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

m— / emdash project

Abstract

We report on an ongoing experiment, `emdash2.lp`, whose goal is a new *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 implementation target is the Lambdapi logical framework, and the guiding methodological stance is proof-theoretic: many categorical equalities are best presented as *normalization* ("cut-elimination") steps rather than as external propositions.

The central construction is a dependent comma/arrow ("dependent hom") operation that directly organizes "cells over a base arrow" in a simplicial manner. Concretely, let B be a category and let E be a dependent category over B (informally a fibration $E : B \to \mathbf{Cat}$). Fix a base object $b_0 \in B$ and a fibre object $e_0 \in E(b_0)$. We construct a Cat-valued functor that assigns to a base arrow $b_{01} : b_0 \to b_1$ and a fibre object $e_1 \in E(b_1)$ the category of morphisms in the fibre over b_1 from the transport of e_0 along b_{01} to e_1. In slogan form, this is a dependent arrow/comma object

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

Iterating this construction yields a simplicial presentation of higher cells (triangles, surfaces, higher simplices), where "stacking" of 2-cells along a 1-cell is expressed *over a chosen base edge*.

As a complementary application, we outline a computational formulation of adjunctions in which unit and counit are first-class 2-cell data and the triangle identities are oriented as definitional reductions on composites (e.g. $\varepsilon_f \circ L(\eta_g) \rightsquigarrow f \circ L(g)$). This showcases the broader emdash theme: coherence is enforced by computation, via stable rewrite heads for functoriality and "off-diagonal" components of transformations.

1. Introduction (what problem are we solving?)

Higher category theory is hard to formalize for two intertwined reasons:

1. *Size of coherence*: weak n-categories and especially weak ω-categories come with an explosion of coherence data.
2. *Where equalities live*: in proof assistants, equalities are propositions to be proved, while in category theory many equalities are "structural" and ought to compute away.

The emdash project explores a kernel design in which:

- "Category theory" is internal: we work *inside* a dependent type theory that has a classifier `Cat` of categories and a classifier `Grpd` of (∞-)groupoids.
- Many categorical laws are definitional: they are enforced by rewrite rules and unification hints, so that normalization *is* diagram chasing.

This paper is a narrative tour of the current `emdash2.lp` kernel. It aims to be readable to a non-specialist, while staying faithful to the code. When we mention a kernel identifier (e.g. `homd_cov_int_base`) we also provide a traditional reading.

1.1 Contributions (what is new here?)

This paper's contributions are primarily expository (the kernel is ongoing work), but the kernel already embodies several concrete design/engineering commitments:

1. **A rewrite-head discipline for categorical computation.** Most "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 for higher cells over base arrows.** The dependent arrow/comma construction `homd_cov` (and the internal pipeline `homd_cov_int_base`) provides a computational home for "triangles/surfaces" in the Grothendieck case, and a compositional target for iteration.
3. **An explicit off-diagonal interface for transfors (ordinary and displayed).** Instead of encoding transfors as records with a naturality law, we expose diagonal components (`tapp0_*`, `tdapp0_*`) and off-diagonal components over arrows (`tapp1_*`, `tdapp1_*`) as first-class stable heads.
4. **Executable feasibility evidence.** The v1 executable kernel (reported in `emdash_cpp2026.md`) demonstrates that this "kernel spec → elaborating proof assistant" pipeline is realistic; the v2 paper build includes automated rendering/console checks for reproducible artifacts.
5. **Continuity with prior Lambdapi warm-ups.** Earlier Lambdapi developments (`cartierSolution14.lp.txt`, `cartierSolution16.lp.txt`) already validate the *style* of emdash: large categorical interfaces presented with computational rules (products/exponentials/adjunction transposes; sieves/sites/sheafification; scheme interfaces). A key claim of the v2 design is that these developments are now largely portable into the `emdash2.lp` kernel discipline, and therefore count as prior progress rather than speculative future work.

2. Technical Overview and Design Principles

emdash is designed around a small set of kernel principles:

1. **Internalization.** Categories, functors, and (higher) transformations are first-class terms, not meta-level structures.
2. **Normalization-first.** Many categorical equalities are oriented as rewrite rules on canonical head symbols.
3. **Stable heads.** Large composite expressions are folded to small "rewrite heads" so normalization is predictable and performant.
4. **Variance by binders.** Functorial/contravariant/object-only dependencies are carried by binder notation at the surface level (see `docs/SYNTAX_SURFACE.md`).

Lambdapi is a natural implementation target because it provides user-defined rewrite rules and higher-order unification. This matters because emdash wants two things simultaneously:

- **Internal definitions**: a functor is not a meta-level function; it is an *object* of a functor category.
- **Computation**: categorical equalities are implemented as *rewrites on stable head symbols*, so they are available during typechecking.

In standard proof assistants you typically have:

- definitional equality: β/δ/ι/ζ reductions;
- propositional equality: theorems you rewrite with.

In emdash we deliberately push a large part of the "categorical equational theory" into definitional equality, using Lambdapi's rewriting and (carefully!) its unification rules.

2.1 Background: emdash v1 (TypeScript kernel) and "functorial elaboration"

Before `emdash2.lp`, the project developed a 1-categorical prototype (`emdash.lp`) together with an executable kernel in TypeScript (a bidirectional elaborator, unification-based hole solving, and normalization). The report `emdash_cpp2026.md` documents this first version as a *real* programming language / proof assistant, not only as a Lambdapi specification.

