A new implementation of dependent types via Dosen's substructural categorial programming: example of the Yoneda lemma for fibrations



Cut-elimination in the double category of fibred profunctors with inner cut-eliminated adjunctions:

<https://github.com/1337777/cartier/blob/master/cartierSolution13.lp>

Applications: datatypes or 1+2=3 via 3 methods: natural numbers category via intrinsic types, natural numbers object via adjunctions, and category of finite sets/numbers via colimits:

<https://github.com/1337777/cartier/blob/master/cartierSolution14.lp>

Outline.

* Categories, functors, profunctors, hom-arrows, transformations.
* Composition is Yoneda “lemma”.
* Outer cut-elimination or functorial lambda calculus.
* Inner cut-elimination or decidable adjunctions.
* Synthetic fibred Yoneda.
* Substructural fibred Yoneda.
* Comma elimination (“J-rule arrow induction”).
* Cut-elimination for fibred arrows.
* Pi-category-of-fibred-functors and Sigma-category.
* And why profunctors (of sets)?
* What is a fibred profunctor anyway?
* Higher inductive types, the interval type, concrete categories.
* Universe, universal fibration.
* Weighted limits.
* Duality Op, covariance vs contravariance.
* Grammatical topology.
* Applications: datatypes or 1+2=3 via 3 methods: nat numbers category, nat numbers object and colimits of finite sets.
* References.

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

• This text 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 text 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 text also implements a *grammatical (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...

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

;

; ’

• Attempt for left-associativity normalization CHOICE A:

✗ KO, unless “accumulate/yoneda” trick to control associativity:

.

✓ OK by naturality.

• Attempt for right-associativity normalization CHOICE B:

OK by naturality.

✗ KO, unless “accumulate/yoneda” trick to control associativity:

.

Motivation 2.

• How to write the (co)unit transformation of an adjunction between a left adjoint functor and right adjoint functor ? 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: where is any *variable*.

CHOICE B: where is profunctor.

CHOICE C: .

CHOICE D: where is an extra *parameter*; also, CHOICE B’ with extra parameter, etc.

BAD CHOICE: .

• 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 , get

the (covariant) accumulating operation: for , get

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 instead of via their usual objects “”.

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);

[A diagram of a cat

Description automatically generated](https://q.uiver.app/#q=WzAsMyxbNCwzLCJCIDogXFxtYXRocm17Y2F0fSJdLFswLDMsIkEiXSxbMiwwLCJJIDogXFxtYXRocm17Y2F0fSJdLFswLDEsIlIgOiBcXG1hdGhybXttb2R9XFwsIEFcXCwgQiIsMCx7InN0eWxlIjp7ImJvZHkiOnsibmFtZSI6ImJhcnJlZCJ9fX1dLFsyLDEsIkYiLDJdLFsyLDAsIkcgOiBcXG1hdGhybXtmdW5jfSBcXCwgSSBcXCwgQiJdLFs0LDUsInIgOiBcXG1hdGhybXtob219XFwsIEZcXCwgUlxcLCBHIiwyLHsic2hvcnRlbiI6eyJzb3VyY2UiOjEwLCJ0YXJnZXQiOjEwfX1dXQ==)

[A diagram of a cat

Description automatically generated](https://q.uiver.app/#q=WzAsNCxbNCwzLCJCIDogXFxtYXRocm17Y2F0fSJdLFswLDMsIkEiXSxbNCwwLCJCJyA6IFxcbWF0aHJte2NhdH0iXSxbMCwwLCJBJyJdLFswLDEsIlIgOiBcXG1hdGhybXttb2R9XFwsIEFcXCwgQiIsMCx7InN0eWxlIjp7ImJvZHkiOnsibmFtZSI6ImJhcnJlZCJ9fX1dLFsyLDAsIkcgOiBcXG1hdGhybXtmdW5jfSBcXCwgSSBcXCwgQiJdLFszLDEsIkYiLDJdLFsyLDMsIlInIDogXFxtYXRocm17bW9kfVxcLCBBJ1xcLCBCJyIsMix7InN0eWxlIjp7ImJvZHkiOnsibmFtZSI6ImJhcnJlZCJ9fX1dLFs2LDUsInJyIDogXFxtYXRocm17dHJhbnNmfVxcLCBSJ1xcLCBGXFwsIFJcXCwgRyIsMix7InNob3J0ZW4iOnsic291cmNlIjoxMCwidGFyZ2V0IjoxMH19XV0=)

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

,

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

,

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;

[A diagram of a triangle

Description automatically generated](https://q.uiver.app/#q=)

Outer cut-eliminatin 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:

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));

