Emdash — A Dependently Typed Logical Framework for Computational Synthetic Category Theory and Functorial Elaboration

by Author

Abstract. We present Emdash, a novel dependently typed logical framework designed to support computational synthetic category theory, drawing inspiration from Kosta Dosen's functorial programming paradigm. Emdash integrates categorical primitives—such as categories, objects, morphisms, and functors—directly into its λΠ-calculus core, facilitating reasoning and computation in a style closer to mathematical practice. The system features a bidirectional type checker with unification-based hole solving for interactive proof, definitional equality via βδι-reduction (including user-supplied rewrite rules and unfolding of injective constants), and Higher-Order Abstract Syntax (HOAS) for binders. A key contribution of Emdash is the concept of *functorial elaboration*, where kernel-level constructors for structures like functors not only receive their components (e.g., object and arrow mappings) but also definitionally verify their coherence laws (e.g., functoriality) during the elaboration process itself, throwing a `CoherenceError` upon failure. Implemented in TypeScript and formally specified in a Lambdapi dialect, Emdash demonstrates a practical pathway from specification to a working kernel. This paper details the Emdash framework, its core algorithms, its interactive proof mode, its alignment with its formal specification, and its role as the formal engine for hotdocX, a web-based platform for AI-assisted formalization of mathematical documents. We report on the successful implementation and validation of the system's core features through a comprehensive test suite.

Keywords: Dependent Type Theory, Logical Frameworks, Category Theory, Functorial Programming, Synthetic Mathematics, Interactive Theorem Proving, AI-Assisted Formalization, Lambdapi, TypeScript.

1. Introduction: Bridging the Chasm

A deep chasm has long existed between the fluid, intuitive world of informal mathematical creativity and the rigid, explicit world of formal computational rigor. While modern mathematics, particularly category theory, thrives on abstraction and structural reasoning, its translation into machine-verifiable artifacts remains a formidable challenge. The dominant approach has been to encode these rich structures within powerful but foundational proof assistants like Coq, Agda, or Lean. In such systems, a category is not a primitive notion but a record or a structure built from set-theoretic or type-theoretic components, and its laws are propositions to be proven as separate lemmas. This approach, while tremendously successful, creates a conceptual gap: the mathematician thinks in terms of "functors" and "natural transformations," but the machine sees only "records satisfying certain proven predicates." The structural properties feel tacked on, rather than intrinsic.

An alternative vision, championed by Kosta Dosen and his collaborators in works on "functorial programming" and "proof-theoretical coherence" [1, 2], suggests a different path. This paradigm advocates for a substructural and inherently computational approach to logic where the core entities of a mathematical domain—such as categories and functors—are not merely encoded but are themselves the primitive building blocks of the formal language. In such a system, proofs of structural integrity would not be separate objects but would manifest as computations, governed by a disciplined rewrite system that is part of the theory's definitional equality. The very act of constructing an object would guarantee its coherence.

This paper introduces **Emdash**, a dependently typed logical framework developed in TypeScript, with the explicit goal of exploring and implementing these functorial programming principles. Emdash distinguishes itself by adopting a *synthetic* methodology: fundamental categorical notions are treated as primitive types and term constructors within its λΠ-calculus core. This approach, combined with the expressive power of dependent types, allows for a more direct and natural articulation of categorical arguments and constructions. The design and implementation of Emdash are systematically guided by a formal specification written in a Lambdapi dialect [3], a logical framework well-suited for prototyping type systems modulo rewrite rules. This ensures a close correspondence between the intended semantics and the behavior of the practical system, a significant portion of which is generated directly from this specification.

A central contribution of our work, and a direct realization of the Dosen-inspired philosophy, is the concept of **functorial elaboration**. In Emdash, defining an instance of a structured object, such as a functor, is not merely a matter of providing its components (e.g., an object map `fmap0` and a morphism map `fmap1`). The system's elaboration engine, via the `MkFunctorTerm` kernel primitive, *computationally verifies* the required coherence laws (e.g., `F(g∘f) = F(g)∘F(f)`) as part of the term's type checking. This check is performed by normalizing both sides of the law under a generic context and asserting their definitional equality. If the laws do not hold, elaboration fails with a specific `CoherenceError`, making structural integrity a definitional property rather than a propositional one.

The impetus for Emdash, however, extends beyond theoretical exploration. It is architected to serve as the formal kernel for **hotdocX** [7], an AI-assisted, web-based platform designed to transform mathematical documents into executable, verifiable, and interactive formal content. This "papers-with-code" vision for mathematics leverages Emdash's robust type checking, computational equality, and interactive proof capabilities to animate mathematical arguments.

This paper provides a comprehensive and didactic introduction to the Emdash logical framework.

- In Section 2, we detail the core type theory and the synthetic categorical primitives that form its foundation.
- In Section 3, we present the formal specification of Emdash in the Lambdapi logical framework, which guides the entire implementation.
- In Section 4, we dissect the elaboration engine, explaining the interplay of bidirectional type checking, implicit argument handling, and unification-based constraint solving.
- Section 5 is dedicated to our central contribution, functorial elaboration, providing a deep dive into the `MkFunctorTerm` primitive and its coherence-checking mechanism.
- In Section 6, we describe the system's full-featured interactive proof mode, demonstrating how users can construct proofs by refining holes with a suite of tactics.
- In Section 7, we provide a crucial overview of the system's validation through its comprehensive test suite, confirming the correct behavior of all major features.
- Finally, in Sections 8 and 9, we situate Emdash within the landscape of related work and outline the ambitious roadmap for its future development.

2. The Emdash Logical Framework

Emdash is built upon a λΠ-calculus modulo a user-extendable theory. Its architecture is designed for clarity, extensibility, and a direct correspondence with its formal specification, facilitating both understanding and verification.

2.1. Core Type Theory

The foundational type system of Emdash includes the standard components of a dependent type theory, with specific implementation choices made for pragmatic implementation.

