Dosen functorial programming: A computational logic (coinductive) interface for algebraic geometry schemes implemented in Lambdapi

Christopher Mary; Camille Nous

Abstract. This is the continuation of an ongoing research programme of discovering a truly computational logic (Lambdapi type theory) for categories, profunctors, fibred categories, univalence, polynomial functors, sites, sheaves and schemes, in the style of Kosta Dosen's book « Cut-elimination in categories » (1999). Firstly, a glue operation for any sheaf over the sheafification and sieve-closure modalities is declared. Now a transformation (predicate) into the sieves-classifier (truth-values) corresponds to a sub-profunctor (fibred/dependent profunctor). It is not necessary to use multi-categories and hyperdoctrines (categories with families …) to manage (tensor/dependent types) contexts; the answer lies in a (computational) implementation of the (usually semantic) context-extension operation. Then, there is an implementation of locally ringed sites and schemes, whose slice-categories satisfy the (computational) interface/specification of an affine scheme. But the affine-scheme interface is coinductive (self-reference), meaning that its slice-categories are also required to satisfy the affine-scheme interface. Moreover, the invertibility-support D(f) for a locally ringed site is defined as the sieve of opens where the function f is invertible; and this sieve becomes generated by a singleton when restricted to a slice affine-scheme. This new proof assistant for schemes will probably succeed to express more applied mathematics such as blow-up of schemes (projective Proj bundle) and Cartier divisors. Indeed, this formulation is already able to compute that 1+2=3 via 3 different methods: the category of natural-numbers as a higher inductive type; the natural-numbers object inside any fixed category; and the colimits inside the category of finite sets/numbers. Indeed, this new functorial programming language, also referred as Dosen's « m— » or « emdash » or « modos », is able to express the usual logic such as the tensor and internal-hom of profunctors, the sigma-sum and pi-product of fibred categories/profunctors; but is also able to express the concrete and inductively-constructed categories/profunctors, to express the adjunctions such as the product-and-exponential or the constant-diagram-and-weighted-limit adjunctions within any fixed category, to express contravariance and duality such as a computational-proof that right-adjoints preserve limits from the dual statement, to express groupoids and univalent universes, to express polynomial functors as bicomodules in the double category of categories, functors, cofunctors and profunctors, etc. Article's source: https://github.com/1337777/cartier/blob/master/cartierSolution16.lp

Keywords: Kosta Dosen, proof assistants, AI assistants, algebraic geometry, polynomial functors, homotopy types

TABLE OF CONTENTS

1 Introduction: Goal.

The *goal* is to implement a programming language and proof-assistant where the types are *categories*, and the implications are *functors*, but with a special construction to recover the usual types which are just (diagrams of) *sets*. And those former usual types, which are the ambient data environment for the new types as 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). In short: the goal is the discovery of a truly computational logic (Lambdapi type theory) for categories, profunctors, fibred categories, univalence, polynomial functors, sites, sheaves and schemes, in the style of Kosta Dosen's book « Cut-elimination in categories » (1999).

Category theory [0] (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 [0] 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 new functorial programming language, also referred as Dosen's « m— » or « emdash » or « modos », is able to express the usual logic such as the tensor and internal-hom of profunctors, the sigma-sum and pi-product of fibred categories/profunctors; but is also able to express the concrete and inductively-constructed categories/profunctors, to express the adjunctions such as the product-and-exponential or the constant-diagram-and-weighted-limit adjunctions within any fixed category, to express contravariance and duality such as a computational-proof that right-adjoints preserve limits from the dual statement, to express groupoids and univalent universes, to express polynomial functors as bicomodules in the double category of categories, functors, cofunctors and profunctors, etc…

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.

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*, 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 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 functor-of-points expressed internally within the Zarisky topos...

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*

$$\text{projectFirst} \circ \text{pair}\langle g, g' \rangle = g; \qquad \text{projectSecond} \circ \text{pair}\langle g, g' \rangle = g'$$

$$\text{case}[f|f'] \circ \text{injectLeft} = f; \quad \text{case}[f|f'] \circ \text{injectRight} = f'$$

Attempt for left-associativity normalization CHOICE A:

$$\big(\big((d \circ e) \circ \text{projectFirst}\big) \circ \text{pair}\langle g, g' \rangle\big) \circ h$$ ✗ 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:

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

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 $C[F, -] : D^{\text{op}} \times C \to \text{Set}$ 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: Review of an implementation of context-extension (categories with families).

The semantics of dependent types is usually studied as a "category with families". For example, Ambrus Kaposi [0] says that a category with N-many families is defined as a category (objects denoted Con, morphisms Sub) with a terminal object $\diamond$ (ϵ denotes the unique morphism into it); for each i a presheaf of types (action on objects denoted Ty – i, action on morphisms –[–]); for each i a locally representable dependent presheaf Tm over Ty – i (actions denoted Tm , –[–] , local representability is denoted – ▹ – : (Γ :Con) → Ty Γ i → Con with an isomorphism (p∘– , q[–]) : Sub Δ (Γ ▹ A) (γ : Sub Δ Γ) × Tm Δ ($A[\gamma]$) : (–,–) natural in Δ).

For implementing this proof assistant for schemes, all the constructions need to have also (substructural) versions of the story in the presence of context: tensor-product context A⊗B ⊢ C, subtype (sum type) context Σ(a:A),P(a) ⊢ C, and dependent-type context (x:A)|B(x) ⊢ C(x). But it is not necessary to use(semantical) multi-categories and hyperdoctrines (categories with families …) to manage contexts.

For dependent-type contexts, the answer lies in a computational (i.e., strictified equalities/conversions) implementation of the (usually semantic) context-extension operation, suitably generalized for a type theory for categories, functors and profunctors.

For subtype contexts, the answer already lies in an implementation of sieves/subobject classifiers/universes, and in the implementation of the Sigma-sum and the Pi-product of fibred categories/profunctors.

For tensor-product contexts, the answer lies in reformulating the elimination rule for the tensor product in multi-categories which says: if a:A, b:B ⊢ C then A⊗B ⊢ C. The Lambdapi logial framework allows for a so-called "higher-order abstract syntax" where the binding features of the Lambdapi metatheory are re-used to describe those of the object theory. But for this type theory for categories, functors and profunctors, some additional functoriality/naturality equality-conditions (on the body of the elimination rule) have to be discharged manually by the end-user (hopefully by a simple definitional reflexivity).