Two takeaways from v1 matter for the v2 story:

1. **Feasibility of an implementation pipeline.** The kernel concepts in the Lambdapi spec can be implemented as an interactive system with holes, bidirectional checking, and a normalization-driven definitional equality. This strongly suggests that `emdash2.lp` can likewise be turned into a usable assistant once the surface syntax and elaboration layer is designed.
2. **Coherence as computation ("functorial elaboration").** In the v1 TypeScript kernel, the constructor of a structured object (e.g. a functor) can *compute-check* its laws during elaboration: the system normalizes both sides of functoriality equations in a generic context, and throws a dedicated `CoherenceError` if they do not match definitionally. This is the same philosophical stance as v2's rewrite-head style: coherence is enforced by computation, not by a separately managed library of lemmas.

The present paper focuses on the v2 Lambdapi kernel itself, but we treat the v1 implementation as evidence that the "specification-first → executable kernel" path is realistic.

2.1.1 Warm-up Lambdapi developments: from arithmetic to schemes

In parallel to the v1 executable kernel, the project produced two "warm-up" Lambdapi developments that are worth crediting explicitly because they already instantiate the intended methodological stance.

(i) `cartierSolution14.lp.txt`: computation by universal properties. This file develops (in a more ad hoc kernel than v2) internal products/exponentials/adjunction machinery and uses it to compute with familiar constructions. A guiding demo is that even a toy statement like "$1 + 2 = 3$" can be realized *three ways*—via an intrinsic datatype of naturals, via a natural numbers object presented by internal adjunctions, and via finite sets with colimits—so that the result is obtained by normalization of the categorical interface rather than by external equational reasoning. The point is not the arithmetic itself but the principle: "universal property = computation rule".

(ii) `cartierSolution16.lp.txt`: a computational interface for sieves, sheaves, and schemes. This file specifies a large amount of categorical infrastructure for Grothendieck-style geometry (sieves, sites, closure/sheafification operations, subobjects, and a schematic interface for ringed sites and localizations), again in the proof-theoretic spirit that core equalities should compute. For example, pullback stability of sieve classifiers and basic glue/closure interactions are recorded as rewrite rules so that a "sheafiness" interface is operational.

These warm-ups are not presented in this paper as *the* final kernel: they rely on earlier encodings (categories/functors/profunctors as primitive records) and therefore do not match the v2 stable-head discipline. Their significance is different: they show that the emdash approach scales to large libraries, and they provide a backlog of definitions and computational laws that can be re-expressed inside `emdash2.lp` (with `Cat`, `Catd`, and the transfor heads) rather than re-invented.

2.2 How to read kernel vs surface syntax

`emdash2.lp` is "kernel-first": it chooses canonical internal heads and rewrite rules. The intended *surface language* (the syntax a user would actually type) is described separately in `docs/SYNTAX_SURFACE.md`. The crucial idea is that **variance is tracked by binder notation**, not by explicit "naturality proof terms".

We will use the following surface-style conventions (matching `docs/SYNTAX_SURFACE.md`):

- **Functorial index** `x : A`: a variable intended to vary functorially (e.g. when you have a functor $F : A \to B$, you write $x : A \vdash F[x] : B$).
- **Contravariant index** `x :^- A`: used for arrow-indexed components like $\epsilon_{(f)}$ (it makes "accumulation" rules orient correctly).
- **Object-only index** `x :^o A`: the default for a generic displayed category $E : \mathrm{Catd}(A)$; you may substitute along *paths* in $\mathrm{Obj}(A)$, but you do not assume transport along base arrows unless the structure provides it.

Several kernel projections are intended to be *silent* in surface syntax:

- `τ` (decoding a `Grpd`-code to a type),
- `F[x]` for `fapp0 F x`,
- `E[x]` for `Fibre_cat E x`,
- diagonal components `ϵ[x]` / `ϵ[e]` for transfors and displayed transfors.

The explicit, computationally important data is typically **off-diagonal**: $\epsilon_{(f)}$ (ordinary) and $\epsilon_{(\sigma)}$ (displayed) for "over-a-base-arrow" components. Those correspond to stable heads like `tapp1_fapp0` and `tdapp1_*`.

2.3 Technical overview (kernel ↔ surface ↔ mathematics)

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)
`Functor_cat A B`	`⊢ F : A → B`	functor category
`fapp0 F x`	`F[x]`	object action F_0
`fapp1_fapp0 F f`	`F[f]` (silent)	arrow action $F_1(f)$ (as a 1-cell)
`Transf_cat F G`	`⊢ ϵ : Transf(F,G)`	transformations / transfors
`tapp0_fapp0 Y ϵ`	`ϵ[Y]` (silent)	component ϵ_Y
`tapp1_fapp0 … ϵ f`	`ϵ_(f)`	off-diagonal component "over f"
`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
`fdapp0 … FF z e`	`FF[e]`	fibrewise object action of displayed functor
`Fibration_cov_catd M`	(silent if `M: Z → Cat`)	Grothendieck construction $\int M$
`Total_cat E`	`∫E` (informal)	total category of a displayed category
`fib_cov_tapp0_fapp0 M f u`	$f_!(u)$	Grothendieck transport on fibre objects (strict today)
`homd_cov`	`Homd_E(w,–)` (informal)	dependent arrow/comma category (triangle classifier)
`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 $\sigma : e \to_f e'$

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 (Obj (Hom_cat C x y))`).

3. Core Type Theory: `Grpd`, `Cat`, and homs-as-categories

3.1 Two classifiers: groupoids vs directed structure

The kernel separates:

- `Grpd : TYPE` — codes for (∞-)groupoids (types with paths);
- `Cat : TYPE` — codes for (strict/lax) ω-categories.

