

# Morpheus: A Categorical Programming Language for Deep Learning

Jovonni L. Pharr  
Georgia Cyber Warfare Range    Blackmind  
GCPU Project

October 2026

## Abstract

We present MORPHEUS, a compiled, statically typed programming language in which category theory *is* the type system and the type system *is* the architecture-constraint system. Motivated by the thesis of Gavranović et al. [1] that neural network layers are (lax) algebra homomorphisms, Morpheus gives every categorical construct—object, morphism, functor, natural transformation, (co)monad, (co)algebra, adjunction, (co)limit, pullback—a concrete language keyword, and every keyword a precise deep-learning meaning. A Morpheus program is a commutative diagram: objects are tensor spaces, morphisms are (parametric) layers in **Para(Vect)**, composition is the categorical composition operator `;`, and an algebra homomorphism annotation (`homomorphism over`) asks the compiler to verify the equivariance diagram  $h \circ \alpha = \beta \circ F(h)$ . The compiler, written in Rust, lexes and parses the surface syntax into a categorical AST, type-checks composition compatibility object-by-object, lowers to a Categorical Intermediate Representation (CIR), optimises the diagram, and emits PyTorch modules, with an optional Lean 4 backend that discharges the categorical proof obligations through a custom `morpheus_verify` tactic. We describe the design through the language’s real example programs—a linear “hello world,” morphisms and composition, functors and natural transformations, monads, the algebra-homomorphism transformer block, residual connections as pullbacks, attention as a limit and pooling as a colimit, an image classifier over the grid endofunctor, a graph attention network, a variational autoencoder in the Kleisli category of a Gaussian monad, an encoder–decoder adjunction, and type-directed architecture search—and ground each in the actual compiler source. Morpheus’s central claim is that every program that type-checks is a structurally valid architecture, turning a large class of architectural mistakes from silent runtime bugs into compile-time theorems.

## 1 Introduction

Modern deep-learning practice is, as the Morpheus specification bluntly puts it, closer to alchemy than to engineering: practitioners stack layers by intuition, tune hyperparameters by trial and error, and have no formal guarantee that an architecture respects the structure of its data. Tensor frameworks such as PyTorch [9] hand the practitioner raw tensors and report shape errors only at runtime. There is no notion, at the level of the program, of whether a composition of layers is *structurally* meaningful—whether an attention block is a legitimate way to process set-structured data, or whether a pooling operation commutes with the layers that precede it.

A recent line of work in applied category theory suggests that this missing structure already exists, mathematically, and only needs a language. Fong, Spivak and Tuyéras showed that backpropagation is functorial [3]; Cruttwell et al. gave categorical foundations of gradient-based learning in terms of lenses and parametric maps [2]; Spivak framed learners as polynomial constructions [6]; and Gavranović, Lessard, Dudzik, von Glehn, Araújo and Veličković [1] argued that *categorical deep learning is an algebraic theory of all architectures*: a neural network layer

that respects the structure of its input data is exactly a (lax) algebra homomorphism for the endofunctor that generates that structure. Crescenzi’s survey [5] collects the broader program. The slogan of the field, attributed in the Morpheus notes to John Baez, is that “every sufficiently good analogy is yearning to become a functor.”

The thesis of this paper, and of the system it describes, is that this analogy should be realised not as a library but as a *programming language*. If layers are algebra homomorphisms, then the commutativity diagram  $h \circ \alpha = \beta \circ F(h)$ —which in the vector-space category **Vect** unrolls into exactly the familiar equivariance equation—is a type-level constraint, and a type checker can decide (or, with a theorem prover, prove) whether a proposed layer satisfies it. The history of mathematics, as the project’s design notes repeatedly observe, teaches that pushing semantic work into syntax is what makes automation possible. Morpheus pushes the semantics of architecture design into a type system built directly on category theory.

