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

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

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.

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.

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 dissect the elaboration engine, explaining the interplay of bidirectional type checking, implicit argument handling, and unification-based constraint solving.
- Section 4 is dedicated to our central contribution, functorial elaboration, providing a deep dive into the `MkFunctorTerm` primitive and its coherence-checking mechanism.
- In Section 5, we describe the system's full-featured interactive proof mode, demonstrating how users can construct proofs by refining holes with a suite of tactics.

- Section 6 provides a crucial overview of the system's validation through its comprehensive test suite, confirming the correct behavior of all major features.
- Finally, in Sections 7 and 8, 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 `Hole`s via the `getTermRef` utility.

2.2. Synthetic Categorical Primitives

Emdash provides categorical structures as first-class citizens, closely mirroring its Lambdapi specification. These are not defined in terms of other structures but are primitive term constructors. The `stdlib.ts` file initializes the system with these primitives.

- `CatTerm()`: The type of categories. This corresponds to `constant symbol Cat : TYPE;` in the Lambdapi specification. It is defined as a global constant via `defineGlobal("Cat", Type(), CatTerm(), true)`.
- `ObjTerm(cat)`: The type of objects in a category `cat`. This corresponds to the injective symbol `Obj : Cat → TYPE;`. Its global definition has the dependent type `Π (A : Cat), Type` and a value that is a lambda wrapping the `ObjTerm` constructor.
- `HomTerm(cat, dom, cod)`: The type of morphisms between objects `dom` and `cod` in category `cat`. Its global definition has the type `Π {A:Cat} (X:Obj A) (Y:Obj A), Type`, making the category an

implicit argument.

- `compose_morph` : A global symbol for morphism composition, with the expected dependent type: `Π {A:Cat} {X,Y,Z:Obj A}. Hom Y Z → Hom X Y → Hom X Z` .
- `identity_morph` : A global constant symbol for the identity morphism: `Π {A:Cat} (X:Obj A), Hom X X` .
- `FunctorTypeTerm(domain, codomain)` : The primitive type for functors.
- `FMap0Term(functor, objectX)` / `FMap1Term(functor, morphism_a)` : Primitive terms representing functor application to objects and morphisms, respectively. These constructors include optional implicit fields for their domain and codomain categories, which are handled by the elaboration engine.

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. The Elaboration Engine: From Raw Terms to Typed Expressions

The heart of Emdash is its elaboration engine, which transforms raw, potentially ambiguous user input into fully typed, unambiguous terms. This process is driven by a bidirectional type checking algorithm and a powerful unification-based constraint solver.

3.1. Bidirectional Type Checking: The `infer` and `check` Dance

Elaboration is orchestrated by the `elaborate(term, expectedType?)` function, which dispatches to one of two modes:

- **Inference Mode (`infer(Γ, t)`):** When no type is expected, `infer` attempts to synthesize the type of a term `t` . It returns both the elaborated term `t'` and its inferred type `T` .
- **Checking Mode (`check(Γ, t, T)`):** When a type `T` is expected, `check` verifies that `t` is a valid inhabitant of `T` . It returns only the elaborated term `t'` , as the type is already known.

This bidirectional approach is highly effective. For example, when inferring the type of an unannotated lambda `λx. e` , the system cannot know the type of `x` . It therefore creates a fresh hole `?A` for its type, extends the context with `x : ?A` , infers the type `B` of the body `e` , and returns the Pi-type `Π (x : ?A), B` as the type of the lambda. Conversely, when checking `λx. e` against an expected type `Π (y : A), B` , the system can use `A` as the type for `x` , extend the context with `x : A` , and recursively check `e` against `B` . The main typing rules are summarized in Figure 1.

