emdash — Functorial programming for strict/lax ω-categories in Lambdapi

https://github.com/hotdocx/emdash

Abstract

We report on **emdash** https://github.com/hotdocx/emdash an ongoing experiment whose goal is a *type-theoretical* account of strict/lax ω-categories that is both *internal* (expressed inside dependent type theory) and *computational* (amenable to normalization by rewriting). The current implementation target is the Lambdapi logical framework, and the guiding stance is proof-theoretic: many categorical equalities are best presented as *normalization* ("cut-elimination") steps rather than as external propositions.

The core mathematical construction is a dependent comma/arrow ("dependent hom") operation that organizes "cells over a chosen base arrow" in a simplicial manner. Concretely, for a base category B and a dependent category over it (morally a fibration $E : B \to \mathbf{Cat}$), fix $b_0 \in B$ and $e_0 \in E(b_0)$ and define a Cat-valued functor:

$$\mathrm{Homd}_E(e_0, (-,-)) : E \times_B \big(\mathrm{Hom}_B(b_0, -)\big)^{\mathrm{op}} \to \mathbf{Cat}.$$

Its value at $(e_1 : E(b_1),\ b_{01} : b_0 \to b_1)$ is the fibre hom-category $\mathrm{Hom}_{E(b_1)}\big((b_{01})_! e_0,\ e_1\big)$. In the current kernel snapshot this is computational in the Grothendieck/Grothendieck probe case (`homd_` pointwise rule), while the full general normalization story is still in progress.

On the transformation side, we expose explicit off-diagonal components $\epsilon_{(f)} : F(X) \to G(Y)$ (indexed by arrows $f : X \to Y$) and orient naturality as *cut accumulation* rewrites. The current snapshot includes phase-2 draft strict naturality/exchange rules with sanity assertions. A key interpretive signal in the current bridge layer is: under simplicial iteration, an identity-like/cartesian source triangle may be mapped to a non-identity/non-cartesian target triangle (the laxness witness). As a complementary application, we outline a computational adjunction interface where unit/counit are first-class 2-cell data and a triangle identity is oriented as a definitional reduction on composites (schematically $\epsilon_f \circ L(\eta_g) \rightsquigarrow f \circ L(g)$). The development is diagram-first: commutative diagrams are specified in a strict JSON format (Arrowgram), enabling rendering, checking, and AI-assisted editing as part of a reproducible paper artifact.

From an engineering perspective, this fits a "MathOps" workflow: a long-running feedback loop between an LLM assistant and a proof-checker/type-checker, where commutative diagrams are first-class artifacts. In emdash we use Arrowgram (a strict JSON diagram format) to make diagrams AI-editable, renderable (e.g. to SVG), and checkable alongside the kernel and the paper.

1. Introduction

Formalizing higher categories in proof assistants is hard for two intertwined reasons:

1. *Size of coherence*: weak n-categories (especially at ω) come with a large amount of coherence data.

2. *Where equalities live*: many “structural equalities” in category theory are best treated as computation, not as external theorems.

   emdash explores a kernel design where:

- “category theory” is internal: we work inside dependent type theory with classifiers `Cat` (directed structure) and `Grpd` (paths),
- many categorical laws are definitional: enforced by rewrite rules and unification hints, so normalization *is* diagram chasing.

1.1 Contributions

1. **Rewrite-head discipline for categorical computation.** Structural equalities are oriented as normalization steps on stable head symbols (e.g. `comp_fapp0`, `fapp1_fapp0`, `tapp0_fapp0`, `tapp1_fapp0`) rather than proved externally.
2. **A simplicial “triangle classifier” over base arrows.** The dependent arrow/comma construction `homd_` (and the internal pipeline `homd_int_base`) provides a computational home for “cells over a chosen base edge” (triangles/surfaces), and a compositional target for iteration.
3. **Explicit off-diagonal interfaces for transfors (ordinary and displayed).** We expose diagonal components (`tapp0_*`, `tdapp0_*`) and off-diagonal arrow-indexed components (`tapp1_*`, `tdapp1_*`) as first-class stable heads.
4. **New bridge rules in the current snapshot.** The kernel now includes explicit Grothendieck morphism-action heads (`Fibration_cov_func`, `Fibration_cov_fapp1_func`) and direct total-hom bridges (`homd_curry`, `Homd_func`) used in computational rules; these bridges also expose the laxness pattern where cartesian source triangles can map to non-cartesian target triangles.