**Contributions.** This paper documents the design and implementation of Morpheus, the first programming language dedicated to deep learning and built strictly through the lens of academic category theory. Concretely:

- A surface language in which each category-theoretic construct is a first-class keyword (Section 3), illustrated throughout with the language’s real `.morph` example programs.
- A categorical type system (Section 4) whose core judgement is composition compatibility: the output object of  $f$  must match the input object of  $g$  before  $f ; g$  type-checks, with parametric morphisms tracked in **Para(Vect)**.
- A compiler pipeline—lexer, parser, type checker, Categorical IR, PyTorch code generator, and an optional Lean 4 verification backend—whose real Rust source we quote.
- Worked examples spanning sequences, grids, graphs, sets, and products: an image classifier, a graph attention network, a VAE, an encoder–decoder adjunction, and a type-directed architecture search.

We emphasise that Morpheus is a research system under active development; Section 6 is candid about what is implemented, what is specified, and where the sharp edges are (notably, the honest observation that self-attention is *not* a strict fold homomorphism, and that the type checker should reject annotating it as one).

## 2 Background: Category Theory for Deep Learning

We recall the categorical vocabulary that Morpheus elevates to language keywords, tying each concept to its deep-learning meaning. The treatment follows Mac Lane [7] for the mathematics and Gavranović et al. [1] and Fong–Spivak [4] for the learning interpretation.

**Categories and objects.** A *category*  $\mathcal{C}$  is a universe of a particular kind of thing: a collection of objects, a collection of morphisms (arrows) between them, an associative composition, and an identity arrow at each object. In Morpheus the built-in categories are **Vect** (finite-dimensional real vector spaces, objects are dimensions, morphisms are linear maps) and **Set** (finite sets, morphisms are functions); the compiler additionally knows the names **Aff**, **Stoch** and **Meas**. An *object* is a tensor space: a shape together with the category it lives in. Discrete vocabularies live in **Set** as `Discrete( $n$ )`; continuous feature spaces live in **Vect** as `Vector( $d$ )`.

**Morphisms and composition.** A *morphism*  $f : A \rightarrow B$  is a pure function between spaces (an activation, a reshape, a linear map). Composition is the fundamental operation: if  $f : A \rightarrow B$  and  $g : B \rightarrow C$  then  $f ; g : A \rightarrow C$ , written in diagrammatic (left-to-right) order. Building a neural network *is* composing morphisms, and the only non-trivial side condition is that the intermediate object  $B$  matches:

$$\begin{array}{ccccc} A & \xrightarrow{f} & B & \xrightarrow{g} & C \\ & \searrow & & \nearrow & \\ & & f ; g & & \end{array}$$

**Parametric morphisms,  $\mathbf{Para}(\mathcal{C})$ .** A learnable layer is not merely a morphism but a *parametric* morphism: a choice of parameter space  $P$  together with a map  $P \times A \rightarrow B$ . These are the morphisms of the category  $\mathbf{Para}(\mathcal{C})$ . Crucially, composition tensors the parameter spaces:  $(Q, g) \circ (P, f) = (P \times Q, x \mapsto g(q, f(p, x)))$ . Because parameters are explicit, one can reason about weight sharing and tying as *2-morphisms* (reparameterisations) between parametric maps. This is the construction that lets Morpheus talk about an entire network’s parameters, not just one layer’s, and that [2, 1] identify as the right setting for gradient-based learning.

**Functors and natural transformations.** A *functor*  $F : \mathcal{C} \rightarrow \mathcal{D}$  maps objects to objects and morphisms to morphisms while preserving identities and composition:  $F(\text{id}) = \text{id}$  and  $F(f ; g) = F(f) ; F(g)$ . An embedding  $\mathbf{Set} \rightarrow \mathbf{Vect}$  (discrete tokens to continuous vectors) is the canonical example in ML. An *endofunctor*  $F : \mathcal{C} \rightarrow \mathcal{C}$  is a data-structure type: the list/sequence endofunctor, the grid endofunctor, the graph endofunctor, the multiset (set) endofunctor. A *natural transformation*  $\eta : F \Rightarrow G$  is a family of morphisms  $\eta_A : F(A) \rightarrow G(A)$  making the naturality square commute for every  $f : A \rightarrow B$ :

$$\begin{array}{ccc} F(A) & \xrightarrow{F(f)} & F(B) \\ \eta_A \downarrow & & \downarrow \eta_B \\ G(A) & \xrightarrow{G(f)} & G(B) \end{array}$$

Mean pooling is a natural transformation from the sequence endofunctor to the identity functor: it is a single, polymorphic “reduce” that commutes with whatever element-wise processing precedes it.

