  [F : func I X] (r : hom F R G), transfd (Fibre_hom_moddu RR r)
(Fibre_intro_funcd _ (F  $\circ$  xx') Id_func (Fibre_elim_funcd A F  $\circ$  FF))
(Fibre_hom_moddu (Sheaf_con_modd RR') (r  $\circ$  rr'))
(Fibre_intro_funcd _ (G  $\circ$  yy') Id_func (Fibre_elim_funcd B G  $\circ$  GG)))  $\rightarrow$ 
transfd rr' RR FF (Sheaf_con_modd RR') GG;
```

A sheaf is data defined over some topology, and sheaf cohomology is linear algebra with data defined over some topology. A closer inspection reveals that there is some intermediate formulation which is computationally-better than Čech 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 (semantically-)possibly-incompatible functions to be (grammatically/formally)-glued regardless, 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, this sheaf data type, gives the gluing operation:

$$\begin{aligned} F(U0) &:= \text{sum over the slice } U0 \rightarrow U01 = \mathbb{Z} \oplus \mathbb{Z}; \\ F(U1) &:= \mathbb{Z} \oplus \mathbb{Z}; F(U01) := \mathbb{Z} \\ F(U) &= \text{kan extension} = \mathbb{Z} \oplus \mathbb{Z} \oplus \mathbb{Z}; \end{aligned}$$

$$\begin{aligned} \text{gluing: } F(U0) \oplus F(U1) \oplus F(U01) &\rightarrow F(U) \\ ((f0, f01), (g1, g01), (h01)) &\mapsto (f0, g1, f01 + g01 - h01) \end{aligned}$$

where the signed sum generalizes to higher degrees because the Euler characteristic is 1 (or inclusion–exclusion principle).

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 :  $\Pi$  [I], func I nat_cat  $\rightarrow$  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))  $\circ$ > 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) :  $\Pi$  [C0] [X0 : func C0 C] [I] [X : func I C0] [Y : func I
_],
hom X (Unit_mod X0 (inat_func C)) Y  $\rightarrow$  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 (iproduct_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 $[12;11] \times [22;21] \times [33;32;31]$ now is given an extra constant arrow $[22;21] \rightarrow [33;32;31]$ 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)
```

Besides, cut-elimination in the double category of fibred profunctors have immediate executable/computational applications to graphs transformations understood as categorial rewriting, where the objects are graphs (or sheaves in a topos), the vertical monomorphisms are pattern-matching subterms inside contexts and the horizontal morphisms are congruent/contextual rewriting steps...

22 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.

23 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

• 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.](#)

• Q2. This text specifies that ideas from the lambda calculus’ beta-eta evaluation for the product $- \times -$ and the exponential $- \rightarrow -$ appear here:

- (A) Only outer in the form of the tensor \otimes of profunctors and the right (or left) implication/lifting $(-\Rightarrow -)$.
- (B) Only inner in the form of the of the product object $- \times -$ and the exponential object $- \rightarrow -$ adjunction within a fixed category.
- (C) Both in those outer forms and inner forms, and moreover in the (bonus) outer form of the product category $- \times -$ and the functor category $- \rightarrow -$.

Q2 ; 30 / quiz [Click or tap here to enter text.](#)

• Q3. This text specifies that the functorial programmer can manually prove some adjunction equation; now for automation:

- (A) Any equation in the language of any one adjunction can already be decided whether it holds or not.
- (B) Only the equations for the product/exponential (lambda calculus) can be decided.
- (C) Decidability is only if all the variables occurring in the equation have been instantiated with concrete data values.

Q3 ; 30 / quiz [Click or tap here to enter text.](#)

• Q4. This text specifies that the goal of functorial programming is:

- (A) To compute the addition $1+2=3$ faster than functional programming.
- (B) To compute the coproduct $1+2=3$ faster than functional programming.
- (C) To compute algorithms such as the addition or coproduct $1+2=3$ and express their logical specification and correctness.

Q4 ; 30 / quiz [Click or tap here to enter text.](#)

• Q5. This text specifies that which functorial logic operation is more primitive?

- (A) The fibred transport (Yoneda action by non-fibred arrows on a fibred profunctor)
- (B) The comma/arrow elimination rule which produces a functor (action) out of the comma/arrow category.
- (C) None of them are computational in this text, they are only manual logical specifications.

Q5 ; 30 / quiz [Click or tap here to enter text.](#)

• Q6. This text specifies that the computational dualizing Op is:

- (A) One single operation on the universe type of types, so to get everything automatically.
- (B) Many dualizing Op operations for categories, functors, adjunctions, limits, etc.
- (C) The hom $(- \rightarrow \perp)$ into a “dualizing unit” object \perp as in a star-autonomous category.

Q6 ; 30 / quiz [Click or tap here to enter text.](#)

24 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