2. Kernel principles and surface reading

emdash is designed around:

1. **Internalization.** Categories, functors, and transformations are first-class terms, not meta-level structures.
2. **Normalization-first.** Many categorical equalities are presented as rewrite rules.
3. **Stable heads.** Composite expressions are folded to small “rewrite heads” so matching remains predictable.
4. **Variance by binders.** Functorial/contravariant/object-only dependencies are carried by binder notation in the intended surface language.

   We use the following surface-style conventions:

- `x : A` is a functorial index (e.g. `F : A → B` gives `x:A ⊢ F[x] : B`).
- `x :^- A` is contravariant, used for arrow-indexed components like `ϵ_(f)` (accumulation/composition laws orient correctly).
- `x :^o A` is object-only (generic displayed categories `E : Catd A` do not provide implicit transport along base arrows).

  Several kernel projections are intended to be *silent* at the surface: `τ`, `Fibre_cat`, `fapp0`, and diagonal components of transfors.

2.1 Kernel ↔ surface ↔ mathematics (overview table)

Kernel head	Surface reading (intended)	Standard meaning

`Cat`	`⊢ C : Cat`	category / ω-category classifier
`Obj : Cat → Grpd`	`⊢ x : C`	object groupoid of a category
`Hom_cat C x y`	`x :^- C, y : C ⊢ f : x → y`	hom-category (so 1-cells are its objects)
`Op_cat A`	`A^op` (surface)	opposite category (computes definitionally)
`Path_cat G`	`Path(G)` (informal)	category of paths in a groupoid
`Core_cat C`	`Core(C)` (informal)	groupoidal core from object paths
`Grpd_cat`	`⊢ Grpd : Cat` (internal)	the category of groupoids
`Cat_cat`	`⊢ Cat : Cat` (internal)	the category of categories
`Functor_cat A B`	`⊢ F : A → B`	functor category
`fapp0 F x`	`F[x]`	object action
`fapp1_fapp0 F f`	`F[f]` (silent)	arrow action (as a 1-cell)
`Transf_cat F G`	`⊢ ϵ : Transf(F,G)`	transformations / transfors
`tapp0_fapp0 Y ϵ`	`ϵ[Y]` (silent)	diagonal component
`tapp1_fapp0 … ϵ f`	`ϵ_(f)`	off-diagonal component “over `f` ”
`hom_`	`Hom(W,F[−])` (informal)	covariant Cat-valued representable
`hom_con`	`Hom(F[−],W)` (informal)	contravariant representable (via opposite)
`Catd Z`	`⊢ E : Catd Z`	displayed category / (iso)fibration over `Z`
`Fibre_cat E z`	`E[z]`	fibre category over `z`
`Functord_cat E D`	`⊢ FF : E →_Z D`	displayed functors over fixed base
`Fibration_cov_catd M`	(silent if `M: Z → Cat`)	Grothendieck construction `∫M`
`Fibration_cov_func`	(internal)	functor object packaging `∫ : (Z→Cat)→Catd(Z)`
`Fibration_cov_fapp1_func`	(internal)	Groth morphism-action bridge on transfors
`Total_cat E`	`∫E` (informal)	total category of a displayed category
`Total_func`	(internal)	internalized totalization `(Z→Cat)→Cat`

`homd_`	`Homd_E(w,-)` (informal)	dependent arrow/comma category (“triangle classifier”)
`homd_curry`, `Homd_func`	(internal)	direct curry/total-hom computational bridge
`homd_int_base`	(internal)	compositional “indexing” pipeline for `homd_`
`Transfd_cat FF GG`	`⊢ ϵ : Transfd(FF,GG)`	displayed transfors
`tdapp0_fapp0 … ϵ`	`ϵ[e]` (silent)	displayed component in a fibre
`tdapp1_*`	`ϵ_(σ)`	displayed off-diagonal component over $\sigma : e \to_f e'$
`adj`	(kernel type former)	adjunction data (unit/counit) and triangle reduction

The kernel also defines stable aliases for “objects of a category” (since `Obj` is not injective in this development): `Hom`, `Functor`, `Transf`, and their displayed analogues (`Fibre`, `Functord`, `Transfd`). We use these aliases in code snippets below.