Sorts and Terms. Emdash currently employs a single sort `Type`, with the judgment `Type : Type`. This is a common simplification in prototypes of type theories that avoids the complexities of a full universe hierarchy. While known to be inconsistent if used without restriction (Girard's Paradox), it is sufficient for the current scope of the project, which focuses on the categorical machinery.

The syntax of terms, defined in `src/types.ts`, is built around the `Term` data type. Key constructors include:

- `Type()`: The term representing the sort of all types.
- `Var(name: string, isLambdaBound: boolean)`: Variables. The `isLambdaBound` flag is a crucial implementation detail that distinguishes variables bound by a lambda from free or globally-defined variables.
- `Lam(paramName, icit, paramType?, body)`: Lambda abstraction, implementing **Higher-Order Abstract Syntax (HOAS)**. The `body` is a TypeScript function `(v: Term) => Term`, which delegates the complexities of variable substitution and α-equivalence to the host language. `icit` denotes implicitness (`Impl`) or explicitness (`Expl`). The `_isAnnotated` flag tracks whether `paramType` was user-provided, guiding the bidirectional checker.
- `App(func, arg, icit)`: Application, with an explicitness flag.
- `Pi(paramName, icit, paramType, bodyType)`: Dependent function type (Π-type), also using HOAS for its body.
- `Hole(id: string, ...)`: Metavariables (e.g., `?h0`, `?h1`) used for unification and interactive proof. A solved hole has its optional `ref` field point to its solution.

Contexts and Globals. Reasoning occurs within a `Context`, implemented as a list of `Binding` objects (`[{ name, type, definition?, icit }, ...]`). This structure naturally handles lexical scoping. A global environment, `globalDefs`, stores `GlobalDef` objects, which map string names to terms with associated types and optional values. This global context is mutable and can be extended by the user. The `defineGlobal` function in `src/globals.ts` populates this map, allowing the user to specify key properties for new definitions:

- `isConstantSymbol`: A boolean flag that prevents a global definition from being unfolded during δ-reduction. This is essential for defining primitive constants whose computational behavior is governed by rewrite rules rather than by a definition. It also prevents such symbols from appearing as the head of a rewrite rule's left-hand side.

- `isInjective`: A boolean flag that marks a constructor as injective for the unification algorithm. For an injective symbol `f`, an equality `f(t₁) ≡ f(t₂)` can be decomposed into `t₁ ≡ t₂`.

Definitional Equality. The notion of definitional equality (≡) is central to Emdash. It is decided by the function `areEqual(t₁, t₂, Γ)` (in `src/equality.ts`), which operates by first reducing both terms to their Weak Head Normal Form (WHNF) and then performing a recursive structural comparison. The `whnf` function (in `src/reduction.ts`) is the engine of computation and incorporates several reduction strategies:

- **β-reduction**: The standard rule `(λx:A. t) u ↪ t[u/x]`.
- **δ-reduction**: The unfolding of global definitions (unless marked as `isConstantSymbol`) and local `let`-bindings from the context.
- **User-defined ι-reduction**: The application of user-provided rewrite rules, which are attempted before any other reduction step.
- **Kernel ι-reduction**: A set of built-in computational rules for Emdash's synthetic primitives.
- **η-contraction**: `λx. F x ↪ F`, performed if the `etaEquality` flag is enabled and `x` is not free in `F`.
- **Hole Dereferencing**: Following the `ref` chain of solved `Holes` via the `getTermRef` utility.

2.2. The Emdash Parser

To enhance usability, Emdash employs a custom parser, implemented using the `parsimmon` library in `src/parser.ts`. This parser supports a more elaborate and user-friendly syntax than a minimal kernel might provide, moving closer to the syntax of languages like Agda or Lean. Key features include:

- **Keywords:** Recognizes keywords like `let`, `in`, `Type`, and `fun` (as an alternative to `\`) for lambda abstractions.
- **Flexible Binders:** It supports multiple binders in a single construct. For example, `fun x y. body` is parsed as `Lam("x", ..., _ => Lam("y", ..., ...))`.
- **Grouped Typed Binders:** It allows grouping variables of the same type, such as `(x y : Nat) -> Type`, which is parsed into a nested Pi-type `(x : Nat) -> (y : Nat) -> Type`. This applies to both explicit `()` and implicit `{}` binders.

- **Implicit Binders:** Implicit arguments in lambdas and Pi-types are denoted with curly braces, e.g., `fun {A : Type}. ...` or `{A : Type} -> ...`.
- **Untyped Binders:** In lambda expressions, binder types are optional (e.g., `fun x. x`), leaving them to be inferred by the bidirectional type checker.
- **Operator Precedence:** The parser correctly handles standard operator precedence, with function application binding more tightly than the function arrow `->`. For example, `f x -> g y` is parsed as `(f x) -> (g y)`.

This parser provides a crucial layer of syntactic sugar, making the formal language more ergonomic without complicating the core term structure.

2.3. Extensibility: Rules and Globals

A key feature of Emdash as a logical framework is its extensibility. The user can augment the system's definitional equality and unification behavior.

- **Global Definitions (`defineGlobal`):** As described earlier, users can add new constants and functions to the global scope. This function uses the elaboration engine to type-check the provided type and value, ensuring that all global definitions are well-formed.
- **Rewrite Rules (`addRewriteRule`):** Users can add their own computational rules. The `addRewriteRule` function takes a raw LHS and RHS pattern. It then elaborates both sides, inferring the types of any pattern variables (`$x`, `$y`, etc.) from the LHS and ensuring the RHS has the same type. This process guarantees that all user-defined rules are type-safe. These rules are stored in `userRewriteRules` and are the first to be consulted by the `whnf` reducer.
- **Unification Rules (`addUnificationRule`):** To guide the unification process in cases of ambiguity (e.g., for non-injective function symbols or to state associativity properties), users can provide unification rules. A rule of the form `lhs₁ ≡ lhs₂ ↪ [rhs₁ ≡ rhs₂, ...]` instructs the unifier that if it encounters a constraint matching `lhs₁ ≡ lhs₂`, it can replace it with the set of new constraints on the right-hand side.