Equality is valued in `Grpd` (so "paths form a groupoid"):

```
 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);
```

3.2 Objects are groupoidal

Every category has an object classifier:

```
symbol Obj : Cat → Grpd;
```

This is an explicit design choice: object "equality" in a category is *not* a primitive directed notion; it is a path in the object groupoid.

3.3 Hom is recursive: homs are categories

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;
```

So 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. This is the standard globular iteration, but the rest of the paper emphasizes a *simplicial* reorganization of this data.

3.4 Opposites and pointwise composition

The kernel treats the opposite category as a primitive constructor `Op_cat : Cat → Cat` with definitional computation rules:

- objects are unchanged: `Obj (Op_cat A) ↪ Obj A`;
- homs are reversed: `Hom_cat (Op_cat A) X Y ↪ Hom_cat A Y X`;
- identities and composition compute by "forgetting" `Op_cat` and reversing the order.

This is a small example of a general engineering pattern in emdash: if an operation is *structural* (opposite, projections, products), we try to make it compute definitionally so that later constructions can rely on normalization rather than on lemmas.

Similarly, composition in a category A is exposed in two complementary ways:

- a pointwise constructor `comp_fapp0` for composing 1-cells $g : y \to z$ and $f : x \to y$;
- an internal packaging `comp_func` (not shown here) that treats composition itself as a functor out of a product of hom-categories, so that it can carry higher structure when needed.

The pointwise head is what most rewrite rules target: it is the stable normal form for "$g \circ f$".

3.5 Paths as morphisms: `Path_cat` and `Core_cat`

The universe `Obj C : Grpd` means objects come with a path/equality structure. To relate that path structure to actual *directed* morphisms, emdash introduces:

- `Path_cat : Grpd → Cat`, the category whose objects are elements of a groupoid and whose morphisms are paths;
- `Core_cat C := Path_cat (Obj C)`, the "core" (groupoidal) category of C built from object paths.

This provides a clean place to talk about univalence later: one can add bridges between paths in `Obj C` and equivalences/isomorphisms in C without collapsing the whole directed structure into paths.

Concretely, `path_to_hom_func` and its stable-head application `path_to_hom_fapp0` give the direction "path ⇒ morphism":

```
constant symbol path_to_hom_func : Π [C : Cat], Π (x y : τ (Obj C)), τ (Obj (Functor_cat
(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 (Obj
(Hom_cat C x y));
```

This is intentionally one-way at the definitional level: the reverse direction (morphisms ⇒ paths) is the dangerous one that can create rewrite loops unless handled by a carefully controlled unification rule (as in the draft univalence bridge in `emdash2.lp`).

3.6 The "universe categories" `Grpd_cat` and `Cat_cat`

emdash internalizes "the category of groupoids" and "the category of categories" as actual categories:

- `Grpd_cat : Cat` with `τ (Obj Grpd_cat) ≡ Grpd`,
- `Cat_cat : Cat` with `τ (Obj Cat_cat) ≡ Cat`,
- and `Hom_cat Cat_cat A B` computing to `Functor_cat A B`.

This lets us write constructions like "postcompose by opposite" or "compose functors" as ordinary morphisms in `Cat_cat`, and then reuse the same functorial action machinery (`fapp0`, `fapp1_func`) uniformly.

4. Functors and Transfors (ordinary)

4.1 Functors are objects of a functor category

For categories A, B : `Cat`, the category of functors is `Functor_cat A B : Cat`.

```
constant symbol Functor_cat : Π (A B : Cat), Cat;
symbol fapp0 : Π [A B : Cat], Π (F : τ (Obj (Functor_cat A B))), τ (Obj A) → τ (Obj B);
symbol fapp1_func : Π [A B : Cat], Π (F : τ (Obj (Functor_cat A B))), Π [X Y : τ (Obj A)],
τ (Obj (Functor_cat (Hom_cat A X Y) (Hom_cat B (fapp0 F X) (fapp0 F Y))));
```

Here `fapp0` is the object action F_0, and `fapp1_func` is the induced functor on hom-categories F_1 (so it already "knows" about higher cells).

4.2 Stability by "rewrite heads"

In `emdash2.lp`, many operations are *declared* as symbols (not definitional abbreviations) so that rewrite rules can canonically fold large expressions into small stable heads. For example, applying `fapp1_func` and then `fapp0` is folded into a stable head `fapp1_fapp0`:

```
symbol fapp1_fapp0 : Π [A B : Cat], Π (F : τ (Obj (Functor_cat A B))), Π [X Y : τ (Obj A)],
Π (f : τ (Obj (Hom_cat A X Y))), τ (Obj (Hom_cat B (fapp0 F X) (fapp0 F Y)));
```

This "stable head discipline" is crucial for performance and for avoiding brittle matching against huge expanded terms.

4.3 Transfors (transformations) and components

Morphisms in a functor category are transfors (natural transformations / higher analogues). In emdash they live in a category `Transf_cat F G`. A key operation is component extraction: given $\epsilon : F \Rightarrow G$ and $Y \in A$, we want a 1-cell

$$\epsilon_Y : \mathrm{Hom}_B(FY, GY).$$

In the kernel this is a *primitive head* `tapp0_fapp0` with rewrite rules connecting it to the internal packaging:

```
symbol tapp0_fapp0 : Π [A B : Cat], Π [F G : τ (Obj (Functor_cat A B))], Π (Y : τ (Obj A)),
Π (ϵ : τ (Obj (Transf_cat F G))), τ (Obj (Hom_cat B (fapp0 F Y) (fapp0 G Y)));
```

The philosophy is: *do not bake "components" into a record definition of transfors*. Instead, compute components by normalization of projection heads.

4.4 Off-diagonal components: `tapp1_fapp0` and "lax naturality data"

In ordinary category theory, a natural transformation $\epsilon : F \Rightarrow G$ is often presented by its *diagonal* components $\epsilon_X : F(X) \to G(X)$, plus a *naturality equation* for every $f : X \to Y$:

$$G(f) \circ \epsilon_X = \epsilon_Y \circ F(f).$$

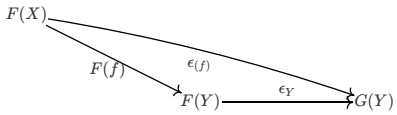
emdash aims at a more ω-friendly view: the "naturality square" is not necessarily an equality; it can carry higher cell data. Operationally, the kernel therefore exposes an explicit **off-diagonal / arrow-indexed component**

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

as a stable-head operation `tapp1_fapp0`. The intended surface syntax is `ϵ_(f)` and it uses the contravariant binder discipline `x :^- A` described in `docs/SYNTAX_SURFACE.md`.

Informally, `tapp1_fapp0` is a way to *name* "the part of a transfor that lives over the base arrow f". This is exactly the kind of interface we need for cut-elimination style rewrites: we can orient coherence laws as reductions on expressions built from `comp_fapp0`, `fapp1_fapp0`, and `tapp1_fapp0`, without committing to any record encoding of transfors.

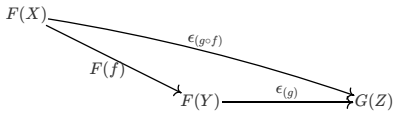
Figure 1: an off-diagonal component as a triangle



In a strict 1-category, you may take $\epsilon_{(f)} := \epsilon_Y \circ F(f)$ (or equivalently $G(f) \circ \epsilon_X$) and prove the two definitions equal. In emdash, we keep $\epsilon_{(f)}$ as explicit data because it is the correct home for higher "laxness" witnesses.

Figure 2: the composite case and "accumulation" over base arrows

The same idea becomes more informative when we look at a composite base arrow. If we have $f : X \to Y$ and $g : Y \to Z$, then an "off-diagonal" component can be presented directly over $g \circ f$:



Kernel-wise, this is exactly where the "contravariant binder / accumulation" discipline pays off: one can orient a coherence law as a rewrite of the form

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

so that normalization *accumulates* the base-arrow index instead of repeatedly expanding/contracting naturality squares.

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 (Obj (Functor_cat A B))],
 Π (X_A Y_A : τ (Obj A)),
 Π (ϵ : τ (Obj (Transf_cat F_AB G_AB))),
 Π (f : τ (Obj (@Hom_cat A X_A Y_A))),
 τ (Obj (@Hom_cat B (fapp0 F_AB X_A) (fapp0 G_AB Y_A)));
```

4.5 Representables and the Yoneda-style heads `hom_cov` / `hom_con`

emdash provides covariant and contravariant representables in a Cat-valued setting:

- `hom_cov` models $\mathrm{Hom}_A(W, F(-))$ (covariant in the variable, with postcomposition behavior);
- `hom_con` models $\mathrm{Hom}_A(F(-), W)$, implemented by a definitional reduction to `hom_cov` in the opposite category.

These are not just for "doing Yoneda"; they are also the glue behind the internal packaging of `tapp*` and `homd_cov_int_base`. A recurring kernel tactic is: *encode a textbook naturality statement as a rewrite rule about postcomposition by functors*, so that "naturality" becomes normalization.

4.6 Strictness as optional structure: `StrictFunctor_cat`

While the global goal includes lax/weak structure, the kernel often begins with strict computation rules and relaxes them later. For example, `StrictFunctor_cat A B` classifies functors equipped with rewrite rules stating that identities and composition are preserved *on the nose* at the level of 1-cells:

```
constant symbol StrictFunctor_cat : Π (A B : Cat), Cat;
injective symbol sfunc_func : Π [A B : Cat], τ (Obj (StrictFunctor_cat A B)) → τ (Obj
(Functor_cat A B));
```

This is a pragmatic move: strictness gives stable computation rules, and later "laxness witnesses reduce to identities" can be recovered as special cases once the simplicial machinery is strong enough.

5. Dependent Categories (`Catd`) and Grothendieck Constructions

5.1 Displayed categories as isofibrations

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

```
constant symbol Catd : Π (Z : Cat), TYPE;
```

In the paper we write E : `Catd` Z and read it as a fibration-like structure $p : \int E \to Z$ whose fibre over z is a category $E[z]$.

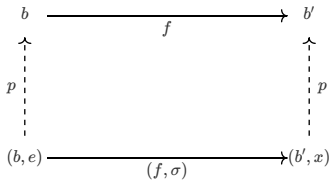
5.2 The Grothendieck constructor as computation

For an honest Cat-valued functor $M : Z \to \mathbf{Cat}$, emdash provides a displayed category `Fibration_cov_catd M : Catd Z` (think "the Grothendieck construction" $\int M$), and in this special case fibres compute definitionally:

```
constant symbol Fibration_cov_catd : Π [Z : Cat], τ (Obj (Functor_cat Z Cat_cat)) → Catd Z;
```

This is the main place where the kernel commits to concrete computations for `Catd` : general displayed categories are abstract, but Grothendieck ones compute.