Notation convention. In surface typing examples, we write `⊢ x : C` as shorthand for `⊢ x : τ (Obj C)` (and similarly `f : x → y` abbreviates `f : τ (Hom C x y)`).

2.2 Executable feasibility evidence (v1)

In parallel to the Lambdapi kernel, the project includes an earlier executable prototype (a bidirectional elaborator with holes and normalization-driven definitional equality). Its key methodological takeaway is: the system can *compute-check coherence* during elaboration by normalizing both sides of functoriality/coherence constraints and rejecting mismatches (reporting dedicated errors, rather than producing proof obligations).

In the v2 story, the stable-head discipline plays the analogous role: instead of proving a growing library of “naturality lemmas”, we design primitives and rewrite rules so the relevant equalities are available by conversion.

2.3 Internalized computational logic and surface elaboration

The kernel is engineered so that it can serve as an internal computational logic for a surface programming language / proof assistant. A reference implementation effort for such a surface layer (parsing + elaboration + rendering) lives at `https://github.com/hotdocx/emdash`.

The essential idea is that *elaboration is guided by binder modes* and by stable-head elimination operators, rather than by asking the user for separate naturality/coherence proof terms. For example, for displayed categories `E,D : Catd Z` a displayed functor is read schematically as:

$$z : Z,\ e : E[z] \vdash FF[e] : D[z],$$

and for a displayed transfor ϵ between displayed functors, diagonal components are silent while the off-diagonal component over a displayed arrow $\sigma : e \to_f e'$ is explicit:

$$\epsilon_{(\sigma)} : FF[e] \to_f GG[e'].$$

This reflects the kernel architecture: “internalization” operators (the `*_int_*` family) are treated as named discharge/abstraction principles whose computation is governed by rewrite rules, so that higher functoriality/naturality is available by normalization.

2.3 Contextual developments (evidence of scale)

This paper focuses on the v2 kernel, but two additional threads provide context. First, the executable prototype demonstrates feasibility of the “kernel spec → interactive assistant” pipeline (holes + elaboration + normalization). Second, earlier large-scale rewrite-centric warm-ups (universal properties/adjunction transposes; Grothendieck-style geometry interfaces) stress-test the approach and provide a backlog of computation laws to port into the v2 stable-head discipline.

3. Core type theory: `Grpd`, `Cat`, and homs-as-categories

The kernel separates:

- `Grpd : TYPE` — codes for (∞-)groupoids (paths/equality live here),
- `Cat : TYPE` — codes for (strict/lax) ω-categories.

Every category has an object classifier `Obj : Cat → Grpd`, so object equality is a *path in a groupoid*.

Equality itself is valued in `Grpd`, so the “equality type” of an object classifier is another groupoid (and can be iterated):

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

Instead of `Hom_C(x,y)` being a set/type, in emdash it is a category:

```
 injective symbol Hom_cat : Π (A : Cat) (X_A Y_A : τ (Obj A)), Cat;
```

Thus a “1-cell” $f : x \to y$ is an *object* of `Hom_cat C x y`. A “2-cell” between parallel 1-cells is then a 1-cell in that hom-category, etc.

Opposites (`Op_cat`) compute definitionally (objects unchanged; homs reversed), and emdash also introduces `Path_cat` and `Core_cat` to relate object paths to directed morphisms (in a controlled, one-way direction).

3.1 Paths as morphisms (one-way, by design)

The object groupoid `Obj C : Grpd` gives a path/equality structure on objects. To connect this to *directed* morphisms without collapsing directed structure into paths, emdash introduces:

- `Path_cat : Grpd → Cat`, the category of paths in a groupoid;
- `Core_cat C := Path_cat (Obj C)`, the “groupoidal core” of C induced by object paths;
- a *one-way* map “path ⇒ morphism”:

```
 constant symbol path_to_hom_func : Π [C : Cat], Π (x y : τ (Obj C)),
   τ (Functor (Path_cat (x = y)) (Hom_cat C x y));
symbol path_to_hom_fapp0 : Π [C : Cat], Π (x y : τ (Obj C)), Π (p : τ (x = y)),
   τ (Hom C x y);
```

This direction is safe for definitional computation (it does not create rewrite loops). The reverse direction (morphism ⇒ path) is the dangerous one and is treated as optional infrastructure (e.g. via carefully controlled `unif_rule` bridges) rather than as a primitive definitional equivalence.