This three-pronged approach to extensibility allows Emdash to be tailored to specific mathematical domains, with custom equational theories and unification heuristics.

3. A Blueprint for Synthetic Reasoning: The Lambdapi Specification

The implementation of Emdash is not ad-hoc; it is guided by a formal specification written in a dialect of Lambdapi [3], a logical framework based on the λΠ-calculus modulo theory. This specification, found in `emdash_specification_lambdapi.lp.txt`, serves as a precise, executable blueprint for the system's core logic. Using Lambdapi allows us to define the types, constants, and computational behavior of our synthetic primitives in a high-level, declarative style. This section provides a didactic walkthrough of this specification, explaining not just *what* is specified, but *why* it is specified in this particular way.

3.1. Why Lambdapi?

Lambdapi is an ideal choice for specifying a system like Emdash for several reasons. It is a logical framework, meaning it is designed to define other logical systems. Its core, the λΠ-calculus, provides dependent types, which are essential for expressing the relationships between our categorical primitives. Most importantly, its "modulo theory" aspect means that definitional equality is not fixed but can be extended with user-defined rewrite rules. This aligns perfectly with our goal of building a system where coherence laws are computational. The commands `constant symbol`, `rule`, and `unif_rule` provide a clear, declarative language for capturing the essence of our synthetic theory.

3.2. The Basic Building Blocks

At the foundation of any category theory are the notions of categories, objects, and morphisms. In our synthetic approach, these are not constructed from sets; they are primitive notions of the theory, introduced with the `constant symbol` command, which declares a symbol without giving it a definition.

```
constant symbol Cat : TYPE;
injective symbol Obj : Cat → TYPE;
injective symbol Hom : Π [A : Cat] (X: Obj A) (Y: Obj A), TYPE;
```

Let's break this down:

- `Cat : TYPE;` declares that `Cat` is a type. An inhabitant of `Cat` is a category.
- `Obj : Cat → TYPE;` declares that `Obj` is a function that takes a category and returns a type. `Obj C` is the type whose inhabitants are the objects of the category `C`. The `injective` keyword is a crucial hint to the type checker's unification engine: if it needs to solve `Obj A ≡ Obj B`, it can safely conclude that `A ≡ B`.
- `Hom : Π [A : Cat] (X: Obj A) (Y: Obj A), TYPE;` defines the type of morphisms. It is a dependent type: it takes a category `A` (as an implicit argument, denoted by `[]`), and two objects `X` and `Y` of that category, and returns the type `Hom X Y` of morphisms from `X` to `Y`.

These three declarations form the static, non-computational bedrock of our theory. The computational aspects arise when we define how these structures relate and compose.

3.3. The Synthetic Leap: Defining Structure by Fiat

The true power of the synthetic approach becomes apparent when we define more complex structures like the functor category. In a traditional setting, one would define a functor as a record containing an object map and an arrow map, then define a natural transformation, and finally prove that these constructions form a category. In Emdash, we take a more direct route: we declare that the structure of the functor category *is* the structure of functors and natural transformations.

```
constant symbol Functor_cat : Π(A : Cat), Π(B : Cat), Cat;

// Objects of the functor category ARE functors
unif_rule Obj $Z ≡ (Functor $A $B) ↪ [ $Z ≡ (Functor_cat $A $B) ];

// Morphisms in the functor category ARE natural transformations
rule @Hom (Functor_cat _ _) $F $G ↪ Transf $F $G;
```

This is a profound statement. The `unif_rule` tells the unifier: "If you are ever trying to make the type `Obj Z` equal to the type `Functor A B`, you can solve this by unifying `Z` with `Functor_cat A B`." This essentially equates being an object in the functor category with being a functor. The `rule` is even more direct: it states that the type of morphisms between two functors `F` and `G` in the functor category *is definitionally equal to* the type of natural transformations between them. We do not prove this; we *define* it to be so. This is the synthetic leap.

A similar move is made for the category of sets, `Set`, which is foundational for many categorical constructions.

```
constant symbol Set : Cat;
unif_rule Obj $A ≡ Type ↪ [ $A ≡ Set ];
rule @Hom Set $X $Y ↪ (τ $X → τ $Y);
```

Here, we declare that the objects of `Set` are simply the system's base types (`Type`), and the morphisms are functions (`τ $X → τ $Y`). This makes `Set` a computational model of the category of types (or small groupoids) inside our theory.

3.4. The Computational Core: Coherence as Rewriting

The philosophy of making properties computational is realized through rewrite rules.

Functoriality: A functor `F` must preserve composition: `F(g ∘ f) = F(g) ∘ F(f)`. In Emdash, this is not a proposition to be proven for each functor. It is a global rewrite rule that is part of the definition of composition itself.

```
symbol compose_morph : Π [A : Cat] ...; // Morphism composition

// F a' ∘> F a  ↪  F (a' ∘> a)
rule compose_morph (@fapp1 _ _ $F $Y $Z $a) (@fapp1 _ _ $F $X $Y $a')
  ↪ fapp1 $F (compose_morph $a $a');
```

This rule instructs the normalization engine: "Any time you see a composition of two morphisms that are themselves the result of applying a functor `F`, rewrite it to `F` applied to the composition of the original morphisms." This makes functoriality a computational reduction, not a logical deduction.

Naturality and the Yoneda Insight: The naturality law for a transformation `ϵ : F ⇒ G` is `ϵ_Y ∘ F(f) = G(f) ∘ ϵ_X`. Simply adding this as a rewrite rule is problematic, as it's not obvious which direction is simplifying. This is where Dosen's key insight comes into play. The composition `G(f) ∘ ϵ_X` can be viewed not as a standard composition, but as the *action* of the morphism G(f) on the element ϵ_X in the hom-set Hom(F(X), G(Y)). This action is precisely what the Yoneda embedding Hom(-, c) provides.