> **Figure 1: Key Bidirectional Typing Rules**
>
> ```
> ------ (Type-Inf)
> Γ ⊢ Type ⇒ Type
>
> Γ, x:A ⊢ B ⇒ Type
> ---------------- (Pi-Inf)
> Γ ⊢ Π(x:A),B ⇒ Type
>
> Γ ⊢ A ⇒ Type     Γ, x:A ⊢ t ⇒ B
> ----------------------------------- (Lam-Inf, unannotated)
> Γ ⊢ λx.t ⇒ Π(x:?A),B'
> (where ?A is a fresh hole, and B' is the inferred type of t in Γ, x:?A)
>
> Γ ⊢ f ⇒ Π(x:A),B     Γ ⊢ a ⇐ A
> ---------------------------------- (App-Inf)
> Γ ⊢ f a ⇒ B[a/x]
>
> Γ ⊢ t ⇒ T'     T' ≡ T
> ------------------------ (Check-from-Inf)
> Γ ⊢ t ⇐ T
>
> Γ ⊢ A ⇐ Type     Γ, x:A ⊢ B ⇐ Type
> ------------------------------------- (Pi-Chk)
> Γ ⊢ Π(x:A),B ⇐ Type
>
> Γ, x:A ⊢ e ⇐ B
> ------------------- (Lam-Chk)
> Γ ⊢ λx.e ⇐ Π(x:A),B
> ```

3.2. Implicit Argument Handling

A key aspect of usability in a dependently typed language is the handling of implicit arguments. Emdash employs two mechanisms for this:

1. **Implicit Application Insertion (`insertImplicitApps`):** When the elaborator is in `infer` mode and encounters an application `f a` where `f` has an implicit Pi-type `Π {x:A}. B` , it knows an implicit argument is missing. The `insertImplicitApps` function is called, which transforms the term into `(f {?h}) a` , where `?h` is a fresh hole. The type of this new term is `B[?h/x]` , and elaboration continues. This process repeats until the function's type is no longer an implicit Pi-type.

2. **Implicit Lambda Insertion (Eta-Expansion):** When in `check` mode, if the expected type is an implicit Pi-type `Π {x:A}. B` but the term `t` being checked is not an implicit lambda, the elaborator wraps `t` in one. It effectively transforms the goal `Γ ⊢ t ⇐ Π {x:A}. B` into a new goal `Γ, x:A ⊢ t ⇐ B` . The final elaborated term is `λ {x:A}. t'` .

3. **Kernel Implicit Handling (`ensureKernelImplicitsPresent`):** Certain primitive term constructors, like `FMap0Term` or `FMap1Term` , have implicit arguments that are essential for their well-formedness (e.g., the domain and codomain categories). The `constants.ts` file contains a specification `KERNEL_IMPLICIT_SPECS` listing these. Before elaboration begins, `ensureKernelImplicitsPresent` traverses the raw term and inserts fresh holes for any of these specified implicit fields that are undefined. This allows for more concise user input, as these arguments are almost always inferable from the context.

3.3. Unification and Constraint Solving

The elegance of the bidirectional algorithm rests on a powerful unification engine to solve the constraints generated during elaboration.

Constraint Generation. Whenever the elaborator requires two terms to be equal, it does not fail immediately. Instead, it adds a `Constraint` object `{ t1, t2, origin }` to a global list. This happens, for example, when `check(Γ, t, T)` finishes by inferring that `t` has type `T'` , leading to the constraint `T' ≡ T` .

Constraint Solving (`solveConstraints`). After the main elaboration traversal, `solveConstraints` is called. It operates in a loop, iteratively attempting to solve the constraints until no more progress can be made:

1. For each constraint $t_1 \equiv t_2$, it first checks if the terms are already definitionally equal via `areEqual` . If so, the constraint is removed.
2. If not, it calls `unify(t₁, t₂, Γ)` .
3. `unify` returns a `UnifyResult` :
   - `Solved` : The constraint is considered solved and removed.

    - `Progress` : A hole was instantiated but the terms may not be fully equal yet. The constraint is kept for the next iteration.
    - `RewrittenByRule` : A user-defined unification rule was applied, replacing the current constraint with new ones.
    - `Failed` : Unification is impossible. `solveConstraints` halts and reports failure.
4. The loop continues as long as the set of constraints shrinks or `Progress` is reported.