**(Co)monads and (co)algebras.** A *monad*  $(T, \eta, \mu)$  is an endofunctor with a unit  $\eta$  and a multiplication  $\mu$  obeying coherence laws; the group-action monad  $G \times (-)$  is the foundation of geometric deep learning, and its algebras are exactly group representations. More generally, an *F-algebra* is a carrier object  $A$  with a structure map  $\alpha : F(A) \rightarrow A$  that executes one step of computation—a fold. The *dual*, an *F-coalgebra*  $A \rightarrow F(A)$ , models productive/infinite computation such as automata and streams. The structure map is how a list is reduced (last-element, mean, sum) or how a grid is pooled.

**The key insight: layers are algebra homomorphisms.** Given two  $F$ -algebras  $(A, \alpha)$  and  $(B, \beta)$ , an *algebra homomorphism* is a morphism  $h : A \rightarrow B$  such that the following square commutes:

$$\begin{array}{ccc} F(A) & \xrightarrow{F(h)} & F(B) \\ \alpha \downarrow & & \downarrow \beta \\ A & \xrightarrow{h} & B \end{array} \quad \text{i.e.} \quad h \circ \alpha = \beta \circ F(h).$$

This is the thesis of [1]. When the diagram is grounded in  $\mathbf{Vect}$ , so that  $h$  is a linear map  $W$ , the equation  $h \circ \alpha = \beta \circ F(h)$  becomes a weight-sharing constraint on  $W$ : for the translation

group it forces  $W$  to be a circulant matrix—i.e. a convolution. “Processing then folding equals folding then processing” is the equivariance of the layer. Weakening strict commutativity to commutativity-up-to-a-2-morphism (the *lax* case) is what recovers weight tying in recurrent, recursive, and automaton-style networks, and is the case the position paper focuses on.

**Adjunctions and (co)limits.** An *adjunction*  $F \dashv G$  (encoder  $\dashv$  decoder) is a duality with unit  $\eta : \text{Id} \Rightarrow G \circ F$  and counit  $\varepsilon : F \circ G \Rightarrow \text{Id}$  obeying the triangle identities; it is the mathematical skeleton of autoencoders, with reconstruction loss minimising the gap between  $G \circ F$  and the identity. A *limit* is a universal cone over a diagram and a *colimit* a universal cocone; Morpheus reads attention as a limit (the universal query-weighted aggregation of key-value pairs) and pooling as a colimit. A *pullback* is the universal fork-then-merge, which is exactly the shape of a residual connection.

### 3 Language Design

Morpheus is designed to feel like writing mathematics in a scripting language: readable to anyone with basic category theory, concise (composition is a single operator), safe (structural errors are compile errors), and gradual (one can start with plain compositions and add categorical annotations as needed). The AST in the compiler makes the one-to-one correspondence explicit; each variant of the top-level `Statement` enum is a category-theoretic construct:

```
pub enum Statement {
  CategoryDecl(CategoryDecl), // declaring a category (universe)
  TypeDecl(TypeDecl),        // an object in a category
  MorphismDef(MorphismDef),    // a pure morphism (arrow)
  ParametricDef(ParametricDef), // a parametric morphism in Para(C)
  FunctorDef(FunctorDef),      // structure-preserving map
  EndofunctorDef(EndofunctorDef), // functor from C to itself
  NaturalDef(NaturalDef),      // natural transformation
  MonadDef(MonadDef),          // endofunctor + unit + multiplication
  ComonadDef(ComonadDef),      // endofunctor + extract + duplicate
  AlgebraDef(AlgebraDef),      // F-algebra (carrier + structure map)
  CoalgebraDef(CoalgebraDef),  // F-coalgebra (dual of algebra)
  AdjunctionDef(AdjunctionDef), // F  $\dashv$  G (encoder-decoder duality)
  PullbackDef(PullbackDef),    // pullback (residual connections)
  LimitDef(LimitDef),          // limit (universal cone -- attention)
  ColimitDef(ColimitDef),      // colimit (universal cocone -- pooling)
  SearchDef(SearchDef),        // architecture search primitive
  // ... (isomorphism, loss, model, share, reparam, check, prove, use, ...)
}
```