Finally, emdash internalizes "the category of groupoids" and "the category of categories" as categories `Grpd_cat` and `Cat_cat`, so that constructions about "categories of categories" can be expressed using the same functorial machinery (e.g. `Hom_cat Cat_cat A B` computing to `Functor_cat A B`).

4. Functors and transfors (ordinary)

For categories $A, B : \mathsf{Cat}$, the category of functors is `Functor_cat A B : Cat`. We expose:

- object action `fapp0`,
- action on hom-categories via `fapp1_func`,
- a stable head for 1-cell action `fapp1_fapp0`.

Morphisms in a functor category are transfors. The kernel avoids a record encoding; instead it exposes projection heads for components:

```
symbol tapp0_fapp0 : Π [A B : Cat], Π [F_AB G_AB : τ (Functor A B)],
  Π (Y_A : τ (Obj A)), Π (ϵ : τ (Transf F_AB G_AB)),
  τ (Hom B (fapp0 F_AB Y_A) (fapp0 G_AB Y_A));
```

4.1 Stable heads and canonicalization (why this style?)

Lambdapi rewriting is powerful but fragile if rewrite rules must match against huge expanded terms. emdash therefore makes a deliberate kernel commitment:

- introduce *stable heads* like `fapp1_fapp0`, `tapp0_fapp0`, `tapp1_fapp0`, `fib_cov_tapp0_fapp0`;
- add canonicalization rules that fold larger redexes into these heads;
- state most computation laws directly on the stable heads.

This keeps normalization predictable (matching sees the head) and makes it realistic to orient coherence as cut-elimination steps.

Rewriting is complemented by unification guidance: emdash uses `rule` for normalization steps that should change terms, and `unif_rule` for inference hints that should guide conversion/elaboration without changing canonical normal forms. This distinction is part of the kernel architecture (not merely tooling).

4.2 Off-diagonal components: `tapp1_fapp0`

Instead of treating naturality as a proposition, emdash exposes an explicit arrow-indexed component

$\epsilon_{(f)} : F(X) \to G(Y)$

as a stable-head operation `tapp1_fapp0`. This is the computational handle for “lax naturality data”.

Figure 1: an off-diagonal component as a triangle

$$\begin{tikzcd}
F(X) \arrow[dr, "F(f)"'] \arrow[drr, bend left, "\epsilon_{(f)}"] & & \\
& F(Y) \arrow[r, "\epsilon_Y"'] & G(Y)
\end{tikzcd}$$

Figure 2: the composite case and “accumulation” over base arrows

$$\begin{tikzcd}
F(X) \arrow[dr, "F(f)"'] \arrow[drr, bend left, "\epsilon_{(g\circ f)}"] & & \\
& F(Y) \arrow[r, "\epsilon_{(g)}"'] & G(Z)
\end{tikzcd}$$

Kernel-wise, this is where the contravariant “accumulation” discipline pays off: one can orient coherence as

$$(\epsilon_{(g)}) \circ F(f) \ \rightsquigarrow\ \epsilon_{(g\circ f)},$$

so normalization *accumulates* the base-arrow index.

Equivalently (and closer to a 2-categorical reading), accumulation can be presented as two rewrite orientations on off-diagonal components:

$$(G(b)) \cdot \epsilon_{(a)} \ \rightsquigarrow\ \epsilon_{(b\cdot a)}, \qquad \epsilon_{(b)} \cdot (F(a)) \ \rightsquigarrow\ \epsilon_{(b\cdot a)}.$$

An instance is the **exchange law** for pasting (vertical cut $\cdot$ and horizontal composition $\circ$):

$$(g\circ\beta) \cdot (e\circ\alpha) \ \rightsquigarrow\ e\circ(\beta\cdot\alpha),$$

so that normalization yields a canonical pasted 2-cell.