[A triangle with lines and letters

Description automatically generated](https://q.uiver.app/#q=WzAsNSxbNCwzLCJDIl0sWzAsMywiRCJdLFsyLDAsIlgiXSxbMCw0LCJcXDsiXSxbNCw0LCJcXDsiXSxbMCwxLCJSID0gVCBcXG90aW1lcyBTIiwwLHsic3R5bGUiOnsiYm9keSI6eyJuYW1lIjoiZGFzaGVkIn19fV0sWzAsMiwiUyIsMix7InN0eWxlIjp7ImJvZHkiOnsibmFtZSI6ImJhcnJlZCJ9fX1dLFsyLDEsIlQiLDIseyJzdHlsZSI6eyJib2R5Ijp7Im5hbWUiOiJiYXJyZWQifX19XSxbMyw0LCJSID0gKFQgXFxvdGltZXMgUykoXFxfLC0pID1cXFNpZ21hX3t4Olh9IFxcLFxcO1QoXFxfLHgpIFxcdGltZXMgUyh4LC0pIiwwLHsic3R5bGUiOnsiYm9keSI6eyJuYW1lIjoibm9uZSJ9LCJoZWFkIjp7Im5hbWUiOiJub25lIn19fV1d)

[A diagram of a triangle with lines and letters

Description automatically generated](https://q.uiver.app/#q=)

[A diagram of a triangle with lines and letters

Description automatically generated](https://q.uiver.app/#q=)

[A diagram of a mathematical function

Description automatically generated](https://q.uiver.app/#q=)

[A diagram of mathematical equations

Description automatically generated](https://q.uiver.app/#q=)

[A diagram of a mathematical equation

Description automatically generated](https://q.uiver.app/#q=)

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 , right adjoint functor , counit transformation (where among many formulations, could be seen as a *parameter* functor exposing a *variable* with , that is ), and unit transformation , is formulated as rewrite rules from any redex outer cut on the left-side to the contractum containing some *smaller* inner cut, where are the (fixed) *parameters* and are the *natural variables*. Those rules are classified as:

* *Accumulation rules* (those accumulation equations can be formulated once generically for all such transformations):
* *Naturality rules*:
* And similarly, for the naturality of the adjunction unit transformation, and for the *functoriality/naturality* (besides the generic accumulation rules) of every functor:
* *Beta-cancellation conversion (or rewrite) rules* (i.e., ; the other half, *Eta-cancellation* is similar):

With (substructural) variations such as:

where indeed the functions on arrows and of those functors are not primitive but are themselves the (Yoneda) “antecedental/consequential transformation” formulations “” or “” 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

[A diagram of a triangle

Description automatically generated](https://q.uiver.app/#q=)

[A diagram of a triangle with lines and dots

Description automatically generated](https://q.uiver.app/#q=)

[A white background with lines and dots

Description automatically generated](https://q.uiver.app/#q=)

A diagram of a triangle

Description automatically generated

[A diagram of a math problem

Description automatically generated](https://q.uiver.app/#q=)A diagram of a triangle

Description automatically generatedA diagram of a triangle

Description automatically generated[A diagram of a triangle

Description automatically generated](https://q.uiver.app/#q=)

[A diagram of a triangle

Description automatically generated](https://q.uiver.app/#q=)

Synthetic fibred Yoneda.

**Lemma 4.41.1** (2-Yoneda lemma for fibred categories). Let be a category. Let be a fibred category over . Let . The functor

given by is an equivalence, whose inverse:

is such that given the associated functor is:

* on objects: , and
* on morphisms: the arrow maps to the universality morphism fibred over the arrow .

=]

[A black and white image of a oval with a white background

Description automatically generated](https://q.uiver.app/#q=)

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 or the inner functorial-action by (or both simultaneously) in the mapping:

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).

[A diagram of a function

Description automatically generated](https://q.uiver.app/#q=)

Cut-elimination for fibred arrows.

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

then

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



then

• 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* ():

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

[A diagram of a circle with lines and arrows

Description automatically generated](https://q.uiver.app/#q=)

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;

[A diagram of a triangle with text

Description automatically generated](https://q.uiver.app/#q=)

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/quantales instead of mere sets could be investigated.

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

[A diagram of a triangle with lines and points

Description automatically generated](https://q.uiver.app/#q=)

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 text 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!

[A diagram of mathematical equations

Description automatically generated](https://q.uiver.app/#q=)

Higher inductive types, the interval type, concrete categories.

• This text 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 ;

[A black and white image of a triangle

Description automatically generated](https://q.uiver.app/#q=WzAsMyxbMSwwLCJcXG1hdGhybXtqb2luXFxfZnN0fSBcXGNpcmMgYSciXSxbNiwxLCJcXG1hdGhybXtqb2luXFxfc25kfSBcXGNpcmMgYiJdLFswLDIsIlxcbWF0aHJte2pvaW5cXF9mc3R9IFxcY2lyYyBhIl0sWzAsMiwiXFxtYXRocm17am9pblxcX2ZzdH0gXFxjaXJjJyBmIiwyXSxbMiwxLCJcXG1hdGhybXtqb2luXFxfaG9tfVxcOyBhXFw7IGIiLDJdLFswLDEsIlxcbWF0aHJte2pvaW5cXF9ob219XFw7IGEnXFw7IGIiXSxbNSwyLCI9IiwyLHsic2hvcnRlbiI6eyJzb3VyY2UiOjIwLCJ0YXJnZXQiOjIwfSwic3R5bGUiOnsiYm9keSI6eyJuYW1lIjoibm9uZSJ9LCJoZWFkIjp7Im5hbWUiOiJub25lIn19fV1d)

Universe, universal fibration.