The `unify` Algorithm. The `unify` function, located in `unification.ts` , is the core of the solver. Its strategy is as follows:

1. **Hole Handling:** If one term is a hole, `unifyHole` is called. It performs an `occurs check` (`termContainsHole`) to prevent circular definitions (e.g., `?h := f(?h)`) and then solves the hole by setting its `ref` field.
2. **Higher-Order Unification:** If one term is a flexible-rigid pair (e.g., `?M x y ≡ g x y` , where `x` and `y` are bound variables), it calls `solveHoFlexRigid` . This function implements Miller's pattern unification. It validates that the spine of the flexible term consists of distinct bound variables, then solves for the hole by abstracting the rigid term over that spine (e.g., `?M := λa b. g a b`).
3. **Structural Decomposition:** If the terms have the same structurally injective head (e.g., they are both `ObjTerm` or both applications of an injective global `f_inj`), `unify` recursively unifies their arguments.
4. **Fallback to Reduction:** If all else fails, the terms are reduced to WHNF and the unification process is retried. If this still fails, the unification attempt is deemed impossible.

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.

4. Functorial Elaboration: Definitional Coherence

A central and novel contribution of Emdash is its mechanism for enforcing coherence laws as part of definitional equality. Traditionally, when defining an algebraic structure like a category or a functor, one provides the components (objects, morphisms, composition, etc.) and then separately proves that these components satisfy the required laws (associativity, identity, functoriality). In Emdash, these laws can be checked *during elaboration*, making them a definitional property of the term itself.

This is achieved through specialized kernel term constructors. The primary example is `MkFunctorTerm` .

The `MkFunctorTerm` Primitive. While a user might interact with a high-level helper function like `mkFunctor` , the core of the system uses the primitive `MkFunctorTerm(domainCat, codomainCat, fmap0, fmap1, proof?)` . This term is not just a container for data; it is an instruction to the elaborator to perform a coherence check.

The `infer_mkFunctor` Process. When the elaborator's `infer` function encounters a `MkFunctorTerm`, it triggers a special procedure, `infer_mkFunctor`, detailed in `elaboration.ts`:

1. **Component Elaboration:** First, it elaborates the components. It checks that `domainCat` and `codomainCat` are of type `CatTerm()`. It then checks `fmap0` against the expected object-map type `Π (x: Obj domainCat), Obj codomainCat`, and `fmap1` against the expected morphism-map type `Π {X Y} (a: Hom X Y), Hom (fmap0 X) (fmap0 Y)`.

2. **Law Synthesis:** If no explicit proof term is provided and coherence checking is not skipped, the engine synthesizes the two sides of the functoriality law for composition. It creates a fresh context `lawCtx` containing generic objects `X, Y, Z` and morphisms `f : Hom X Y`, `g : Hom Y Z`. In this context, it constructs the two terms to be compared:

   - `LHS := compose_morph {B} (fmap1 g) (fmap1 f)`
   - `RHS := fmap1 (compose_morph {A} g f)`

3. **Computational Verification:** The core of the check lies in the next step. The system computes the normal forms of both `LHS` and `RHS` within `lawCtx`:

   - `normLhs := normalize(lhs, lawCtx)`
   - `normRhs := normalize(rhs, lawCtx)` This normalization process will unfold the definitions of `fmap1` and `compose_morph` and apply any relevant rewrite rules.

4. **Equality Assertion:** It then asserts their equality using `areEqual(normLhs, normRhs, lawCtx)`.

5. **Error or Success:** If `areEqual` returns `false`, the elaboration process is aborted, and a `CoherenceError` is thrown. This error is specific and informative, indicating which law failed and printing the non-equal normal forms. If the check passes, the `infer` function successfully returns the elaborated `MkFunctorTerm` with its type, `FunctorTypeTerm(domainCat, codomainCat)`.