For reference, the stable head has the following kernel type (here `@` just disables implicit arguments):

```
symbol tapp1_fapp0 : Π [A B : Cat], Π [F_AB G_AB : τ (Functor A B)],
 Π [X_A Y_A : τ (Obj A)],
 Π (ϵ : τ (Transf F_AB G_AB)),
 Π (f : τ (Hom A X_A Y_A)),
 τ (Hom B (fapp0 F_AB X_A) (fapp0 G_AB Y_A));
```

4.3 Diagonal components as evaluation-at-identity (intuition)

Conceptually, the diagonal component $\epsilon_Y : F(Y) \to G(Y)$ is "the off-diagonal component over the identity edge". In the kernel, `tapp0_fapp0` is implemented by evaluating a packaged arrow-indexed construction at `id_Y`. The stable head `tapp0_fapp0` is retained so later rewrite rules do not need to unfold that packaging.

4.4 Representables: `hom_` / `hom_con` (Yoneda-style heads)

emdash also provides Cat-valued representables:

- `hom_` models $\mathrm{Hom}_A(W, F(-))$ (covariant),
- `hom_con` models $\mathrm{Hom}_A(F(-), W)$ (contravariant, by reduction to `hom_` in the opposite category).

These heads are not only for "doing Yoneda"; they are also glue in the internal packaging of transfors and dependent homs, where "naturality" is recorded as postcomposition behavior and exposed as normalization on stable heads.

4.5 Strictness as optional structure: `StrictFunctor_cat` (brief)

While the long-term goal includes lax/weak structure, the kernel often begins with strict computation rules and relaxes them later. In particular, `StrictFunctor_cat A B` classifies functors equipped with definitional computation laws expressing preservation of identities and composition *on the nose* at the 1-cell level, and is related to ordinary functors via `sfunc_func`. This provides a pragmatic "strict core" that can later be embedded into the simplicial/lax story.

5. Dependent categories (`Catd`) and Grothendieck constructions

The kernel has a classifier `Catd Z` of dependent categories over a base Z (intended: displayed categories / isofibrations).

For a Cat-valued functor $M : Z \to \mathbf{Cat}$, emdash provides a displayed category `Fibration_cov_catd M : Catd Z` (Grothendieck construction). In this special case fibres and several structural operations compute definitionally.

Two additional heads matter for compositionality:

- `Total_cat E` packages the total category $\int E$ of a displayed category,
- `Total_func` internalizes totalization as an object in `Functor_cat (Functor_cat Z Cat_cat) Cat_cat`, so it can be composed inside `Cat_cat` without unfolding large definitions.

- `Fibration_cov_func` / `Fibration_cov_fapp1_func` package object- and morphism-action of Grothendieck construction as stable heads.

In the current kernel snapshot, the object layer of totals is now computational for arbitrary displayed categories:

$$\tau\big(\mathrm{Obj}(\mathrm{Total\_cat}(E))\big) \rightsquigarrow \sum_{z\in\mathrm{Ob}(Z)} \mathrm{Obj}(E[z]).$$

In the Grothendieck case this specializes to the expected

$$\mathrm{Ob}(\textstyle\int M) \simeq \sum_{z\in\mathrm{Ob}(Z)} \mathrm{Ob}(M(z)).$$

Hom-level computation for general `Total_cat E` remains mostly abstract, with dedicated computational bridges in Grothendieck-shaped cases.

Concretely (Grothendieck case), the kernel includes:

```
 symbol Total_func [Z : Cat] : τ (Functor (Functor_cat Z Cat_cat) Cat_cat);
rule @fapp0 _ _ (@Total_func $Z) $M ↪ @Total_cat $Z (@Fibration_cov_catd $Z $M);
```

5.1 Fibrewise products: `Product_catd` and `prodFib` (why it appears in `homd_int_base`)

Displayed categories over a fixed base admit a fibrewise product `Product_catd U A : Catd Z`, whose fibre over z is (informally) the product category $U[z] \times A[z]$. For Grothendieck displayed categories $\int E$ and $\int D$, emdash also introduces a stable-head `prodFib` for pointwise product of Cat-valued functors, so that composite constructions can match on a small head rather than on a large expanded product/total expression.

Figure 3: Grothendieck morphisms lie over base arrows

$$\begin{tikzcd}
b \arrow[r, "f"] & b' \\
(b,e) \arrow[u, dashed, "p"'] \arrow[r, "{(f,\sigma)}"] & (b',x) \arrow[u, dashed, "p"]
\end{tikzcd}$$

emdash also exposes a stable head for (strict, placeholder) Grothendieck transport on fibre objects, so nested transports fold to one transport along a composite base arrow.