For comparison and for an introduction to Lambdapi, here is a computational implementation of the usual context-extension:

```
constant symbol  Con : TYPE;

constant symbol  Ty : Con → nat → TYPE;

constant symbol  ◇ : Con;

injective symbol  ▹ : Π (Γ : Con) [i], Ty Γ i → Con;

notation ▹ infix right 90;

constant symbol  Sub : Con → Con → TYPE;

symbol  ∘ : Π [Δ Γ Θ], Sub Δ Γ → Sub Θ Δ → Sub Θ Γ;

notation ∘ infix right 80;

rule /* assoc */  $γ ∘ ($δ ∘ $θ) ↪ ($γ ∘ $δ) ∘ $θ;

constant symbol  id : Π [Γ], Sub Γ Γ;

rule /* idr */  $γ ∘ id ↪ $γ

with /* idl */  id ∘ $γ ↪ $γ;

symbol  '⊤_ : Π [Γ Δ i], Ty Γ i → Sub Δ Γ → Ty Δ i;

notation '⊤_ infix left 70;

rule /* '⊤_-∘ */  $A '⊤_ $γ '⊤_ $δ ↪ $A '⊤_( $γ ∘ $δ )

with /* '⊤_-id */  $A '⊤_ id ↪ $A;

constant symbol  Tm : Π (Γ : Con) [i], Ty Γ i → TYPE;

symbol  'ᵗ_ : Π [Γ i A Δ], @Tm Γ i A → Π (γ : Sub Δ Γ), Tm Δ (A '⊤_ γ);
```

```
notation '^t_ infix left 70;

rule /*  '^t_-∘ */  $a '^t_ $γ '^t_ $δ ↪ $a '^t_( $γ ∘ $δ )

with /* '^t_-id */  $a '^t_ id ↪ $a;

injective symbol  ε : Π [Δ], Sub Δ ◇;

rule /* ε-∘ */  ε ∘ $γ ↪ ε

with /* ◇-η */  @ε ◇ ↪ id;

injective symbol  p_x : Π [Γ i] [A : Ty Γ i], Sub (Γ ▹ A) Γ;

injective symbol  q_x : Π [Γ i] [A : Ty Γ i], Tm (Γ ▹ A) (A '^T_ p_x);

injective symbol  &_x : Π [Γ Δ i] [A : Ty Γ i], Π (γ : Sub Δ Γ), Tm Δ (A '^T_ γ) ⟶ Sub Δ (Γ ▹ A);

notation &_x infix left 70;

rule /*  &_x-∘ */  ($γ &_x $a) ∘ $δ ↪ ($γ ∘ $δ &_x ($a '^t_ $δ));

rule /*  ▹-β₁ */  p_x ∘ ($γ &_x $a) ↪ $γ;

rule /* ▹-β₂ */  q_x '^t_ ($γ &_x $a) ↪ $a;

rule /* ▹-η */  (@&_x _ _ _ $A (@p_x _ _ $A) q_x) ↪ id;
```

5 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 [0]. 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 λΠ-calculus modulo theory. It is an interactive proof system that features dependent types like in Martin-Lőf'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 [0] about category theory are either about *categorial algebra* or *categorial logic*. For example, categorial algebra would understand a category similarly as an abstract algebra's *ring* and understand a "profunctor" as a *module of this ring*. Another example: categorial 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 $\to$) 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 $\to$) 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 $\bot$ which forces the evaluation arrow $A \vdash (A \to \bot) \to \bot$ into an isomorphism. For linearly distributive categories with negation, one adds a "monoidal unit" object $\bot$ 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 ealier attempt at categorial abstract machines (1986) and categorial datatypes (1987) by Curien-Hagino [0] contains a subtle bug... Another attempt by Cisinski [0] at categorial/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 [0] 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* or *module*) ϕ covariant over a category C and contravariant over a category D, written $\phi: C \nrightarrow D$ or $\phi: D \nleftarrow C$, 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 $\psi: E \nleftarrow D$ 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 $x \in \phi(d', c)$.

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*) $B : A \to U$, 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

$$\begin{gathered}\mathrm{List}(A) \coloneqq W_{(x:1+A)} B(x), \\ B(\mathrm{inl}(1_1)) \;=\; 0, \\ \text{forall } a,\;\; B(\mathrm{inr}(a)) \;=\; 1,\end{gathered}$$

where 1_1 is the sole inhabitant of $\mathbf{1}$. The value of $B(\mathrm{inl}(1_1))$ corresponds to the 0-ary constructor nil for the empty list, whereas the value of $B(\mathrm{inr}(a))$ 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

$$\frac{a{:}A \qquad f{:}B(a) \to W_{x:A}B(x)}{\sup(a,f){:}W_{x:A}B(x)}$$

The *elimination rule* for W-types works similarly to *structural induction* on trees. If, whenever a property (under the *propositions-as-types* interpretation) $C{:}W_{x:A}B(x) \to U$ 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:

$$\frac{\begin{gathered}w{:}W_{a:A}B(a),\;\; a{:}A,\;\; f{:}B(a) \to W_{x:A}B(x), \\ c{:}\textstyle\prod_{b:B(a)} C\big(f(b)\big) \vdash h(a,f,c){:}C\big(\sup(a,f)\big)\end{gathered}}{\mathrm{elim}(w,h){:}C(w)}$$

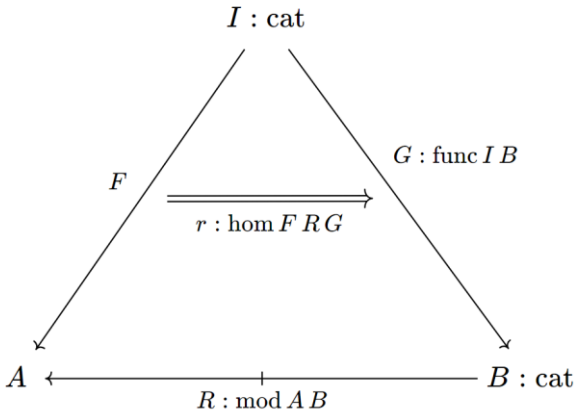
Higher inductive types not only define a new type with constants and functions that create elements of the type, but also new instances of the *identity type* ($\_ = \_$) that relate those elements. A simple example is the *circle type*, which is defined with two constructors (a basepoint and a loop):

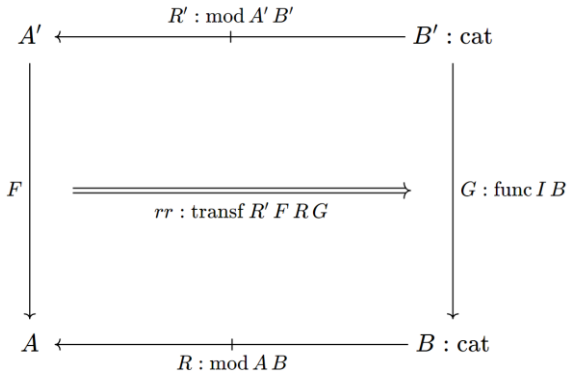
$$\begin{gathered}\mathrm{base} : \mathrm{circle} \\ \mathrm{loop} : \mathrm{base} \;=_{\mathrm{circle}}\; \mathrm{base}\end{gathered}$$