This "functorial elaboration" mechanism is a powerful design pattern. It shifts the burden of proving structural integrity from the user (who would otherwise need to provide a separate proof term) 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.

5. 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.

6. Implementation and Validation

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.

6.1. System Architecture

The TypeScript implementation is organized into a set of loosely coupled modules, each with a clear responsibility, as detailed in Section 3.1. 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`.

6.2. 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.

- **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.

- **Elaboration and Unification Engine:**
    - The `implicit_args_tests.ts` suite verifies numerous scenarios of implicit argument insertion and inference, from simple `const` and `id` functions to more complex higher-order applications.
    - `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]`).

- **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`, as expected.
    - 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.

7. Discussion and 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.

8. 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:

- **Phase 2 Implementation and Library Expansion:** The immediate priority is to implement and test the remaining primitives from the standard library specification. This includes:

- **Natural Transformations:** Fully integrating `NatTransTypeTerm` and `NatTransComponentTerm` and validating the naturality square rewrite rules.
- **The `Set` Category:** Finalizing the implementation of `SetTerm()` as a primitive category where `Obj Set` is `Type` and `Hom {Set} X Y` is `Π(x:X), Y`, and verifying all associated rewrite rules.
- **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.
- **Dependent Category Theory:** Begin implementing the structures necessary for fibrations and dependent functors, guided by the more abstract parts of the Lambdapi specification (`Context_cat`, `Sigma_catd`, etc.).

- **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.
  - 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

Appendix: Formal Typing Rules (Selection)

This appendix provides a more formal summary of the key judgments in Emdash's bidirectional type system.

Contexts and Judgments:

- A context `Γ` is a list of bindings `x:T` or `x:T=t`.
- `Γ ⊢ t ⇒ T`: In context `Γ`, term `t` is inferred to have type `T`.
- `Γ ⊢ t ⇐ T`: In context `Γ`, term `t` is checked to have type `T`.
- `Γ ⊢ t ≡ u : T`: In context `Γ`, terms `t` and `u` are definitionally equal at type `T`.

Core Rules:

- **Formation:**

  ```
  ----------- (Type-Form)
  Γ ⊢ Type ⇒ Type

  Γ ⊢ A ⇐ Type     Γ, x:A ⊢ B ⇐ Type
  ------------------------------------ (Pi-Form)
  Γ ⊢ Π(x:A), B ⇒ Type
  ```

- **Inference (⇒):**

  ```
  x:T ∈ Γ
  ----------- (Var-Inf)
  Γ ⊢ x ⇒ T

  Γ ⊢ f ⇒ Π(x:A),B     Γ ⊢ a ⇐ A
  ---------------------------------- (App-Inf)
  Γ ⊢ f a ⇒ B[a/x]

  Γ ⊢ A ⇐ Type     Γ, x:A ⊢ t ⇒ B
  ```

```
---------------------------------- (Lam-Inf)
Γ ⊢ λ(x:A).t ⇒ Π(x:A),B
```

- **Checking (⇐):**

```
Γ ⊢ t ⇒ T'      Γ ⊢ T' ≡ T : Type
---------------------------------- (Switch)
Γ ⊢ t ⇐ T

Γ, x:A ⊢ t ⇐ B
------------------------ (Lam-Chk)
Γ ⊢ λx.t ⇐ Π(x:A),B

Expected: Π{x:A}.B    Γ,x:A ⊢ t ⇐ B
-------------------------------------- (Implicit-Lam-Ins)
Γ ⊢ t ⇐ Π{x:A}.B
(where the elaborated term becomes λ{x:A}.t')
```

- **Equality:**

```
Γ ⊢ t₁ ⇒ T      Γ ⊢ t₂ ⇒ T      whnf(t₁) ≡ₛ whnf(t₂)
------------------------------------------------ (Eq)
Γ ⊢ t₁ ≡ t₂ : T
(where ≡ₛ denotes recursive structural equality)
```