Concretely, transport on fibre objects is strict today (as an engineering placeholder for a later lax story). Informally:

- $(\mathrm{id})_!(u) \rightsquigarrow u$,
- $g_!(f_!(u)) \rightsquigarrow (g\circ f)_!(u)$.

6. Dependent arrow/comma categories: `homd_` and `homd_int`

Let Z be a category, E a dependent category over Z (morally $E : Z \to \mathbf{Cat}$), and fix a base object $W \in Z$ and a fibre object $w \in E(W)$. Given a base arrow $f : W \to z$ and a fibre object $x \in E(z)$, we want the category of "arrows from the transported probe to the target":

$$\mathrm{Hom}_{E(z)}(f_! w, x).$$

This is the basic "triangle classifier": *a 2-cell is a 1-cell in a dependent arrow category*.

This "dependent arrow/comma" construction is also reminiscent of directed "bridge type" constructions in internal parametricity: it internalizes a relation indexed by a base arrow without introducing an external interval object.

6.1 Input shape and "over a base edge"

In the Grothendieck case, the value of `homd_` at a point is indexed by:

- a base target object $z \in Z$,
- a displayed target object $d \in D(z)$ (in the probe family),
- and a base edge $f : W \to z$ (the edge along which we transport the probe object $w \in E(W)$).

In the current kernel snapshot, such points are represented in a canonical Σ-pair normal form `Struct_sigma z (Struct_sigma d f)`. This "syntactic normal form" is essential: it makes it possible for the computation rule of `homd_` to match and reduce without requiring additional definitional unfolding.

In the Grothendieck–Grothendieck case, the kernel contains a pointwise computation rule for `homd_`:

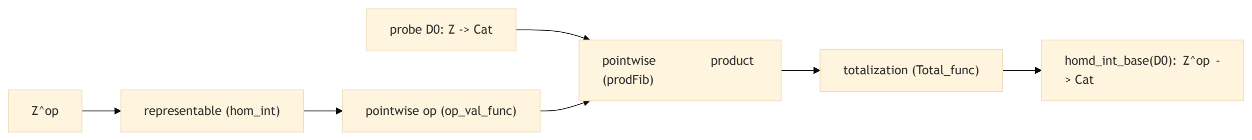
```
rule fapp0 (@homd_ $Z
              (@Fibration_cov_catd $Z $E0)
              (@Fibration_cov_catd $Z $D0)
              $FF
              $W_Z $W)
            (Struct_sigma $z (Struct_sigma $d $f))
  ↪ Hom_cat (fapp0 $E0 $z)
      (fapp0 (fapp1_fapp0 $E0 $f) $W)
      (@fdapp0 _ _ _ $FF $z $d);
```

Figure 4: a displayed arrow over a base arrow

$$\begin{tikzcd}
e \in E(b) \arrow[r, bend left, "\sigma : f_! e \to x"] \arrow[d, dashed, "p"'] & x \in E(b') \arrow[d, dashed, "p"] \\
b \arrow[r, "f"] & b'
\end{tikzcd}$$

For compositionality, the kernel also contains a “more internal” pipeline `homd_int_base` built from stable-head building blocks (opposite, fibrewise products, totalization). It is designed so later constructions can range over base arrows explicitly without rewriting blowups.

In parallel, the current kernel includes a direct `homd_curry` / `Homd_func` path used as a computational bridge for homs of total categories in Grothendieck-shaped cases.


probe D0: Z -> Cat
Z^op
representable (hom_int)
pointwise op (op_val_func)
pointwise product (prodFib)
totalization (Total_func)
homd_int_base(D0): Z^op -> Cat

7. Displayed transfors and simplicial iteration (sketch)

In addition to ordinary transfors, the kernel includes displayed transfors between displayed functors over a fixed base. As with ordinary transfors, the interface is via projection heads (pointwise components `tdapp0_*` and off-diagonal components `tdapp1_*`) rather than via a record encoding.