• This text 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;

[A diagram of a triangle

Description automatically generated](https://q.uiver.app/#q=)

Weighted limits.

• This text implements *profunctor-weighted limits* (that the *right-extension* is representable as ) and *profunctor-weighted colimits* (that the *right-lifting* is representable as ).

• 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)));

[A diagram of a mathematical equation

Description automatically generated](https://q.uiver.app/#q=)

Duality Op, covariance vs contravariance.

• This text 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));

[A diagram of mathematical equations

Description automatically generated](https://q.uiver.app/#q=)

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 where is the classifier of (sub-)objects (sieves) of the object , and where is the (opaque) set of witnesses that the pullback-sieve is covering. Then the transitivity axiom: if , iff , becomes expressible.

constant symbol **covering** : Π [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 → TYPE ;

constant symbol **coveringu** : Π [I : cat] [A : catd I] [B : catd I] (RR : moddu A B), funcd (Triv\_catd I) Id\_func B → TYPE ;

constant symbol **Total\_covering** : Π [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],  
 (Π [F : func I X] (r : hom F R G), **coveringu** (Fibre\_hom\_moddu RR r) (Fibre\_intro\_funcd \_ G Id\_func H )) →  
 **covering** RR H ;

constant symbol **Glue\_transfd** : Π [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'], Π [I : cat] [G : func I Y] [H : funcd (Triv\_catd I) G B] , **covering** RR H →  
(Π [F : func I X] (r : hom F R G), **transfduu** (Fibre\_hom\_moddu RR r)  
 (Fibre\_intro\_funcd \_ (F ∘> xx') Id\_func (Fibre\_elim\_funcd A F ∘>d FF))  
 (Fibre\_hom\_moddu (Sheaf\_con\_modd RR') (r '∘ rr'))  
 (Fibre\_intro\_funcd \_ (G ∘> yy') Id\_func (Fibre\_elim\_funcd B G ∘>d GG))) →  
**transfd** rr' RR FF (Sheaf\_con\_modd RR') GG;

[A diagram of a diagram

Description automatically generated](https://q.uiver.app/#q=)

• 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 Cech 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:

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

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 text 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.

• Via the natural numbers category:

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;

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))**

• Via the natural numbers object:

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;

compute λ C : cat, (iprod\_pair\_hom (Product\_pair\_func (itermin\_func C) (itermin\_func C))

(Func\_cov\_hom (itermin\_func C) Id\_func) (Func\_cov\_hom (itermin\_func C) Id\_func)) '∘ (\_)\_'∘>

((Id\_hom \_) ∘>'\_(\_) ∘'

((Product\_pair\_cov\_hom (Product\_pair\_func \_ \_)

(Product\_proj\_left\_func \_ \_) [Id\_func] (Product\_proj\_right\_func \_ \_) [Id\_func] Id\_func

(Succ\_inj\_inat\_hom C (Zero\_inj\_inat\_hom C)) (Succ\_inj\_inat\_hom C (Succ\_inj\_inat\_hom C (Zero\_inj\_inat\_hom C))))

'∘ (\_)\_'∘> (add\_inat\_hom C)));

// **λ C, Succ\_inj\_inat\_hom C (Succ\_inj\_inat\_hom C (Succ\_inj\_inat\_hom C (Zero\_inj\_inat\_hom C)))**

• Via limits/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. 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) ⸬ (Pair\_natUniv (Pair\_natUniv (Pair\_natUniv (Base\_natUniv 0) (Base\_natUniv 13)) (Base\_natUniv 22)) (Base\_natUniv 31) ⸬ (Pair\_natUniv (Pair\_natUniv (Pair\_natUniv (Base\_natUniv 0) (Base\_natUniv 12)) (Base\_natUniv 21)) (Base\_natUniv 31) ⸬ (Pair\_natUniv (Pair\_natUniv (Pair\_natUniv (Base\_natUniv 0) (Base\_natUniv 12)) (Base\_natUniv 22)) (Base\_natUniv 31) ⸬ (Pair\_natUniv (Pair\_natUniv (Pair\_natUniv (Base\_natUniv 0) (Base\_natUniv 11)) (Base\_natUniv 21)) (Base\_natUniv 31) ⸬ (Pair\_natUniv (Pair\_natUniv (Pair\_natUniv (Base\_natUniv 0) (Base\_natUniv 11)) (Base\_natUniv 22)) (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…

References.

[1] Dosen-Petric: Cut Elimination in Categories 1999;

[2] Proof-Theoretical Coherence 2004;

[3] Proof-Net Categories 2005;

[4] Coherence in Linear Predicate Logic 2007;

[5] Coherence for closed categories with biproducts 2022;

[6] Cut-elimination in the double category of fibred profunctors with inner cut-eliminated adjunctions:

<https://github.com/1337777/cartier/blob/master/cartierSolution13.lp>

[7] Pierre Cartier