Figure 3: Grothendieck morphisms lie over base arrows



Here (f, σ) is the usual Grothendieck idea: a morphism in the total category contains a base arrow f plus a fibre morphism σ with the correct transported source.

5.3 Slice-style displayed functors: `Functord_cat`

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

1. **General base map**: a displayed functor can 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 objects of the slice $\mathbf{Cat}/Z$.

emdash2 chooses 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 in the same base automatically.

The kernel still supports the "general base map" intuition via pullback: a functor over F becomes an ordinary slice-style functor into a pullback `Pullback_catd D F` .

5.4 Totals, projections, and "internalized" totalization

Given $M : Z \to \mathbf{Cat}$, the Grothendieck total category $\int M$ is represented as `Total_cat (Fibration_cov_catd M)` . The kernel commits to a definitional Σ-description of objects *only in this Grothendieck case*:

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

This is a key engineering boundary: for a generic displayed category $E : \mathrm{Catd}(Z)$ we do **not** force `Total_cat E` to be definitionally a Σ-type; it remains semantic data carried by `E` .

For composing constructions inside `Cat_cat` , emdash also packages "totalization" as a functor object:

```
symbol Total_func [Z : Cat] : τ (Obj (Functor_cat (Functor_cat Z Cat_cat) Cat_cat));
```

with a β-rule on objects `fapp0 (Total_func[Z]) M ↪ Total_cat (Fibration_cov_catd M)`. This is the same stable-head pattern as elsewhere: we want *a small head symbol* we can compose with, rather than unfolding a large definition every time.

5.5 Strict Grothendieck transport on fibre objects: `fib_cov_tapp0_fapp0`

If $M : Z \to \mathbf{Cat}$ is Cat-valued, then for a base arrow $f : z \to z'$ we can "transport" a fibre object $u \in M(z)$ to $f_!(u) := M(f)(u) \in M(z')$. emdash exposes this as a stable head `fib_cov_tapp0_fapp0` with strict functoriality on objects:

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

The orientation is cut-elimination style: nested transports fold to one transport along the composite. This strictness is explicitly temporary; it is a placeholder for the later lax/weak story where "functoriality of transport" comes with higher cells rather than with definitional equalities.

5.6 Fibrewise products of displayed categories: `Product_catd` and `prodFib`

Displayed categories over the same base admit a fibrewise product `Product_catd U A : Catd Z`, with fibres

$$(U \times A)[z] \equiv U[z] \times A[z].$$

For Grothendieck displayed categories $\int E$ and $\int D$, emdash includes a definitional rule

$$(\textstyle\int E) \times_Z (\int D) \;\rightsquigarrow\; \int (E \times D),$$

implemented via a stable-head functor `prodFib` for pointwise product of Cat-valued functors. This is a representative example of "rewrite hygiene": we introduce a small head so later rules can match without normalizing a huge composed expression in `Cat_cat`.

6. Dependent Arrow/Comma Categories: `homd_cov` and `homd_cov_int`

This section is the technical center of the paper. It explains the kernel constructions that we use to model "cells over a base arrow" without starting from a raw globular presentation.

6.1 Classical picture: dependent hom / comma object

Let:

- Z be a category,
- E be a dependent category over Z (morally $E : Z \to \mathbf{Cat}$),
- $W \in Z$ be a chosen base object, and $w \in E(W)$ be a chosen fibre object.

Given a base arrow $f : W \to z$ and an object $x \in E(z)$, we want the category of "arrows from the transport of w along f to x". In textbook fibration language this is:

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

This is a *dependent arrow category*: its objects are "triangles" with base edge f and a displayed edge in the fibre.

6.2 What `homd_cov` computes (in the Grothendieck case)

The kernel symbol `homd_cov` is declared abstractly, but it has a key computation rule when the displayed categories involved are Grothendieck constructions. In words:

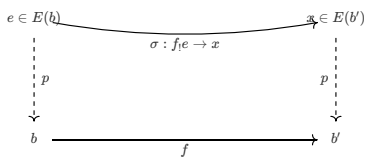
> if $E = \int E_0$ and $D = \int D_0$ are Grothendieck displayed categories, then `homd_cov` evaluated at a triple (z, d, f) reduces to the hom-category in the fibre $E_0(z)$ from the transported w to the image of d under the displayed functor.

This is precisely the "dependent arrow/comma" intuition.

At the definitional level, the key pointwise computation rule in `emdash2.lp` is (Grothendieck–Grothendieck case):

```
rule fapp0 (@homd_cov $Z (@Fibration_cov_catd $Z $E0) $W_Z $W
             (@Fibration_cov_catd $Z $D0) $FF)
           (Struct_sigma $z (Struct_sigma $d $f))
 ↪ @Hom_cat (fapp0 $E0 $z)
     (@fib_cov_tapp0_fapp0 $Z $E0 $W_Z $z $f $W)
     (@fdapp0 $Z (@Fibration_cov_catd $Z $D0) (@Fibration_cov_catd $Z $E0) $FF $z $d);
```

Figure 4: a displayed arrow over a base arrow



The slogan is: *a 2-cell is a 1-cell in a dependent arrow category*.