Our specification captures this by defining the Yoneda embedding hom_cov and orienting the naturality rule accordingly.

```
// The Yoneda embedding as a primitive functor
constant symbol hom_cov : Π [A : Cat], Π (W: Obj A), Functor A Set;

// The action of the Yoneda functor is composition
unif_rule (fapp1 (hom_cov $W) $a) $f ≡ compose_morph $a $f ↪ [ tt ≡ tt ];

// The naturality rule, oriented using the Yoneda action
//  ϵ._X ∘> G a  ↪  (F a) _∘> ϵ._X'
rule compose_morph (@fapp1 _ _ $G $X $X' $a) (@tapp _ _ $F $G $ϵ $X)
  ↪ fapp1 (hom_cov _) (tapp $ϵ $X') (fapp1 $F $a);
```

The left-hand side is a standard composition. The right-hand side is rewritten into the form fapp1 (hom_cov _) ..., which represents the Yoneda action. This rule is not just a statement of equality; it is a directed computation that replaces a standard composition with a more primitive action, effectively implementing the naturality check as a reduction step. The benchmark for this specification is its ability to prove that the "super Yoneda functor" is indeed a functor using only definitional equality (reflexivity), which has been successfully verified.

3.5. Glimpsing the Future: Specifying Higher Structures

This specification methodology extends naturally to higher-categorical structures. The path towards ω-categories is paved by internalizing the comma category construction. Given a functor M : A -> Cat, the total category Total_cat M (the Grothendieck construction) can be specified. Its objects are pairs (X, f) where X is an object of A and f is an object of M(X).

```
constant symbol Total_cat [A : Cat] (M: Functor A Cat_cat): Cat;
rule Obj (Total_cat $M) ↪ `τΣ_ X : Obj _, Obj_Type (fapp0 $M X);
```

Here, τΣ_ is the dependent sum type. This rule defines the objects of the total category synthetically. A similar, more complex rule defines the morphisms. This allows us to represent fibrations and, by iteration, higher-categorical structures in a purely computational and synthetic manner. This specification-driven approach provides a solid formal foundation for Emdash, ensuring that its implementation is a faithful realization of a well-defined and computationally-oriented synthetic type theory.

4. Breathing Life into Syntax: The Elaboration Engine

The heart of Emdash is its elaboration engine, which transforms raw, potentially ambiguous user input into fully typed, unambiguous terms. This process is a sophisticated dance between type inference, type checking, and constraint solving, designed to make the formal language feel as natural as possible.

4.1. The Bidirectional Dialogue: `infer` and `check`

Imagine a dialogue between a user and the system. Sometimes the user provides a term and asks, "What is this?" This is **inference**. Other times, the user provides a term and a type, asserting, "This thing has this type." This is **checking**. Emdash's elaborator, driven by the `elaborate(term, expectedType?)` function, formalizes this dialogue.

- **Inference Mode (`infer(Γ, t)`):** When no `expectedType` is given, `elaborate` calls `infer`. The `infer` function acts like a detective, examining the term `t` to deduce its type. For example, if it sees a variable `x`, it looks up its type in the context `Γ`. If it sees an application `f a`, it first infers the type of `f`, ensures it's a function type `Π(x:A), B`, checks that the argument `a` has type `A`, and concludes that the whole application has type `B` (with `a` substituted for `x`).
- **Checking Mode (`check(Γ, t, T)`):** When an `expectedType T` is provided, `elaborate` calls `check`. Here, the system's job is verification. It uses the `expectedType` as a guide. For instance, to check if an unannotated lambda `λx. e` has the type `Π(y:A), B`, it doesn't need to guess the type of `x`. It knows `x` must have type `A`. It then extends its context with `x:A` and recursively checks if the body `e` has the type `B`.

This bidirectional approach is not just an implementation detail; it is fundamental to the system's usability. It allows the user to omit redundant type annotations, as the system can often infer them in checking mode, while still providing the power to synthesize types from first principles in inference mode.

4.2. The Art of Implicitness

A dependently typed language without good support for implicit arguments can become overwhelmingly verbose. Emdash employs three mechanisms to manage this, making interaction feel more natural.

1. **Implicit Application Insertion:** Consider a function `const : {A B : Type} -> A -> B -> A`. A user would prefer to write `const true 5` rather than the fully explicit `const {Bool} {Nat} true 5`. When the elaborator infers the type of `const`, it sees it begins with implicit Π-binders (`{A : Type}` and `{B : Type}`). Before proceeding to check the application to `true`, the `insertImplicitApps` function automatically inserts holes for these implicit arguments, effectively transforming the user's term into `const {?h0} {?h1} true 5`. These holes, `?h0` and `?h1`, are metavariables that the unification engine will later solve to be `Bool` and `Nat`, respectively, based on the types of the subsequent explicit arguments.

2. **Implicit Lambda Insertion (Eta-Expansion):** The converse situation also occurs. Suppose the system *expects* a function that takes an implicit argument, for instance, a term of type `Π {A:Type}. List A`. If the user provides a term `t` that is not an implicit lambda, the elaborator doesn't immediately fail. Instead, it helpfully wraps the term, transforming the goal from `Γ ⊢ t ⇐ Π {A:Type}. List A` into a new goal of checking `t` in a context where `A` is known: `Γ, A:Type ⊢ t ⇐ List A`. The final elaborated term becomes `λ {A:Type}. t'`, where `t'` is the result of checking `t` in the extended context. This is a form of eta-expansion that makes the system more flexible.

3. **Kernel Implicit Handling:** Some of Emdash's primitive term constructors, like `FMap0Term` (functor application on objects), have implicit arguments that are structurally necessary but tedious to write (e.g., the domain and codomain categories of the functor). The `ensureKernelImplicitsPresent` function, guided by the `KERNEL_IMPLICIT_SPECS` specification, automatically inserts holes for these missing arguments before the main elaboration process even begins. This allows the user to write `FMap0Term(F, X)` and have the system automatically fill in the details to `FMap0Term(F, X, catA_IMPLICIT: ?h2, catB_IMPLICIT: ?h3)`.

4.3. The Unification Engine: Solving for '?'

The elegance of the bidirectional algorithm and implicit argument handling rests on a powerful unification engine to solve for the unknown holes (`?h...`) that are generated.