6 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 "$F{:}\mathrm{Ob}(C)$". 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);
```





7 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);
```

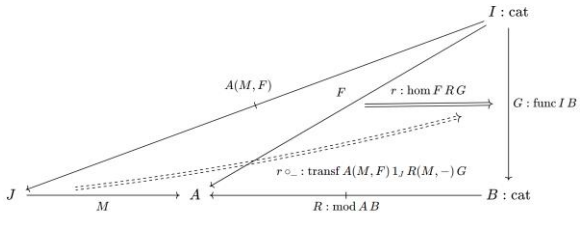
But the usual inner composition/cut inside categories

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

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

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

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




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

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

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

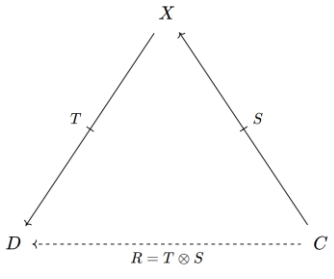
8 Outer cut-elimination or functorial lambda calculus.

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 _⊗_ has right adjoint implication bifunctor _⇒_ 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:

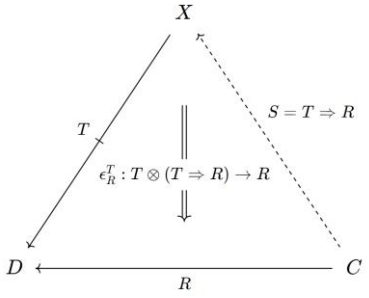
$$\text{“eval}_{B,O} \circ B \otimes (g)\text{”} \circ (x \otimes f)$$

$$= \text{“eval}_{A,O} \circ A \otimes \big((x \Rightarrow O) \circ (g \circ f)\big)\text{”}, \qquad x{:}A \to B$$

```
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' M Q G →  // usual asymmetry for composable pair

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));
```

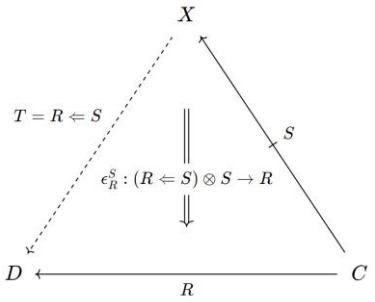


$R = (T \otimes S)(\_,-) = \Sigma_{x:X}\ T(\_,x) \times S(x,-)$



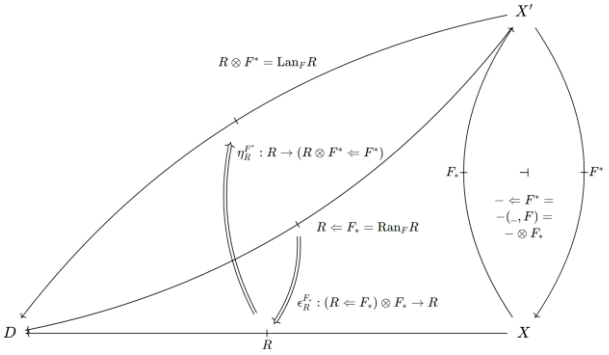
$S = (T \Rightarrow R)(\_,-) = \Pi_{d:D}\ T(d,\_) \to R(d,-)$

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

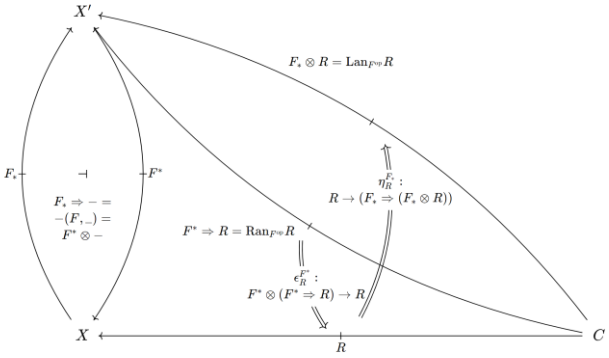


$$T = (R \Leftarrow S)(\_,-) = \Pi_{c:C}\ R(\_,c) \leftarrow S(-,c)$$

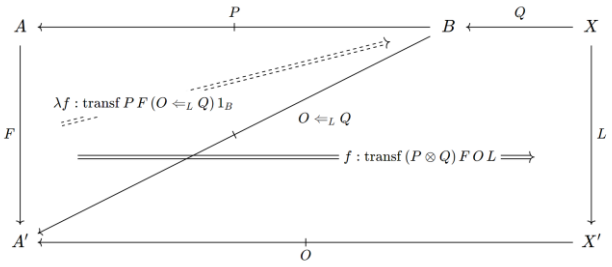
$$- \otimes S \dashv - \Leftarrow S \quad \text{covariant imply (“right extension”)}$$



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



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



9 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: catB → catA, right adjoint functor G: catA → catB, counit transformation ϕ_A: $F\ G\ A \to A$ (where among many formulations, A could be seen as a *parameter* functor exposing a *variable* X with ϕ^A_X: $F\ (G\ A\ X) \to A\ (X)$, that is ϕ^A_X: $\hom_{\mathrm{catA}(F,A)}\ (G\ A\ X)\ (X)$), and unit transformation γ_B: $B \to G\ F\ B$, 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):

$$f_2 \circ (f_1)\text{“}A \circ \phi \circ F\text{”} = (f_2 \circ f_1)\text{“}A \circ \phi \circ F\text{”}$$

$$\text{“}A \circ \phi \circ F\text{”}(g_1)\text{“} \circ F\text{”} \circ g_2 = \text{“}A \circ \phi \circ F\text{”}(g_1 \circ g_2)$$

- *Naturality rules*:

$$(f_1)\text{“}A \circ \phi \circ F\text{”} \circ ((f_2)\text{“}GA \circ 1\text{”}) = ((f_1)\text{“}1 \circ 1\text{”} \circ f_2)\text{“}A \circ \phi \circ F\text{”}$$

$$f_2 \circ \text{“}A \circ \phi \circ F\text{”}(g_1) = \text{“}A \circ \phi \circ F\text{”}((f_2)\text{“}GA \circ 1\text{”} \circ g_1)$$

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

$$(\text{“}1 \circ FB\text{”}(g_2)) \circ \text{“}G \circ \gamma \circ B\text{”}(g_1) = \text{“}G \circ \gamma \circ B\text{”}(g_2 \circ \text{“}1 \circ 1\text{”}(g_1))$$

$$(\text{“}1 \circ B\text{”}(g_2)) \circ \text{“}1 \circ B\text{”}(g_1) = \text{“}1 \circ B\text{”}(g_2 \circ \text{“}1 \circ 1\text{”}(g_1))$$