6.3 The "more internal" pipeline: `homd_cov_int_base`

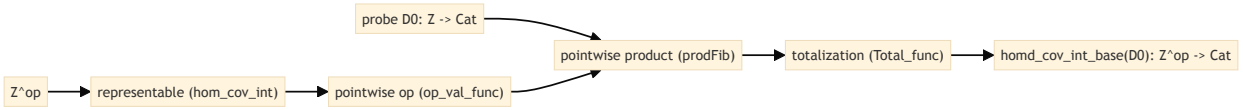
The file `emdash2.lp` contains a more internal version of the dependent hom construction, designed to be composed with other internal operations without exploding rewriting. It is built from stable-head building blocks (pointwise opposite, pointwise product, totalization, etc.). The key helper is:

```
symbol homd_cov_int_base [Z : Cat] (D0 : τ (Obj (Functor_cat Z Cat_cat)))
  : τ (Obj (Functor_cat (Op_cat Z) Cat_cat));
```

Conceptually, `homd_cov_int_base D0` is a Cat-valued functor on Z^{op} whose value at z packages:

- a representable $\mathrm{Hom}_Z(-, z)$ (built via `hom_cov_int` and then dualized),
- paired with the probe family $D0[z]$ (via `prodFib`),
- then "totalized" (via `Total_func`) so that later constructions can range over *base arrows* explicitly.

The point is not the exact combinatorics but the *shape*: we are building a simplicial indexing category (objects z together with base edges into z) in a way that remains computationally stable.



6.4 `homd_cov_int` as a displayed functor (current status)

The kernel exposes `homd_cov_int` as a displayed functor object (currently without computation rules). Informally it packages:

- a probe displayed category $\int D0$,
- a displayed functor $FF : \int D0 \to E$,
- and returns a displayed functor over E^{op} whose fibre over (z, e) classifies arrows "from e to $FF(d)$ over a base arrow".

In the source:

```
constant symbol homd_cov_int : Π [Z : Cat], Π [E : Catd Z],
  Π (D0 : τ (Obj (Functor_cat Z Cat_cat))),
  Π (FF : τ (Obj (Functord_cat (Fibration_cov_catd D0) E))),
  τ (Obj (Functord_cat (Op_catd E) ... ));
```

The important message for the reader is: *emdash organizes higher cells by iterating dependent arrow categories*; `homd_cov_int` is the internalized, compositional version of the triangle classifier.

7. Displayed Transfors and Simplicial Iteration

7.1 Displayed transfors: pointwise (`tdapp0_*`) and off-diagonal (`tdapp1_*`)

In addition to ordinary transfors between ordinary functors, the kernel includes **displayed transfors**: transformations between displayed functors over the same base Z.

If $FF, GG : E \to_Z D$ are displayed functors (objects of `Functord_cat E D`), then `Transfd_cat FF GG` is the category of displayed transfors. As with ordinary transfors, emdash provides stable-head projections rather than a record encoding:

- `tdapp0_fapp0` extracts the **diagonal component** at a base point (Y, V), i.e. a morphism in the fibre $D[Y]$:

```
 symbol tdapp0_fapp0 : Π [Z : Cat], Π [E D : Catd Z], Π [FF GG : τ (Obj (Functord_cat E
D))], Π (Y_Z : τ (Obj Z)), Π (V : τ (Obj (Fibre_cat E Y_Z))), Π (ϵ : τ (Obj (Transfd_cat FF
GG))), τ (Obj (Hom_cat (Fibre_cat D Y_Z) (fdapp0 FF Y_Z V) (fdapp0 GG Y_Z V)));
```

- `tdapp1_fapp0_funcd` (and internal variants like `tdapp1_int_*`) package the **off-diagonal component** over a displayed arrow $\sigma : e \to_f e'$ (surface syntax `ϵ_(σ)`), mirroring the ordinary `tapp1_*` story.

This mirrors the surface discipline in `docs/SYNTAX_SURFACE.md` : diagonal components are silent (`ϵ[e]`), while off-diagonal components are explicit computational interfaces.

One particularly important normalization rule connects identity displayed transfors to the *action on dependent homs*. Informally:

> the identity transfor 1_{FF} induces exactly the "functorial action of FF on homd-data".

In the kernel this is a fold from `tdapp1_*` at an identity to the stable head `fdapp1_funcd` . This is the displayed analogue of the philosophy "coherence is computation": functorial action on higher structure is obtained by normalizing a canonical expression rather than by proving a lemma.

7.2 From globes to simplices: triangles, surfaces, and "stacking"

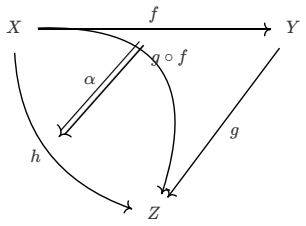
Classically, ω-categories are presented globularly: 0-cells, 1-cells, 2-cells between 1-cells, etc. emdash keeps the globular core (homs are categories), but it tries to *use simplicial indexing for computation*:

- a 2-cell is a triangle over a base edge;
- a 3-cell is a tetrahedron over a base triangle;
- and so on.

The kernel contains the beginnings of this "simplicial engine":

- `homd_cov` (triangle/surface classifier),
- and draft operations like `fdapp1_funcd` (dependent action on morphisms), intended to iterate the construction.

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



This is the common 2-categorical picture (a 2-cell between parallel composites). The kernel aim is to recover such cells as objects of a dependent hom construction, so that "stacking" and exchange laws can be derived from functoriality of the indexing.