Constraint Generation and Solving. The elaborator is optimistic. When it requires two terms to be equal, it doesn't fail immediately. Instead, it adds a `Constraint` object `{ t1, t2 }` to a global list. The `solveConstraints` function is then called, which loops through the constraints, attempting to solve them using the `unify` function until no more progress can be made.

The `unify` Algorithm. The `unify` function is the heart of the solver. Its strategy is a cascade of techniques:

1. **Hole Unification:** The simplest case is unifying a hole `?h` with a term `t`. The engine first performs an **occurs check** to ensure `t` does not contain `?h`, preventing circular definitions like `?h := f(?h)`. If the check passes, it solves the hole by setting its internal reference to `t`.
2. **Higher-Order Unification:** Emdash supports a decidable fragment of higher-order unification known as **Miller's pattern unification**. This is triggered for "flex-rigid" pairs, such as `?M x y ≡ g x y`, where `?M` is a hole (the "flexible" part) applied to a spine of distinct bound variables (`x`, `y`), and the other side is a "rigid" term. The `solveHoFlexRigid` procedure can solve for `?M` by abstracting the rigid term over the spine, yielding the solution `?M := λa b. g a b`. This is crucial for matching and applying rewrite rules with higher-order pattern variables.
3. **Structural Decomposition:** For terms with the same "injective" head symbol (e.g., `Obj A ≡ Obj B`), the unifier can decompose the problem, creating a new constraint for their arguments (`A ≡ B`).
4. **Reduction and Rules:** If direct structural methods fail, `unify` reduces both terms to their Weak Head Normal Form (WHNF) and retries. If that still fails, it consults the user-defined `unificationRules` for guidance before finally admitting failure.

This combination of bidirectional checking and unification-based constraint solving gives Emdash a powerful and flexible elaboration engine capable of inferring a great deal of information, making the surface language more ergonomic for the user.

5. The Payoff: Coherence as a Definitional Check

A central and novel contribution of Emdash is its mechanism for enforcing coherence laws as part of definitional equality. This section delves into this concept of "functorial elaboration," which represents the practical payoff of the system's synthetic, computation-first philosophy.

5.1. The Problem: Separating Data from Proof

In traditional formalizations, defining a structured object is a two-step process. First, you provide the data (the components), and second, you provide the proofs (the laws). For a functor `F : C → D`, you would define:

1. **Data:** An object map $F_0 : Ob(C) \to Ob(D)$ and a morphism map $F_1 : Hom(X,Y) \to Hom(F_0X, F_0Y)$.
2. **Proof:** A separate lemma, say `F_preserves_comp`, proving that $F_1(g \circ f) = F_1(g) \circ F_1(f)$.

This separation, while logically sound, feels artificial. To a mathematician, a functor *is* a structure-preserving map; the preservation of composition is part of its essence, not an afterthought. Emdash aims to close this gap.

5.2. The Solution: The `MkFunctorTerm` Primitive

Emdash introduces a special kernel term constructor, `MkFunctorTerm(domainCat, codomainCat, fmap0, fmap1, proof?)`. This term is not just a passive container for data; it is an active instruction to the elaboration engine to perform a coherence check.

When the elaborator's `infer` function encounters a `MkFunctorTerm`, it triggers a special procedure, `infer_mkFunctor`, detailed in `elaboration.ts`. This procedure is the heart of functorial elaboration. Let's walk through its steps with a narrative lens.

Step 1: Checking the Components. First, the system acts as a diligent student, checking that the provided components have the correct types. It verifies that `domainCat` and `codomainCat` are indeed categories. It then checks that `fmap0` is a valid object map (`Obj domainCat → Obj codomainCat`) and that `fmap1` is a valid, dependently-typed morphism map.

Step 2: Staging the Coherence Law. If the components are well-typed, the system prepares to test the functoriality law. If no explicit proof term is provided, it synthesizes the two sides of the equation it needs to verify. It creates a hypothetical scenario—a fresh context `lawCtx` containing generic objects `X, Y, Z` and generic morphisms `f : Hom X Y` and `g : Hom Y Z`. In this hypothetical world, it constructs the two terms to be compared:

- `LHS := compose_morph (fmap1 g) (fmap1 f)` (The image of the composition)
- `RHS := fmap1 (compose_morph g f)` (The composition of the images)

Step 3: The Moment of Truth - Computational Verification. This is the core of the check. The system does not try to *prove* `LHS = RHS`. Instead, it *computes* their normal forms within the hypothetical context `lawCtx`.

- `normLhs := normalize(lhs, lawCtx)`
- `normRhs := normalize(rhs, lawCtx)` This normalization process is the ultimate test. It will unfold the definitions of `fmap1` and `compose_morph` provided by the user and apply any relevant rewrite rules. If the definitions of `fmap1` and `compose_morph` are correct and work in harmony, the two terms should compute to the exact same normal form.

Step 4: The Verdict. The system then asserts their equality using `areEqual(normLhs, normRhs, lawCtx)`.

- **Success:** If `areEqual` returns `true`, the check passes. The elaborator successfully returns the `MkFunctorTerm`, now certified as coherent, with its type `FunctorTypeTerm(domainCat, codomainCat)`. The term is accepted into the system.
- **Failure:** If `areEqual` returns `false`, the provided maps do not respect functoriality. The elaboration process is immediately aborted, and a specific `CoherenceError` is thrown. This error is not a generic type error; it is highly informative, indicating that the "Functoriality check failed" and printing the two non-equal normal forms, allowing the user to debug their definitions.

This "functorial elaboration" mechanism is a powerful design pattern. It shifts the burden of proving structural integrity from the user to the system's computation engine. A term's well-formedness now includes not just its type signature but also its adherence to definitional laws.

This is particularly valuable in a synthetic setting, where the goal is to work directly with structures that are correct-by-construction.

6. Interactive Theorem Proving

Beyond its role in elaborating complete terms, Emdash is also designed for interactive theorem proving, where a user constructs a proof term step-by-step. A proof-in-progress is represented by a term containing unsolved `Hole`s, each representing a subgoal. The `src/proof.ts` module provides a lightweight but effective API for this workflow.