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

$$(f_1)\text{“}A \circ \phi \circ F\text{”}((f_2)\text{“}G \circ \gamma \circ B\text{”}(g_1)) = (f_1)\text{“}A \circ 1\text{”}((f_2)\text{“}1 \circ FB\text{”}(g_1))$$

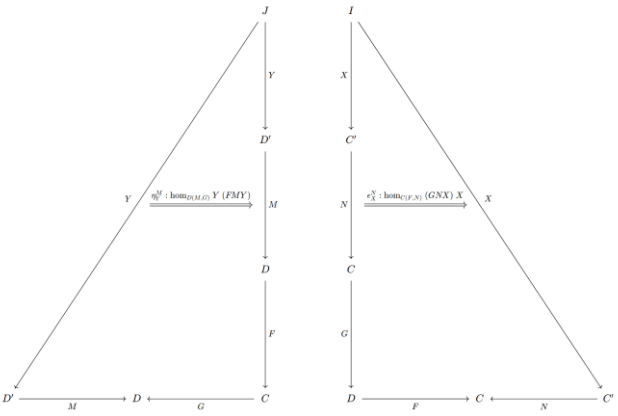
With (substructural) variations such as:

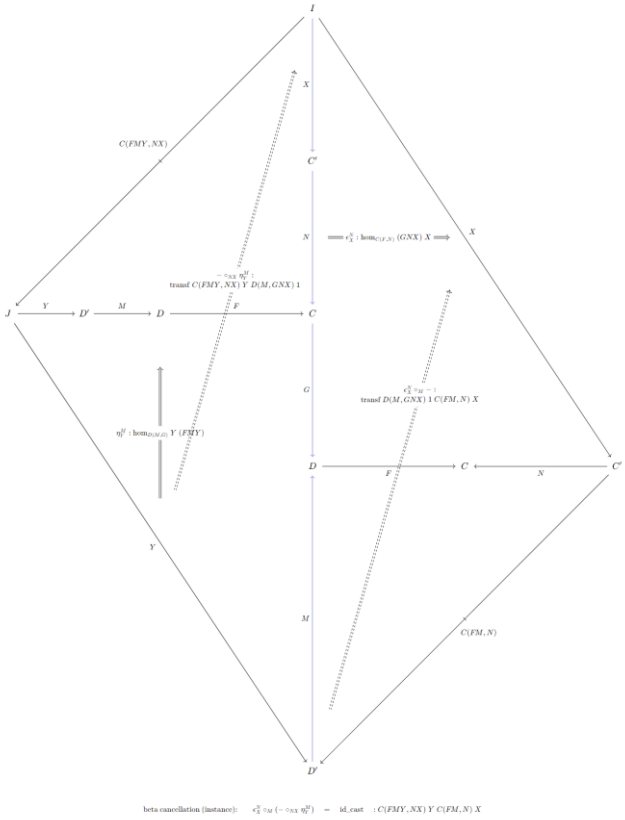
$$\text{“}A \circ \phi \circ F\text{”}((f_2)\text{“}G \circ \gamma \circ B\text{”}) = f_2$$

$$\text{“}1 \circ \phi \circ F\text{”}(\text{“}G \circ \gamma \circ B\text{”}g_2) = \text{“}1 \circ FB\text{”}g_2$$

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 : Π [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 : Π [L R : cat] [LAdj_func : func R L] [RAdj_func : func L R] (aj : adj
LAdj_func RAdj_func), Π [I] (Z : func I R) [J] (N : func J I),
  hom N (Unit_mod Z RAdj_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')   ⊢ eq_refl _ : π (

 ((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
```





10 Review of synthetic fibred Yoneda.

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

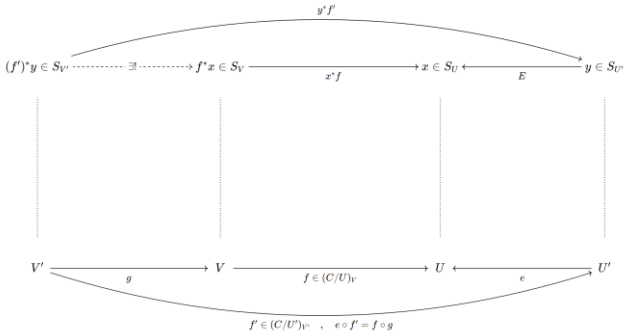
$$Mor_{Fib/C}\,(C/U\,,S) \longrightarrow S_U$$

given by $G \mapsto G(id_U)$ is an equivalence, whose inverse:

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

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

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



11 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 : Π [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),

Π (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 : Π [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' '◦ ((x'0x)_'◦> f))) HH (Unit_modd (GG ◦>d (Fibre_elim_funcd II G)) Id_funcd)
((Fibre_elim_funcd JJ (x''x')) ◦>d FF) →
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 ◦>d'_ : Π [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 →
Π [J: cat] [M : func J Y] [JJ : catd J] (MM : funcd JJ M B),
transfd ( r ◦>'_(M) ) (Unit_modd GG MM) FF (RR d<<◦ 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:

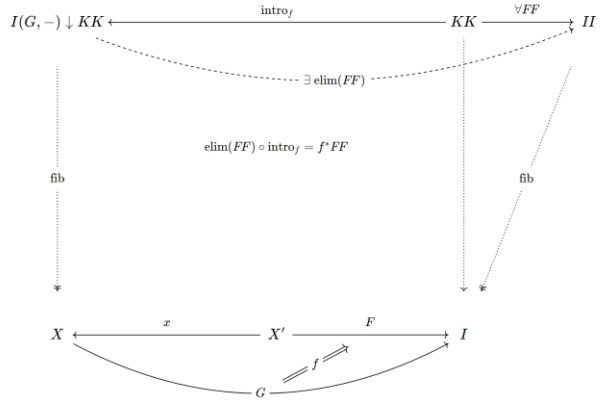
$$S_U \longrightarrow Mor_{Fib/C}\,(C/U\,, S)$$

12 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 : Π [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 : Π [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).



13 Cut-elimination for fibred arrows.

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

$$p\colon X \to \boldsymbol{Y} \text{ then } q\colon \boldsymbol{Y} \to Z$$

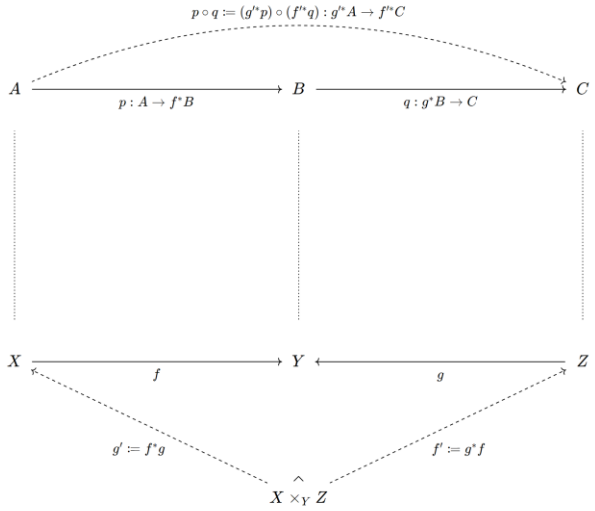
Fibred composition cut-elimination should also consider pairs of arrows:

$$p\colon A \to f^*\boldsymbol{B} \text{ then } q\colon g^*\boldsymbol{B} \to C$$

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* ($\Pi\Sigma = \Sigma\Pi\varepsilon^*$):

$$r\colon g^*A \longrightarrow f^*Z$$

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



14 Pi-category, Sigma-category, Pi-profunctor, Sigma-profunctor.

The implementation of the Sigma-sum and the Pi-product of fibred categories/profunctors requires a (computational) implementation of the (usually semantic) context-extension operation to manage dependent-type contexts. Here is a version of the implementation in the introduction paragraphs, but for a type theory for categories, functors and profunctors:

```
injective symbol Context_cat : Π [X : cat], catd X → cat;

injective symbol Context_elimCat_func : Π [X : cat] (A : catd X), func (Context_cat A) X;

injective symbol Context_elimCatd_funcd : Π [X : cat] (A : catd X), funcd (Terminal_catd _)
(Context_elimCat_func A) A;

injective symbol Context_intro_func : Π [X Y : cat] [A : catd X] [B : catd Y] [xy : func X Y],

funcd A xy B → func (Context_cat A) (Context_cat B);


rule Context_cat (Terminal_catd $A) ↪ $A;

rule Context_elimCat_func (Terminal_catd $A) ↪ Id_func;

rule Context_elimCatd_funcd (Terminal_catd $A) ↪ Id_funcd;

rule Context_intro_func (Id_funcd)  ↪ Id_func

with Context_intro_func (Terminal_funcd $A $xy)  ↪ Context_elimCat_func $A ∘> $xy;


symbol Context_intro_single_func [Y : cat] [B : catd Y] [X] (xy : func X Y)

(FF : funcd (Terminal_catd X) xy B) : func X (Context_cat B)

≔ @Context_intro_func _ _ (Terminal_catd _) _ _ FF;

symbol Context_intro_congr_func [Y : cat] [B : catd Y] [X] (xy : func X Y)

: func (Context_cat (Fibre_catd B xy)) (Context_cat B)

≔ Context_intro_func (Fibre_elim_funcd B xy);
```

```
rule (@Context_intro_func _ _ $A $B $F $FF) ∘> (Context_elimCat_func $B)

↪ (Context_elimCat_func $A) ∘> $F;

rule (Terminal_funcd (Terminal_catd $X) (@Context_intro_func _ _ (Terminal_catd $X) $B $xy $FF)) ∘>d
(Context_elimCatd_funcd $B)  ↪ $FF;

rule Context_intro_func (Context_elimCatd_funcd $A) ↪ rule_Context_cat_Terminal_catd_func $A;

rule @Context_intro_func _ _ _ _ $xy (Terminal_funcd (Terminal_catd _) _) ↪ $xy;

rule (@Context_intro_func $X $Y $A $B $F $FF) ∘> (@Context_intro_func $Y $Z $B $C $G $GG)

↪ Context_intro_func ($FF ∘>d $GG)

with $z ∘> @Context_intro_func _ _ (Terminal_catd _) _ _ $FF

↪ Context_intro_func ( (Terminal_funcd (Terminal_catd _) $z)  ∘>d $FF );
```

Then the implementation of the Sigma-sum (for isofibrations, but also along fibrations) and the Pi-product (for isofibrations, but also along opfibrations …) of fibred categories/profunctors (with beck-chevalley-commutation along fibres), is as follows:

```
injective symbol Sigma_catd : Π [X : cat] (F : catd X) (Z : catd (Context_cat F)), catd X;

injective symbol Sigma_intro_funcd : Π [X : cat] (F : catd X) (Z : catd (Context_cat F)),

funcd Z (Context_elimCat_func F) (Sigma_catd F Z);

injective symbol Sigma_elim_funcd : Π [X : cat] (F : catd X) [Z : catd (Context_cat F)] [X'] (G :
func X X') [C : catd X'], funcd Z ((Context_proj_func F) ∘> G) C → funcd (Sigma_catd F Z) G C;

injective symbol Pi_catd : Π [X : cat] (F : catd X) (Z : catd (Context_cat F)), catd X;

injective symbol Pi_intro_funcd : Π [X' : cat] (C : catd X') [X : cat] (G : func X' X) (F : catd X)
[Z : catd (Context_cat F)]

(CC : funcd (Fibre_catd C (Context_elimCat_func (Fibre_catd F G))) (Context_intro_func
(Fibre_elim_funcd F G)) Z),

funcd C G (Pi_catd F Z);

injective symbol Pi_elim_funcd : Π [X : cat] [F : catd X] [Z : catd (Context_cat F)] [X'] [x : func
X' X] [E : catd X'], funcd E x (Pi_catd F Z) → funcd (Fibre_catd E (Context_proj_func (Fibre_catd F
x))) (Context_func (Fibre_elim_funcd F x)) Z;
```

15 Sets, underlying groupoids, core, univalent categories.

The first question is: 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/quantales instead of mere sets could be investigated.

The next question is: Can those ambient sets be upgraded to groupoid (i.e., sets whose elements have intrinsic symmetry)? In reality, recall that for an intensional Martin-Lőf's type theory, the (infinity) groupoid interpretation (a.k.a. "homotopy type theory") is more natural than an attempt to truncate everything down to sets. The situation in this new type theory is now that the operation of recasting any "discrete set/groupoid" as a category (type_cat —) is left adjoint to the operation of taking the "underlying set/groupoid of objects" of any category (func Terminal_cat —); and the resulting comonad (iso_cat —) is usually known as the "groupoidal core" of the category. In short: Dosen's type theory subsumes "homotopy type theory", and the implementation of (axiomatic) univalent categories becomes:

```
constant symbol = [a] : τ a → τ a → Type /* not Prop */; notation = infix 10;

constant symbol eq_refl [a] (x:τ a) : τ (x = x);
```

```
constant symbol ind_eq [a] [x y:τ a] : τ (x = y) → Π p, τ (p y) → τ (p x);


constant symbol type_cat : Type → cat;

symbol type_cat_intro_func : Π [I : Type] [A : cat], func (type_cat I) A → (τ I → func Terminal_cat A); /* _→_ is groupoidal functor for _=_ */

injective symbol type_cat_elim_func : Π [I : Type] [A : cat], (τ I → func Terminal_cat A ) → func (type_cat I) A;

injective symbol type_cat_functorial : Π [I J : Type], (τ I → τ J) → func (type_cat I) (type_cat J);

rule type_cat_functorial $ij ↪ type_cat_elim_func (λ i, type_cat_intro_func Id_func ($ij i));


injective symbol iso_cat : Π (A : cat) , cat;

rule iso_cat $A ↪ type_cat (func_Type Terminal_cat $A);

injective symbol iso_elim_func : Π (A: cat), func (iso_cat A) A;

rule iso_elim_func $A ↪ type_cat_elim_func (λ F, F);

injective symbol iso_intro_func : Π [A: cat] [I : Type], func (type_cat I) A → func (type_cat I) (iso_cat A);

rule iso_intro_func $F ↪ type_cat_functorial (type_cat_intro_func $F);


injective symbol iso_inv_hom: Π [A: cat] [X : func Terminal_cat (iso_cat A)] [Y : func Terminal_cat (iso_cat A)],

hom Id_func (Unit_mod X Id_func) Y → hom Id_func (Unit_mod Y Id_func) X;

constant symbol iso_intro_hom : Π [A: cat]  [X Y : func Terminal_cat A],

Π f : hom Id_func (Unit_mod X Id_func) Y, Π f' : hom Id_func (Unit_mod Y Id_func) X,

τ (((f) ∘>'_(_)) ∘' f' = Func_con_hom X Id_func )

→ τ (((f') ∘>'_(_)) ∘' f = Func_con_hom Y Id_func )

→ hom Id_func (Unit_mod (type_cat_intro_func Id_func X) Id_func) (type_cat_intro_func Id_func Y);


injective symbol type_cat_intro_hom : Π [I : Type]  [X Y : τ I], // (I ≔ func_Type Terminal_cat A)

τ (X = Y) → hom Id_func (Unit_mod (type_cat_intro_func Id_func X) Id_func) (type_cat_intro_func Id_func Y);

rule type_cat_intro_hom (eq_refl _) ↪ Func_con_hom _ _;

injective symbol univalent_iso_cat_eq : Π [A: cat]  [X Y : func Terminal_cat (iso_cat A)],

hom Id_func (Unit_mod X Id_func) Y →  τ (X ∘> iso_elim_func _ = Y ∘> iso_elim_func _);


rule univalent_iso_cat_eq  (type_cat_intro_hom $e) ↪ $e;
```

```
rule type_cat_intro_hom (@univalent_iso_cat_eq _ (type_cat_intro_func Id_func $X)
(type_cat_intro_func Id_func $Y)  $f) ↪ $f;
```

16 Container sets, polynomial functors, categories as polynomial comonoids.

Another similar question is: Can those ambient sets be upgraded to polynomials (i.e., sets whose elements are container for elements of a parameter set)? A fundamental explanation of polynomials comes from the *dualities in the many ways to store the data info of a category*. Recall, from Spivak's (2023) "*Functorial Aggregation*" [0], 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 $p : d\text{-Set} \to c\text{-Set}$, written as $p : c \nleftarrow d$ or $p : c\text{-Set}[d]$, which is some sum of representables; i.e., there exists

- some set/diagram $\mathrm{Ob}(p) \in c$-Set (covariant over c), and
- some sets/diagrams $p[i] \in d$-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:

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

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:

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

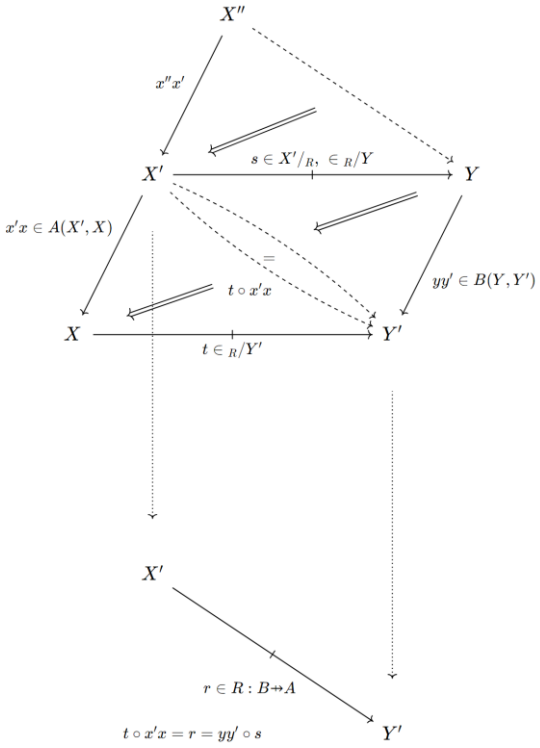
And the goal here is to express the computational interface of the polynomial operations: composition/substitution, coclosure, local monoidal closure, cofunctors, categories as polynomial substitution-comonoids, etc.

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

injective symbol poly_mod : Π [A : cat] [PA : mod Terminal_cat A] [B : cat], poly PA B → Π [I] (f :
func I (Elements_cat PA)), mod I B;

rule  $i ∘>> poly_mod $R $f ↪ poly_mod $R ($i ∘> $f);

injective symbol ↘poly_base [A B : cat] [PA : mod Terminal_cat A] (R : poly PA B):

      mod Terminal_cat B → mod Terminal_cat A

≔ λ PB, ((PB ⇐ (poly_mod R Id_func)) ⊗ (Unit_mod (Elements_proj_func PA) Id_func));

constant symbol ↘poly : Π [A B X : cat] [PA : mod Terminal_cat A], Π (R : poly PA B), Π [PB : mod
Terminal_cat B], Π (S : poly PB X), poly (R ↘poly_base PB) X;

constant symbol ↘poly_intro_hom : Π [A B C : cat] [PA : mod Terminal_cat A], Π (R : poly PA B),

Π [PB : mod Terminal_cat B], Π (S : poly PB C), Π [I] [f : func I (Elements_cat PA)]

(k : hom (Terminal_func I) (PB ⇐ (poly_mod R f) ) Id_func)

[G : func I A] (a : hom f (Unit_mod (Elements_proj_func PA) Id_func) G),

Π [K : func I C], Π [H : func I B] (r : hom Id_func (poly_mod R f) H),

   hom Id_func (poly_mod S (Elements_intro_func ((Eval_cov_hom_transf k) ∘' r))) K →
```

```
hom (Elements_intro_func (Tensor_cov_hom_hom _ k a)) (poly_mod (R ↘poly S) Id_func) K;
```

17 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 (Terminal_catd B)
s)
  ((Comma_con_comp_funcd R (Terminal_catd B) F)
◦>>d ((Comma_modd (Triv_catd A) R
(Terminal_catd B)) d<<◦ (Comma_cov_comp_funcd
(Terminal_catd A) R G)))
  (Comma_cov_intro_funcd (Terminal_catd A) t);
```

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

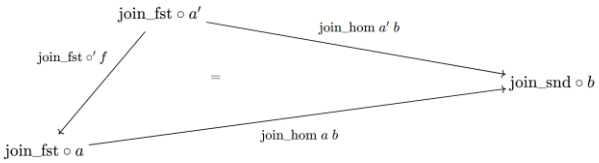
For witnessing the (no-computational-content) pasting along the diagonal, this implementation uses for the *LambdaPi-metaframework's equality predicate* which internally-reflects all the conversion-rules; in particular the implementation uses here *the categorial-associativity equation axiom, which is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

18 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);
```