The pointwise displayed component head has the shape:

```
symbol tdapp0_fapp0 : Π [Z : Cat], Π [E D : Catd Z],
 Π [FF GG : τ (Functord E D)],
 Π (Y_Z : τ (Obj Z)),
 Π (V : τ (Fibre E Y_Z)),
 Π (ϵ : τ (Transfd FF GG)),
 τ (Hom (Fibre_cat D Y_Z) (fdapp0 FF Y_Z V) (fdapp0 GG Y_Z V));
```

As with ordinary transfors, the kernel also provides **off-diagonal** displayed components over displayed arrows (stable heads `tdapp1_*`), intended to be the home for higher “lax naturality” data in the displayed setting (surface syntax `ϵ_(σ)` for $\sigma : e \to_f e'$).

The intended geometric reading is simplicial: triangles over base edges, and higher simplices obtained by iterating “dependent hom” layers. The current kernel snapshot contains the beginning of this machinery (enough to state and normalize many pointwise computations in the Grothendieck case), together with phase-2 draft strict naturality/exchange rewrite rules on `tapp1_fapp0` and a representable exchange-law sanity assertion; general iteration remains an interface to be refined.

In compact schematic form (matching the `fapp1_funcd` / `fdapp1_funcd` comments), a lax action

$$F_1 : \mathrm{Hom}_C(x,-) \to \mathrm{Hom}_D(F_0x, F-)$$

induces

$$((F_1)_1)_0 : \mathrm{Homd}_{\mathrm{Hom}_C(x,-)}\big(f,(g\circ f,g)\big) \to \mathrm{Homd}_{\mathrm{Hom}_D(F_0x,F-)}\big((F_1)_0f,\big((F_1)_0(g\circ f),(F_1)_0g\big)\big),$$

and the image of an identity-like/cartesian source triangle may be non-identity/non-cartesian in the target. That non-identity image is the laxness witness.

Figure 5: a 2-cell between parallel composites (Arrowgram arrow-to-arrow)

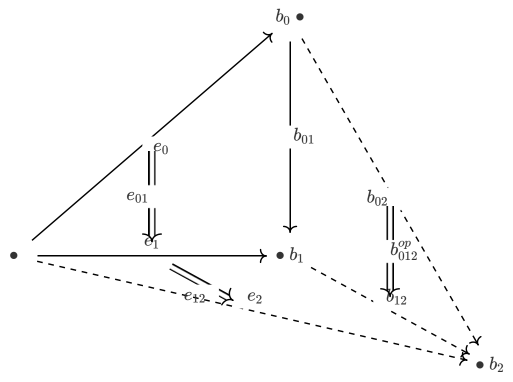
$$\begin{tikzcd}
X \arrow[rr, "f"] \arrow[ddr, bend right, "h"', ""{name=H}] \arrow[ddr, bend left, "g\circ f", ""{name=GF, swap}] & & Y \arrow[ddl, "g"] \\
& & \\
& Z &
\arrow[Rightarrow, from=GF, to=H, "\alpha"]
\end{tikzcd}$$

Figure 6: stacking 2-cells along a 1-cell (a tetrahedral “over-a-base-edge” picture)

Recall from §6 that the dependent arrow/comma classifier organizes “2-cells over a chosen base edge” via a functor

$$\mathrm{Homd}_E(e_0,--) : E \times_B \big(\mathrm{Hom}_B(b_0,-)\big)^{\mathrm{op}} \to \mathbf{Cat}.$$

Stacking corresponds to composing such base edges and asking for a computational interface where normalization re-associates and “exchanges” these simplicial indices.



8. Computational adjunctions (cut-elimination rule)

The kernel contains a draft interface for adjunctions where unit and counit are first-class 2-cell data and a triangle identity is oriented as a rewrite rule. Very roughly:

$$\epsilon_f \circ L(\eta_g) \;\rightsquigarrow\; f \circ L(g).$$

This is implemented as a rewrite rule at the level of stable heads (`comp_fapp0`, `fapp1_fapp0`, `tapp1_fapp0`). The key point is that normalization of a composite term performs the triangle reduction. In the current snapshot this remains draft-level: the rewrite is present and typechecks, while a fully closed regression term is still marked TODO in the kernel comments.

```
rule comp_fapp0
     (@tapp1_fapp0 _ _ _ _
       (fapp0 (@LeftAdj $R $L _ _ _ _ $a) $X')
       $Y
       (@CoUnitAdj $R $L _ _ _ _ $a)
       $f)
     (fapp1_fapp0 (@LeftAdj $R $L _ _ _ _ $a)
       (@tapp1_fapp0 _ _ _ _
         $X
         $X'
         (@UnitAdj $R $L _ _ _ _ $a)
         $g))
 ↪ comp_fapp0
     $f
     (fapp1_fapp0 (@LeftAdj $R $L _ _ _ _ $a) $g);
```