Goal Inspection. The first step in interactive proving is to understand the current state.

- `findHoles(proofTerm)`: This function traverses the entire proof term, including the values of global definitions, and returns a flat list of all unsolved `Hole`s.
- `getHoleGoal(proofTerm, holeId)`: To provide meaningful feedback for a specific goal, this function finds the hole with the given `holeId` and reconstructs its local context and expected type. It does this by traversing the term from the root, accumulating binders (`Lam`, `Pi`) into a `Context` object. The hole's expected type is retrieved from its `elaboratedType` field, which is populated during the initial elaboration of the proof statement.
- `reportProofState(proofTerm)`: This function combines the above to generate a human-readable report, listing each unsolved goal with its context and type, similar to the display in proof assistants like Agda or Lean.

Proof by Refinement. Progress is made by "refining" holes with more detailed terms. This is managed by a small set of primitive "tactics."

- `refine(proofTerm, holeId, refinementTerm)`: This is the core primitive. It takes the ID of a hole to solve and a `refinementTerm`. It first uses `getHoleGoal` to find the hole's context and expected type. Then, it calls the main `elaborate` function to type-check `refinementTerm` within that context and against that type. If elaboration succeeds, the hole is solved by setting its `ref` field to the elaborated refinement term. This mutates the proof term in place. If elaboration fails, an error is thrown, and the proof state remains unchanged.

- `intro(proofTerm, holeId, varName?)`: This tactic applies to a goal whose type is a Pi-type, e.g., `⊢ Π (x : A), B`. It refines the goal hole with a lambda abstraction `λ (x`

`: A). ?h_body`, where `?h_body` is a fresh hole. The `refine` mechanism then automatically creates a new subgoal for `?h_body` with type `B` in a context extended with `x : A`.

- `exact(proofTerm, holeId, solutionTerm)`: This is a direct wrapper around `refine`. It is used to solve a goal completely with a given term. The `elaborate` call within `refine` ensures the term is complete and has the correct type.

- `apply(proofTerm, holeId, funcTerm)`: This tactic applies a function to the goal. For a goal `⊢ T`, applying a function `f : A → B → T` refines the goal hole with the term `f {?h₁} ... (?hₙ)`, where `?hᵢ` are fresh holes created for each of `f`'s arguments. This solves the current goal and generates new subgoals for each of the arguments.

Example Proof Session. To prove the identity function on `Nat`, a user would proceed as follows:

1. **State Goal:** `defineGlobal("id_nat_proof", Pi("n", Expl, Nat, _ => Nat), Hole("?g0"))`.
2. **Inspect:** `reportProofState(Var("id_nat_proof"))` shows `Goal ?g0: ⊢ Π (n : Nat). Nat`.
3. **Introduce:** `intro(Var("id_nat_proof"), "?g0", "n")`. The hole `?g0` is solved with `λ n. ?g1`.
4. **Inspect:** The report now shows `Goal ?g1: n : Nat ⊢ Nat`.
5. **Solve:** `exact(Var("id_nat_proof"), "?g1", Var("n"))`. The hole `?g1` is solved with `n`.
6. **Complete:** `findHoles` now returns an empty list. The final proof term is `λ (n : Nat). n`.

This interactive workflow, built on the same elaboration engine as the core type checker, provides a powerful and consistent environment for both defining and proving.

7. From Theory to Practice: Validation and Testing

The Emdash system is not merely a theoretical design; it is a working implementation whose correctness and robustness have been verified through a comprehensive suite of tests. This section highlights how our testing strategy validates the core claims of the paper.

7.1. System Architecture

The TypeScript implementation is organized into a set of loosely coupled modules, each with a clear responsibility. This modularity was crucial for managing the complexity of the system and for enabling targeted testing. The use of TypeScript's static typing was invaluable in maintaining consistency across the different parts of the codebase, from term definitions in `types.ts` to the logic in `unification.ts` and `elaboration.ts`.

7.2. Validation Through Testing

The correctness of Emdash is demonstrated by an extensive test suite located in the `tests/` directory. These tests are not mere unit tests of isolated functions; they are integration tests that exercise the entire elaboration pipeline, from raw term construction to final type and value verification. All tests currently pass successfully.

- **Validating the Foundations: Inductive and Dependent Types:**
    - The `inductive_types.ts` suite validates the definition and use of `Nat`, `Bool`, and polymorphic `List`. It confirms that functions like `add` and `map` can be defined both via a primitive eliminator with rewrite rules and via direct rewrite rules on the function symbol, demonstrating the flexibility of the equational reasoning system.
    - The `dependent_types_tests.ts` suite uses length-indexed vectors (`Vec A n`) to stress-test the dependent type checker, confirming that terms like `vcons {Bool} {z} true (vnil {Bool})` are correctly elaborated to have type `Vec Bool (s z)`.
    - The `equality_inductive_type_family.ts` suite successfully defines propositional equality (`Eq`, `refl`) and proves fundamental properties like symmetry (`symm`) and transitivity (`trans`), both with a J-like eliminator and with rewrite rules. This demonstrates that the framework is powerful enough to encode and reason about its own meta-theory.

- **Stress-Testing the Elaboration and Unification Engine:**
    - The `implicit_args_tests.ts` and `church_encoding_implicit_tests.ts` suites verify numerous scenarios of implicit argument insertion and inference, from

simple `const` and `id` functions to more complex higher-order applications and Church encodings.

  - `higher_order_unification_tests.ts` contains a battery of tests for the `solveHoFlexRigid` component, confirming that it correctly solves flex-rigid problems (e.g., `?M x = f x` ⇒ `?M = λy. f y`), handles multiple spine variables, and correctly fails the occurs check.
  - `higher_order_pattern_matching_tests.ts` validates the `matchPattern` function, including its ability to handle patterns with higher-order variables and, crucially, to respect scope restrictions on those variables (e.g., `$F.[x]`).