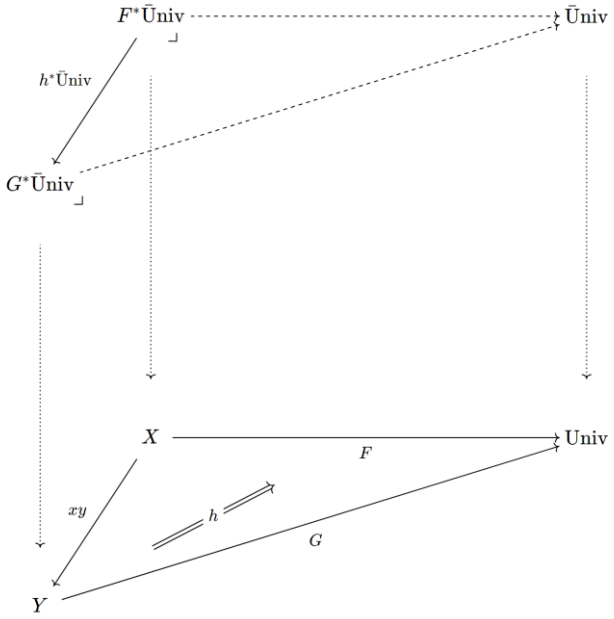
But for the elimination rules, those (3-dimensional) naturality cone conditions are expressed using the LambdaPi equality predicate ≔ (which internally-reflects its rewrites rules), to relate the (2-dimensional) arrows arguments. Note that the naturality cone conditions *carry no computational content*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

```
symbol join_elim_con_func : Π (A B : cat) [E : cat] (first_func : func A E)  (second_func : func B E)
(one_hom : Π (I : cat) (a : func I A) (b : func I B), hom a (Unit_mod (first_func) Id_func)
(second_func <◦ b))
(natural_eq : Π [I : cat] (a : func I A) (b : func I B) [a'] (r : hom a' (Unit_mod Id_func a)
Id_func),
      π (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 _) ↪ $F0
with  join_snd_func $A $B ◦> (join_elim_con_func $A $B $F0 $F1 $r _) ↪ $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)))) ↪ $one_hom _ $a $b ;
```

19 Universe, universal fibration.

This article implements a *grammatical (univalent) universe* and the universal fibration classifying small fibrations, together with the dual universal opfibration. 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 : Π [X : cat] (A :
catd X) (A_isf : isFibration_con A), func  X
Universe_con_cat;

symbol Universe_con_funcd : Π [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 : Π [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) ↠
hom xy (Unit_mod G Id_func) F;
```

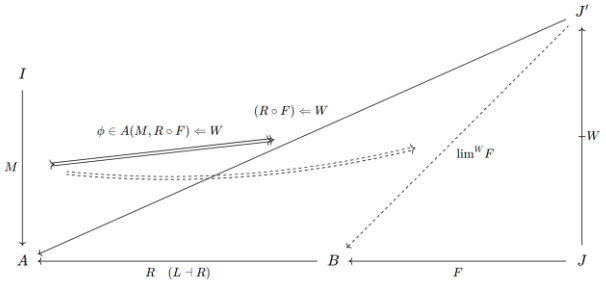


20 Weighted limits.

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

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.*

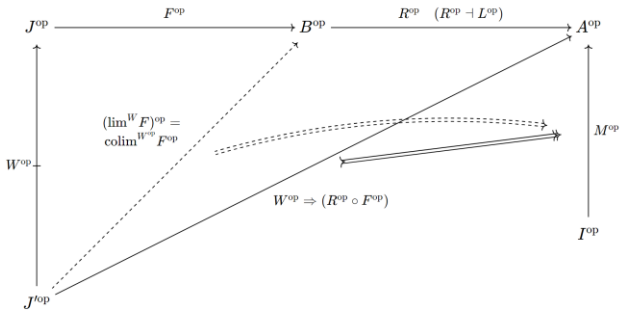
A computational-proof is to be contrasted from a logical deduction which uses the reflected/internalized LambdaPi propositional equality.




```
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 righ_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)) ∘''
             (Imply_cov_transf ((M)_'∘> Adj_cov_hom isa F Id_func) (Id_transf W)));
```

21 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));
```

22 Grammatical topology, sheaves, locally-ringed sites, schemes.

Finally, there is an 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)...

Firstly, a glue operation for any sheaf `S` over the sheafification modality `mod_smod` is declared:

```
constant symbol glue : Π [A B : cat] (A_site : site A)  [S : smod A_site B] [I : cat] [G : func I B] (L : mod A I),

transf L Id_func (smod_mod S) G

→  transf (smod_mod (mod_smod A_site L)) Id_func (smod_mod S) G;
```

From which a glue operation for any sheaf `S` over the sieve-closure modality `sieve_ssieve` is defined, where `sieve` is the presheaf (profunctor …) which classifies sieves:

```
symbol glue_sieve_mod_def : Π [A B : cat] (A_site : site A) [I : cat] [F : func I A]  [S : smod A_site B]  [G : func I B] [D : cat] [K : func I D] (ff : hom F (sieve A D) K),

transf (sieve_mod ff) Id_func (smod_mod S) G)

→  transf (smod_mod (ssieve_smod ((ff '∘ (sieve_ssieve A_site _))))) Id_func (smod_mod S) G ≔ glue …;
```

Now a transformation (predicate) into the sieves-classifier (truth-values) `sieve` corresponds to a subprofunctor (fibred/dependent profunctor), such as the maximal sieve or the intersection sieve, but also a novel “sub sieve” construction.

And when the sieve-closure `sieve_ssieve` happens to evaluate to the maximal sieve, then this glue operation is indeed the expected inverse operation of the Yoneda action by sieve-elements. In short: Nicolas Tabareau ″Lawvere-Tierney sheafification in homotopy type theory” [0] is only a formalization of the semantics of sheafification, not an actual computational logic; and moreover, its Definition 5.2 causes flaws: instead of “a subobject of E is dense in E if …”, it should be “a subobject P of E is dense in another subobject Q of E if …”.