#### 3.1 Objects: types in a category

The simplest Morpheus program declares a category, two objects, and a single parametric arrow between them. This is `00_hello_world.morph`:

```
-- 00_hello_world.morph -- The simplest possible Morpheus program
--
-- A single linear map: one parametric morphism from Vector(4) to Vector(2).
-- This is one arrow in the category Para(Vect).

category Vect

type Input  : Vect = Vector(4)
```

```

type Output : Vect = Vector(2)

let hello_world = Linear(4, 2)

-- Categorical structure: a single parametric arrow in Para(Vect)
-- Linear(4,2) : Vector(4) -> Vector(2) is the simplest non-trivial morphism
-- The identity endofunctor on Vect witnesses that this arrow is well-typed
endofunctor Identity on Vect

```

A **type** declaration names an object and annotates the category it inhabits. Objects may live in different categories, and categories may be bridged by functors. The example `01_category_and_objects.mor` introduces both **Vect** and **Set**, the terminal and initial objects, and an embedding functor that maps discrete tokens to vectors:

```

category Vect
category Set

type Hidden : Vect = Vector(256)
type Feature : Vect = Vector(64)
type Token : Set = Discrete(32000)
type Label : Set = Discrete(10)
type Pixel : Vect = Tensor(3, 224, 224)

terminal UnitType
initial VoidType

-- Set and Vect are linked by an embedding functor
functor TokenEmbedding : Set -> Vect {
  map_object(A) = embed_object(A)
  map_morphism(f) = linear_extension(f)
}
check laws(TokenEmbedding)

```

Internally an object is an `ObjectType`, and the type checker's notion of *composition compatibility* is a method on it. The rule is deliberately simple and structural: two objects are compatible when they are the same vector dimension (or the same discrete cardinality), or when at least one is a polymorphic placeholder:

```

pub fn compatible(&self, other: &ObjectType) -> bool {
  match (self, other) {
    (ObjectType::Vector(a), ObjectType::Vector(b)) => a == b,
    (ObjectType::Discrete(a), ObjectType::Discrete(b)) => a == b,
    (ObjectType::Poly(_), _) | (_, ObjectType::Poly(_)) => true,
    (a, b) => a == b,
  }
}

```

The polymorphic case is what lets built-in activations (which are endomorphisms on any dimension  $N$ ) slot into a composition without the programmer re-stating the dimension.

## 3.2 Morphisms and composition

A *pure* morphism has no parameters. The example `02_morphisms.morph` defines three endomorphisms on `Vector(256)` and the identity:

```

category Vect

morphism rectified_linear : Vector(256) -> Vector(256) {

```

```

    max(x, 0)
}

morphism sigmoid_activation : Vector(256) -> Vector(256) {
  sigmoid(x)
}

morphism normalize : Vector(256) -> Vector(256) {
  layer_normalization(x)
}

let skip = identity(Vector(256))

```

Composition uses the `; operator` in diagrammatic (left-to-right) order, as in `03_composition.morph`:

```

category Vect

-- Inline composition with ; (diagrammatic order: left to right)
let two_layer = Linear(256, 512) ; gaussian_error_linear ; Linear(512, 256)

-- Named intermediate compositions
let projection = Linear(256, 128)
let expansion  = Linear(128, 512)
let pipeline   = projection ; gaussian_error_linear ; expansion

```

Here `two_layer` type-checks because `Vector(512)` (output of the first `Linear`) matches the polymorphic input of the activation and then the input of `Linear(512, 256)`. Associativity  $(f ; g) ; h = f ; (g ; h)$  is a category axiom and need not be checked. The result type of a composition is itself a morphism whose source is the first operand's source, target is the last operand's target, and parameter list is the concatenation of the operands'—mirroring the tensoring of parameter spaces in `Para(Vect)`.