- **Confirming the Central Thesis: Functorial Elaboration and Proof Mode:**

  - The `functorial_elaboration.ts` suite provides the key validation for our definitional coherence-checking mechanism. It confirms that a correctly defined functor is successfully elaborated. More importantly, it asserts that a deliberately ill-defined functor (one whose `fmap1` violates the composition law) correctly causes the elaborator to throw a `CoherenceError`. The successful throwing of this error is not a failure of the test, but a confirmation of the system's design principle: coherence is a check, not a proof.
  - The `proof_mode_tests.ts` suite contains complete, successful proof constructions using the interactive API. It demonstrates that a sequence of calls to `intro`, `apply`, and `exact` correctly navigates the proof state, solves all subgoals, and produces a final, fully elaborated proof term that is definitionally correct.

The successful execution of this diverse and challenging test suite provides strong evidence for the correctness and stability of the Emdash framework and its novel features.

8. Related Work

Emdash situates itself within a rich landscape of logical frameworks and proof assistants, yet carves out a unique niche through its specific design choices and goals.

- **Emdash as a Logical Framework:** Emdash is a logical framework in the tradition of λΠ-calculus modulo theory systems like **Lambdapi** [3]. Its primary use case is the *synthetic* encoding of category theory. Unlike foundational systems where categories are built from sets or types, Emdash provides `Cat`, `Obj`, `Hom` as primitives. This approach aims for a more direct correspondence between mathematical practice and formal representation. Its extensibility via user-defined rewrite and unification rules allows it to be tailored to specific equational theories, a hallmark of a true logical framework. The use of **HOAS** for binders is another common technique in logical frameworks, simplifying the implementation by delegating α-conversion and substitution to the host language.

- **Relation to Functorial Programming:** Emdash is deeply inspired by Kosta Dosen's vision of "functorial programming" [1, 2]. This philosophy is reflected in several key design choices:
    - **Computational Laws:** Categorical laws are not just theorems to be proven but are implemented as rewrite rules, making them part of the system's definitional equality. Verification becomes computation.
    - **The Yoneda Principle:** The system begins to capture the computational essence of the Yoneda lemma. For instance, the covariant Hom functor is a primitive `HomCovFunctorIdentity`, and a unification rule `unif_hom_cov_fapp1_compose` directly equates its action `fapp1 (hom_cov W) a` with composition `compose_morph a`. This makes the action of the Yoneda embedding directly computational.

- **Comparison with Mainstream Proof Assistants:** Mature proof assistants like **Coq**, **Agda**, and **Lean** have extensive and powerful libraries for category theory. These are typically built *atop* a foundational type theory (CIC or MLTT). Emdash's synthetic approach offers a different perspective, aiming for more direct computational interpretation. The most significant differentiator is functorial elaboration. In Lean or Coq, proving a functor is a functor involves constructing a proof object for the functoriality laws. In Emdash, this proof is replaced by a definitional check within the elaborator itself. While currently lacking the vast libraries and automation of these systems, Emdash's approach offers a compelling alternative for domains where coherence is a central concern.

- **Relation to AI-Assisted Formalization:** The ultimate goal of Emdash is to power the hotdocX platform. There is a growing body of work on using LLMs for theorem proving [5, 6]. hotdocX and Emdash aim to integrate these capabilities from the ground up, focusing on the human-AI collaboration loop. The interactive proof mode is designed to be driven by either a human or an AI, and the structured nature of Emdash's terms and errors provides a clear interface for AI interaction.

9. Conclusion and Future Work

Emdash, in its current state, successfully realizes its initial goal: to establish a dependently typed logical framework with integrated primitives for synthetic category theory. Its core type theory, powered by a bidirectional type checker, unification-based hole solving, and a computationally-driven definitional equality, has been validated through a comprehensive suite of tests. The framework's key innovations—the concept of functorial elaboration for definitional coherence and a full-featured interactive proof mode—demonstrate its potential as both a theoretical tool and a practical engine for formalization.

The development of Emdash will proceed along several interconnected axes:

- **Deepening the Categorical Synthesis:** The immediate priority is to implement and test the remaining primitives from the standard library specification, moving towards a complete synthetic theory of (higher) categories. This includes:
    - **Higher Categories and Schemes:** The long-term goal is to realize the full vision of the Lambdapi specification, including primitives for ω-categories via dependent comma categories (`Total_cat`) and a computational interface for algebraic schemes. This involves implementing the dependent sum (`τΣ_`) and product types within the categorical setting and leveraging the `Total_cat_proj` functor to reason about fibrations.
    - **Advanced Categorical Structures:** Introducing primitives for profunctors, adjunctions, limits, and colimits, continuing the synthetic approach and leveraging functorial elaboration to handle their coherence laws definitionally.
    - **Univalence:** Formalizing and implementing the `isEquiv` primitive and the rules connecting it to the identity type (`=`), thereby realizing a univalent category of types within the framework, where `eq` is `iso`.

- **Framework Enhancements:**
    - **Universe Management:** Evolve from the `Type:Type` axiom to a predicative hierarchy of universes ($\texttt{Type}_0 : \texttt{Type}_1 : \ldots$) to soundly support constructions like a category of all small categories.
    - **Performance and Error Reporting:** Profile core algorithms like `whnf` and `unify` for performance bottlenecks and significantly improve the user-friendliness and

diagnostic detail of error messages from the elaborator.
    - **Tactic Language:** Expand the proof mode with a more expressive tactic language, allowing for the composition of primitive tactics into more powerful, automated proof strategies.

- **hotdocX Integration and AI Capabilities:**
    - Continue developing the pipeline for transforming mathematical documents into Emdash scripts, including the use of OCR and multimodal models to parse LaTeX, PDF, and images.
    - Refine AI models for suggesting definitions, types, lemmas, and rewrite rules based on informal input and the current Emdash context.
    - Develop tools for visualizing Emdash terms and computations, especially for categorical diagrams and rewrite sequences, to be embedded within the hotdocX platform.