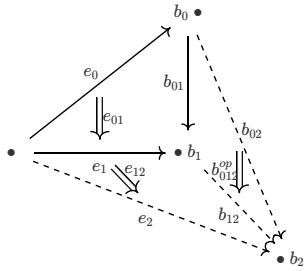
Figure 6: stacking 2-cells along a 1-cell (a tetrahedral "over-a-base-edge" picture)

The "stacking" operation (horizontal composition of 2-cells along a 1-cell rather than along a 0-cell) is easiest to see as a tetrahedron of base edges, with 2-cells comparing *direct* edges with *composite* edges.

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

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

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



In emdash terms, this is the kind of geometry the kernel is aiming to make *computational*: rather than proving a separate "exchange law", one wants an interface where these higher comparisons arise by functoriality/transport in a suitably internalized indexing.

8. Computational Adjunctions (cut-elimination rules)

The `emdash2.lp` file contains a draft interface for adjunctions inspired by Došen–Petrić's proof-theoretic view of categories.

8.1 Adjunction data as first-class terms

An adjunction is represented by:

- categories R, L,
- functors $LAdj : R \to L$ and $RAdj : L \to R$,
- transfors (unit and counit)

$$\eta : \mathrm{Id}_R \Rightarrow RAdj \circ LAdj, \qquad \epsilon : LAdj \circ RAdj \Rightarrow \mathrm{Id}_L.$$

In the kernel this is a type former:

```
constant symbol adj :
 Π [R L : Cat],
 Π (LAdj : τ (Obj (Functor_cat R L))),
 Π (RAdj : τ (Obj (Functor_cat L R))),
 Π (η : τ (Obj (Transf_cat (id_func R) (comp_cat_fapp0 RAdj LAdj)))),
 Π (ϵ : τ (Obj (Transf_cat (comp_cat_fapp0 LAdj RAdj) (id_func L)))),
 TYPE;
```

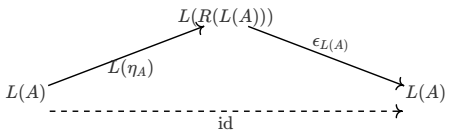
8.2 The triangle law as a rewrite rule

One of the most concrete successes of this approach is that a triangle identity can be oriented as a rewrite rule ("cut-elimination"). Very roughly (omitting indices), the rule has the shape:

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

This matches the abstract's goal: make the triangle identity a *definitional computation step* rather than a lemma.

Figure 7: the familiar triangle identity (as reduction)



In `emdash2.lp` the rule is implemented at the level of the stable heads for composition (`comp_fapp0`), functor action on morphisms (`fapp1_fapp0`), and arrow-indexed transfor components (`tapp1_fapp0`). The result is

that normalizing a composite term *performs* the triangle reduction.

This computational adjunction story is best viewed as an application of the *off-diagonal component infrastructure for transfors* (`tapp1_*` / `tdapp1_*` and their "functional" packaging), rather than as a special application of the dependent-hom layer. The dependent-hom constructions (`homd_cov` , `homd_cov_int`) target a different goal: simplicial organization of higher cells over chosen base arrows, so that stacking/exchange phenomena can later be obtained uniformly.

9. Metatheory and Normalization Discipline

emdash relies on Lambdapi's core metatheoretic contract: typechecking is done modulo a conversion relation generated by β-reduction plus user-supplied rewrite rules (and some built-in definitional equalities). Lambdapi checks subject-reduction for each rewrite rule; global termination/confluence are left to the user.

The practical question, therefore, is not "can we state the equations?" but "can we orient a useful subset as a terminating, robust normalization strategy?" The kernel comments call this **rewrite hygiene**.

9.1 Stable heads and canonicalization

The most frequent move is to introduce a *stable-head* symbol for a conceptually important operation and add a canonicalization rule that folds a large redex into that head. Examples:

- `fapp1_fapp0` folds "apply `fapp1_func` then `fapp0` " into a single head.
- `tapp0_fapp0` is the head for "component at Y", rather than unfolding it as an evaluation of a packaged functor.
- `fib_cov_tapp0_fapp0` is the head for Grothendieck-style object transport.

This makes rewriting predictable (matching sees the head) and keeps later rewrite rules small.

9.2 Rewriting vs unification rules

emdash uses both:

- `rule` for *computation* (normalization steps that should fire during reduction),
- `unif_rule` for *inference hints* (guiding elaboration/conversion without changing normal forms).

The slogan is: if a law should not change canonical normal forms (e.g. "the component at an identity arrow agrees with the diagonal component"), prefer `unif_rule` over `rule` .

9.3 Avoiding rewrite explosions

Two common failure modes for rewrite-based kernels are:

- **conversion blowups** caused by matching against too much inferred structure in a rule LHS;
- **non-termination** caused by loops between "expanding" and "folding" rules.

`emdash2.lp` mitigates this by (i) keeping inferred arguments as `_` on LHS patterns unless essential, and (ii) orienting many laws in a cut-elimination direction (nested structure folds to a single constructor).

9.4 Timeouts as a diagnostic tool

Because a hung typecheck often indicates rewrite/unification pathology, the repo's workflow treats a short timeout as a diagnostic: a timeout is a bug signal, not a reason to increase the timeout.

10. Implementation and Evaluation (January 2026 snapshot)

The kernel is intentionally "small but sharp". Some parts compute definitionally today; others are interfaces intended to be refined.

- **Computational today** (examples): opposites (`Op_cat`), products, Grothendieck fibres for `Fibration_cov catd` , strict Grothendieck object transport (`fib_cov_tapp0_fapp0`), pointwise computation for `homd_cov` in the Grothendieck–Grothendieck case, pointwise component extraction for transfors (`tapp0_fapp0`) and displayed transfors (`tdapp0_fapp0`), and a draft triangle cut-elimination rule for adjunctions.
- **Abstract / TODO** (examples): full computation rules for general displayed categories `E : Catd Z` , full simplicial iteration (`fdapp1_funcd` depends on more `homd_cov` infrastructure), and the user-facing surface syntax/elaboration layer (variance-aware binders, implicit coercions).