Next, the presentation of affine schemes is such that it exposes a logical interface/specification which computes; for example the structure sheaf `ascheme_mod_ring_loc` localized away from r : R has restrictions along D(s·r) ⊆ D(r) (and, substructurally, along explicit radicals D(r) ⊆ D(r^n) …) which compute (by the usual recasting of any sheaf element `f/r^n` as `(s^n)·f/(s·r)^n`):

```
rule (@ascheme_mult_hom $Ml $Af _ $U $s $r) '∘ (_) _'∘> (ring_loc_intro (ascheme_mod_ring_loc $Af $r) $f $n)

↪ ring_loc_intro (ascheme_mod_ring_loc $Af (ring_mult (mod_loc_ring $Ml $U) $s $r)) (ring_mult (mod_loc_ring $Ml $U) (ring_exp (mod_loc_ring $Ml $U) $s $n) $f) $n;
```

And similar computations have been implemented for formal (colimits) joins D(f)∨D(g) of basic opens. But Joyal’s covering axiom D(s+r) ⊆ D(s)∪D(r) of the basic open D(s+r), has been reformulated approximately as a cover by the inclusions D(s·√(a·s + b·r)) ⊆ D(√(a·s + b·r)) (similarly for r) such to simultaneously also handle the unimodular (that is, ⟨r, s⟩ = 1) cover of a basic open. By the way, this functorial double-category framework is justified even for (the limit of) any functor (diagram) of rings, rather than a single ring, because localization is left-exact …

Then, there is an implementation of locally ringed sites and schemes. But the traditional definition of schemes via isomorphisms to affine schemes won’t work computationally; instead, one has to declare that the slice-category sites of the base site satisfy the interface/specification of an affine scheme. But what is a slice-category site? How does its glue operation relate to the glue operation of the base site? This has required subtle reformulations of a continuous(-and-cocontinuous) morphism of sites with a continuous right adjoint [0] (with “technical lemma 7.20.4 00XM”):

```
rule site_morph_mod_adjL $Sm $R (glue $r_)

↪ glue (adj_mod_adjL (site_morph_adj $Sm) (smod_mod $R) $r_);
```

But then, what is the invertibility-support D(–) for a locally ringed site and how does it relate to each affine-scheme’s invertibility-support D(–) in the slice-categories. The author of cartierSolution16.lp claims that D(f) is intended to be the SIEVE (generated by a singleton when

restricted to a slice affine-scheme …) under U of all opens V where the restriction of the function f: O(U) becomes invertible, together with (a computational reformulation of) the limit condition:

lim_{V:D(f)} O(V) = O(U)[1/f]

In short, some structured data is being transferred from a base scheme to its slice-categories where they are required to satisfy the affine-scheme interface.

Moreover, the affine-scheme interface is coinductive (self-reference), meaning that its slice-categories are also required to satisfy the affine-scheme interface. This slice coinduction/self-reference is deeply intrinsic and unavoidable; so that all the constructions are always relativized (in the slice-category under an object). That is, even the affine-scheme invertibility-support D(–) constructors `ascheme_inv_slice_func`, which are the basic objects which generate the topology, are also relativized by-definition:

```
constant symbol ascheme_inv_slice_func : Π [Ml : struct_mod_loc] (Af : ascheme Ml), Π [I] [U : func I
(mod_loc_cat Ml)] (r : hom U (smod_mod (mod_loc_smod Ml)) (Terminal_func _)), func I (slice_cat U);


constant symbol scheme_slice_ascheme : Π [Ml : struct_mod_loc] [Cs : struct_cov_sieve (mod_loc_site
Ml)] (Sc : scheme Ml Cs), Π [U : func (cov_sieve_cat Cs) (mod_loc_cat Ml)] (u : hom U (sieve_mod
(cov_sieve_hom Cs)) Id_func),

ascheme (@Struct_mod_loc (slice_cat U) (slice_site (mod_loc_site Ml) U)

  (site_morph_pullback_smod (scheme_site_morph Sc U) (mod_loc_smod Ml))

  (mod_pullb_mod_ring (mod_loc_mod_ring Ml) (slice_proj_func U))

  (mod_pullb_mod_loc (mod_loc_mod_loc Ml) (slice_proj_func U)));
```

A consequence of this formulation is that these various structure sheaves are now canonically related, beyond the mere usual knowledge that for any ring `R`: R[1/fg] ≅ R[1/f][1/g]

MAX Zeuner "Univalent Foundations of Constructive Algebraic Geometry" [0] recently defended an equivalence between functorial schemes `X` and locally-ringed-lattice schemes `Y`, approximately:

LRDL^op(X ⇒ Spec , Y) ≅ (X ⇒ LRDL^op(Spec(–), Y))

where (X ⇒ X') := Fun(CommRing , Set)(X , X') and Spec: Fun(CommRing^op, LRDL^op). But the author of cartierSolution16.lp claims that their profunctor framework should allow a hybrid handling of schemes as locally ringed sites together with their functor-of-points semantics. And most importantly, it is not necessary to try and express those things using an internal logic (not truly-computational …) within the Zarisky topos, in the style of Thierry Coquand "A foundation for synthetic algebraic geometry" [0].

Ultimately, it is a conjecture that this new framework could solve the open problem of discovering a (graded) differential linear logic formulation for the algebraic-geometry's cohomology differentials via the profunctorial semantics of linear logic in the context of sieves as profunctors…

It is also a conjecture that this new framework could solve the search of a hybrid framework combining polynomial functors (good algebra) "depending" on analytic functors (good logic) as motivated by Ehrhard's "An introduction to differential linear logic: proof-nets, models and antiderivatives" [0, section 3.1.1, page 44] outrageous definition of the composition of polynomials by the use of differentials instead of by elementary algebra.

Another open question: would such an algebra-independent computational-logical interface for commutative (affine) schemes be able to also specify schemes of (noncommutative) associative algebras; for example, in the sense of Siqveland Arvid "Schemes of Associative algebras" [0]?

23 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 :  Π [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)
```

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…

24 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.

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, is not yet implemented in full generality (when the acting-categories are also fibred); also, some details about the logical-properties content of concrete limits, besides their data content, have been skipped. The future sub-goals should be to develop urther 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 and "homotopy type theory". This goal opens the possibility to standardize the contacting of (VIP) researchers/businesses by the (general-public) reviewers whose side-goals are to co-author or by-product more intelligent (AI, proof assistant) interfaces for their papers/apps API.

25 Discussion: Qualifier for editorial review.

The ability-or-not of proof-and-AI-assistants to intelligently search at the interface of 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. An implementation of this methodology is at: < re365.net > Now by the nature that the results here are computer implementations, they have no absolute-mistake; any dissenting opinion could only be about the specification of the goal, relative to which these results are no longer correct nor complete. Here is a test question to qualify that any editorial reviewer (AI or "expert") is in agreement with our specification of the goal:

• 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.

26 References

[0] https://www.bing.com/search?q=
[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] Christopher Mary. "Functorial programming for polynomial functors, categories as polynomial comonoids*"* (2010)
https://github.com/1337777/cartier/blob/master/cartierSolution15.lp
[9] Pierre Cartier.