Emdash, while still in its early stages, offers a unique and powerful combination of dependent type theory, synthetic category theory, and user extensibility. Its ongoing development, driven by the principles of functorial programming and the practical needs of AI-assisted formalization, aims to contribute a valuable new tool and perspective to the communities of logical frameworks, category theory, and computational mathematics.

References

[1] Dosen, K., & Petrić, Z. (1999). *Cut-Elimination in Categories*.

[2] Dosen, K., & Petrić, Z. (2004). *Proof-Theoretical Coherence*. KCL Publications.

[3] Blanqui, F. (Ongoing). *Lambdapi Logical Framework*. Deducteam. https://github.com/Deducteam/lambdapi

[4] The Univalent Foundations Program. (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study.

[5] Polu, S., et al. (2022). *Formal Mathematics Statement Curriculum Learning*. arXiv:2202.01344.

[6] OpenAI et al. (Various). Research on models like GPT-f for formal mathematics.

[7] hotdocX Project. https://hotdocx.github.io

[8] emdash Project. https://github.com/hotdocx/emdash

[9] 1337777. (2024). *Cartier Solution 16 (Schemes)*. https://github.com/1337777/cartier/blob/master/cartierSolution16.lp

[10] 1337777. (2024). *Cartier Solution 18 (Emdash Spec)*. https://github.com/1337777/cartier/blob/master/cartierSolution18.lp

[11] hotdocX. (2024). *jsCoq Template Example*. https://hotdocx.github.io/#/hdx/25191CHRI43000

[12] hotdocX. (2024). *Arrowgram AI Template Example*. https://hotdocx.github.io/#/hdx/25188CHRI26000

[13] hotdocX. (2024). *Emdash Re-formattable Example*. https://hotdocx.github.io/#/hdx/25188CHRI25004

[14] hotdocX. (2024). *Emdash Experiment-able Example*. https://hotdocx.github.io/#/hdx/25188CHRI27000

[15] Arrowgram Project. https://github.com/hotdocx/arrowgram

[16] hotdocX Sponsorship. https://github.com/sponsors/hotdocx

[17] Bertot, Y. (2019). *Coq Exchange*. https://project.inria.fr/coqexchange/news/

[18] Lamiaux, T. (2025). *Coq Platform Docs*. https://coq.inria.fr/docs/platform-docs

[19] 1337777. (2024). *Simplicial-cubical functorial programming*. https://cutt.cx/fTL

Appendix: Emdash Kernel Specification in Lambdapi (Selection)

This appendix contains a selection of the formal specification for Emdash's core categorical primitives, written in Lambdapi.

```
// ===== START OF BASIC KERNEL SPECIFICATION  =====

flag "eta_equality" on;

constant symbol Type : TYPE;
injective symbol τ : Type → TYPE;
builtin "T" ≔ τ;
builtin "Prop" ≔ Type;
builtin "P" ≔ τ;

// # --- Categories ---

constant symbol Cat : TYPE;

injective symbol Obj : Cat → TYPE;

constant symbol Obj_Type : Cat → Type;
rule τ (Obj_Type $A) ↪ Obj $A;

injective symbol Hom : Π [A : Cat] (X: Obj A) (Y: Obj A), TYPE;

constant symbol Hom_Type : Π [A : Cat] (X: Obj A) (Y: Obj A), Type;
rule τ (@Hom_Type $A $X $Y) ↪ @Hom $A $X $Y;


// # --- Functors ---

constant symbol Functor : Π(A : Cat), Π(B : Cat), TYPE;

constant symbol Functor_Type : Π(A : Cat), Π(B : Cat), Type;
rule τ (@Functor_Type $A $B) ↪ @Functor $A $B;

symbol fapp0 : Π[A : Cat], Π[B : Cat], Π(F : Functor A B), Obj A → Obj B;
symbol fapp1 : Π[A : Cat], Π[B : Cat], Π(F : Functor A B),
  Π[X: Obj A], Π[Y: Obj A], Π(a: Hom X Y), Hom (fapp0 F X) (fapp0 F Y);
```

```
// # --- Natural Transformations ---

constant symbol Transf : Π [A : Cat], Π [B : Cat], Π (F : Functor A B), Π (G : Fun

constant symbol Transf_Type : Π [A : Cat], Π [B : Cat], Π (F : Functor A B), Π (G
rule τ (@Transf_Type $A $B $F $G) ↪ @Transf $A $B $F $G;

symbol tapp : Π [A : Cat], Π [B : Cat], Π [F : Functor A B], Π [G : Functor A B],
  Π (ϵ : Transf F G), Π (X: Obj A), Hom (fapp0 F X) (fapp0 G X);


// # --- Functor category ---

constant symbol Functor_cat : Π(A : Cat), Π(B : Cat), Cat;

unif_rule Obj $Z ≡ (Functor $A $B) ↪ [ $Z ≡ (Functor_cat $A $B) ];

rule @Hom (Functor_cat _ _) $F $G ↪ Transf $F $G;


// # --- Set category classifier, discrete fibrations, yoneda representable Functo

constant symbol Set : Cat;

unif_rule Obj $A ≡ Type ↪ [ $A ≡ Set ];

rule @Hom Set $X $Y ↪ (τ $X → τ $Y);

constant symbol hom_cov : Π [A : Cat], Π (W: Obj A), Functor A Set;

rule fapp0 (hom_cov $X) $Y ↪ Hom_Type $X $Y;

// # --- Cat category classifier, fibrations, isofibrations, yoneda representable

constant symbol Cat_cat : Cat;
```

```
unif_rule Obj $A ≡ Cat ↪ [ $A ≡ Cat_cat ];

rule @Hom Cat_cat $X $Y ↪ Functor $X $Y;

constant symbol hom_cov_cat : Π [A : Cat], Π (W: Obj A), Functor A Cat_cat;

rule Obj (fapp0 (hom_cov_cat $W) $Y) ↪ Hom $W $Y;
rule fapp0 (fapp1 (hom_cov_cat $W) $a) $f ↪ (fapp1 (hom_cov $W) $a) $f;

// ===== END OF BASIC KERNEL SPECIFICATION =====
```