Figure 7: the familiar triangle identity (as reduction)

$$\begin{tikzcd}
& L(R(L(A))) \arrow[dr, "\epsilon_{L(A)}"] & \\
L(A) \arrow[ur, "L(\eta_A)"] \arrow[rr, dashed, "\mathrm{id}"'] & & L(A)
\end{tikzcd}$$

The point is methodological: coherence can be enforced by computation (normalization on stable heads) rather than by building a separate library of propositional equalities.

9. Displayed functors and limits of computation

9.1 Displayed functors: slice-style and pullback

There are (at least) two common ways to organize displayed functors:

1. **General base map.** A displayed functor may live over an arbitrary base functor $F : X \to Y$.
2. **Slice-style.** Fix a base Z, and consider only functors over id_Z between displayed categories over Z.

The kernel uses the slice-style presentation because it makes composition and normalization stable:

- displayed categories over Z are terms of `Catd Z`,
- displayed functors over Z are objects of `Functord_cat E D`,
- composition stays over the same base automatically.

The "general base map" viewpoint is recovered by pullback: a functor over F can be represented as an ordinary slice-style functor into a pullback `Pullback_catd D F`.

9.2 Limitations (what the current kernel does *not* compute)

This paper emphasizes computations that are already stable in the kernel. In particular:

- **Displayed categories are only partially computational.** Generic `Catd` now has computational object-level structure in several places (e.g. fibres, pullback/opposite/terminal rules, and `τ (Obj (Total_cat E))`), but hom-level behavior remains mostly computational only in Grothendieck-shaped cases.
- **Strictness placeholders exist.** Some computations (e.g. Grothendieck transport on fibre objects) are strict today as a normalization placeholder; the long-term goal is a lax/weak story where functoriality/transport come with higher cells rather than definitional equalities.
- **Full simplicial iteration is still an interface.** `homd_` computes pointwise in the Grothendieck–Grothendieck case, `homd_curry` / `Homd_func` provide direct bridges for total homs in Groth-shaped cases, and `homd_int_base` provides compositional indexing, but uniform exchange/stacking normalization, full `homd_int` computation, and explicit rewrite-level cartesian-vs-non-cartesian triangle infrastructure remain ongoing work.

- **Adjunction layer remains draft-level.** A first triangle cut-elimination rewrite exists, but surrounding bridge infrastructure and closed regression terms are still incomplete.

References

1. F. Blanqui et al. *The Lambdapi Logical Framework*.
2. K. Došen and Z. Petrić. *Cut-Elimination in Categories*.
3. E. Finster and S. Mimram. *A Type-Theoretical Definition of Weak ω-Categories*.
4. T. Altenkirch, Y. Chamoun, A. Kaposi, M. Shulman. *Internal Parametricity, without an Interval*.
5. H. Herbelin, R. Ramachandra. *Parametricity-based formalizations of semi-simplicial / semi-cubical structures*.

Appendix A. Reading guide (kernel identifiers → math)

- `Cat` , `Obj` , `Hom_cat` : category classifier, object groupoid, hom-category (iterated for higher cells).
- `Functor_cat` , `fapp0` , `fapp1_fapp0` : functors as objects; object and (1-cell) arrow action.
- `Transf_cat` , `tapp0_fapp0` , `tapp1_fapp0` : transfors; diagonal and arrow-indexed components.
- `Catd` , `Fibre_cat` , `Functord_cat` : displayed categories, fibres, displayed functors over a fixed base.
- `Fibration_cov_catd` , `Fibration_cov_func` , `Fibration_cov_fapp1_func` : Grothendieck construction and its object-/morphism-action bridge.
- `homd_` , `homd_curry` , `Homd_func` : dependent arrow/comma constructions and direct total-hom computational bridges.
- `homd_int_base` , `homd_int` : internalized indexing pipeline and displayed packaging (currently mostly interface-level).