This division is deliberate: the kernel tries to avoid committing to heavy encodings (Σ-records for functors/transfors) until the rewrite story is stable.

10.2 Typechecking the kernel

The Lambdapi development is designed to typecheck quickly; long typechecks are treated as a symptom of rewrite/unification pathologies. The repo provides a timeout-protected check (`EMDASH_TYPECHECK_TIMEOUT=60s make check`) and a watch mode that logs to `logs/typecheck.log` .

The paper renderer is also exercised as a reproducible artifact:

- `npm run validate:paper -w print` validates all embedded Arrowgram/Vega-Lite JSON blocks.
- `npm run check:render -w print` runs validation + build + a headless browser console check (fails on KaTeX warnings and rendering errors).

11. Limitations, Related Work, and Future Directions

11.1 Limitations (current kernel snapshot)

The current kernel is intentionally incomplete; some gaps are design choices, others are pending infrastructure.

- **No global record encoding for functors/transfors.** Objects of `Functor_cat` and `Transf_cat` are intentionally abstract; the user-facing interface is via projection heads (`fapp0` , `tapp0_*` , `tapp1_*` , etc.).
- **Displayed categories are semantic except for Grothendieck constructors.** Computation rules for `Catd` focus on special constructors like `Fibration_cov_catd` ; generic `Catd` is meant to model isofibrations, but the explicit cleavage/iso-lift interface is not yet exposed.

- **Strictness placeholders.** Some computation rules (e.g. Grothendieck object transport) are strict today and are intended to be relaxed to a lax/weak story with higher cells.
- **"Metatheory by engineering."** Termination/confluence are managed by rewrite discipline and tests rather than by a formal metatheorem at this stage.

11.2 Related ideas and influences

emdash draws on three complementary threads:

1. **Proof-theoretic categories** (Došen–Petrić): treat categorical equalities as cut-elimination / normalization steps.
2. **Type-theoretic higher categories** (e.g. Finster–Mimram): define weak ω-categories by internal type-theoretic structure.
3. **Parametricity-inspired internalization** (e.g. "bridge types"): avoid explicit interval objects while retaining internal reasoning principles reminiscent of parametricity.

Our distinctive emphasis is *computational organization of higher cells over base arrows* via dependent arrow/comma categories, combined with an explicit off-diagonal component interface for transfors that supports computation-first coherence laws.

11.3 Next steps

- Finish the simplicial iteration story (derive/justify the external heads like `fdapp1_funcd` from internal `tdapp1_int_*` pipelines; extend computation beyond the Grothendieck case).
- Add the missing user-facing layer: a variance-aware elaborator that makes `docs/SYNTAX_SURFACE.md` executable (as in v1's TypeScript kernel, but for the v2 primitives).
- Extend the adjunction interface from the first triangle cut-elimination rule to a robust set of whiskering/triangle/exchange normalizations.
- Port the warm-up libraries: re-express the `cartierSolution14` "compute by adjunction/colimit" patterns and the `cartierSolution16` sieve/site/sheafification interfaces within the v2 heads (`Functor_cat` , `Transf_cat` , and eventually a displayed-profunctor layer), so they become stable regression tests for the kernel discipline.

12. Conclusion

emdash is an attempt to make a small computational kernel in which:

- higher categorical structure is *internal* (functors and transformations are first-class objects),
- equalities are *operational* (rewrite rules),
- and higher cells are organized *simplicially* via dependent arrow categories (`homd_cov` , `homd_cov_int`).

The most concrete result so far is a faithful computational core for (parts of) Grothendieck-style dependent categories and a first computational adjunction rule. The next step is to finish the simplicial iteration so that exchange/stacking laws and higher triangle identities can also become normalization steps.

References

(Placeholder list; the final submission should expand these to full bibliographic entries with venue/year/identifiers.)

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 to the code (kernel identifiers → math)

- `Cat`, `Obj`, `Hom_cat`: category classifier, object groupoid, hom-category (iterated for higher cells).
- `Functor_cat`, `fapp0`, `fapp1_func`, `fapp1_fapp0`: functors as objects; object and hom action; stable-head application.
- `Transf_cat`, `tapp0_fapp0`, `tapp1_fapp0`: transfors; object-indexed and arrow-indexed components (lax naturality lives "off-diagonal").
- `Transfd_cat`, `tdapp0_fapp0`, `tdapp1_fapp0_funcd`, `fdapp1_funcd`: displayed transfors; pointwise components in fibres; packaged off-diagonal components over displayed arrows; action on dependent-hom data.
- `Catd`, `Fibre_cat`, `Functord_cat`: displayed categories (isofibrations); fibres; displayed functors over a fixed base.
- `Fibration_cov_catd`: Grothendieck construction $\int M$ for $M : Z \to \mathbf{Cat}$ (this is where definitional computation for fibres exists).
- `hom_cov`, `hom_cov_int`: (co)representables / internalized hom functors.
- `homd_cov`: dependent arrow/comma construction (triangle classifier), with a computation rule in the Grothendieck case.
- `homd_cov_int_base`, `homd_cov_int`: internalized pipeline and displayed packaging for the same idea.
- `adj` and the triangle rewrite: adjunction data and (draft) cut-elimination rule.