### 3.3 Endofunctors: data-structure types

An `endofunctor` declares *what kind* of data a model processes. The example `06_endofunctor.morph` declares the sequence, grid, and molecular-graph endofunctors, each with its object- and morphism-maps, and then a parametric morphism annotated to respect the sequence structure:

```

category Vect

-- Sequence endofunctor: processes ordered data
endofunctor Sequence(512) on Vect {
  map_object(A) = Array(512, A)
  map_morphism(f) = map(f)
}

-- Grid endofunctor: processes 2D spatial data (images)
endofunctor Grid(224) on Vect {
  map_object(A) = Tensor(224, A)
  map_morphism(f) = map_spatial(f)
}

-- Graph endofunctor: processes graph-structured data
endofunctor MolecularGraph(64) on Vect {
  map_object(A) = graph_object(64, A)
  map_morphism(f) = graph_map(f)
}

```

```

-- equivariant projection preserves Sequence structure
parametric EquivariantProjection(dimension: Natural)
  : Vector(512) -> Vector(512)
  homomorphism over Sequence
{
  Linear(512, 512)
}

```

The functor’s action on morphisms— $\text{map}(f)$  for sequences,  $\text{map\_spatial}(f)$  for grids,  $\text{graph\_map}(f)$  for graphs—is precisely the  $F(f)$  in the homomorphism square. Different ML problems differ *only* in their endofunctor declaration; the grammar is otherwise unchanged across language modelling (sequence), image classification (grid), molecular property prediction (graph), point-cloud segmentation (multiset), and multimodal fusion (product of endofunctors).

### 3.4 Algebras and algebra homomorphisms: the layer

An **algebra** gives the structure map that folds a data structure to a carrier. An algebra homomorphism is a parametric morphism carrying the **homomorphism** over  $F$  annotation, which is the heart of the language. The example `12_algebra_homomorphism.morph` is the canonical transformer block:

```

-- 12_algebra_homomorphism.morph
-- Demonstrates: parametric with homomorphism over (THE key CDL concept)

category Vect

endofunctor Sequence(512) on Vect

-- A parametric morphism that respects the Sequence structure.
-- The compiler checks: h . alpha = beta . F(h)
-- This IS the equivariance equation for neural network layers.
parametric TransformerBlock(dimension: Natural, heads: Natural)
  : Vector(256) -> Vector(256)
  homomorphism over Sequence
{
  pullback Residual(Attention(256, 8))
  pullback Residual(FeedForward(256, 1024))
}

```

The annotation **homomorphism** over `Sequence` is parsed into a `HomomorphismAnnotation` carrying a `Strictness` (strict, lax, or oplax, reflecting the three flavours of 2-categorical commutativity) and the endofunctor name. It attaches to the morphism’s type and becomes a proof obligation (Section 4.4). The commutativity it demands is the diagram of Section 2: *processing the sequence then folding equals folding then processing*.

### 3.5 Pullbacks: residual connections

A residual connection forks the input, transforms one branch, and merges the branches by addition—the universal fork-then-merge, i.e. a pullback. The example `13_pullback_residual.morph`:

```

category Vect

-- A residual connection is a pullback: fork then merge
pullback Residual : Vector(256) -> Vector(256) {
  fork(x) = pair(f(x), x)
  merge(a, b) = add(a, b)
}

```

```
-- the pullback gives the universal cone over the fork
endofunctor ResidualStream on Vect
```

Diagrammatically the residual block  $x \mapsto x + f(x)$  is the cone:

$$\begin{array}{ccc} R & \xrightarrow{\pi_2} & A \\ \pi_1 \downarrow & & \downarrow \text{id} \\ A & \xrightarrow{f} & A \end{array}$$

with the merge add realising the universal map. Note the transformer block above uses `pullback Residual(...)` around both its attention and feed-forward sublayers, exactly matching the post-norm residual structure the PyTorch backend emits.

### 3.6 Natural transformations, monads, limits, colimits

Mean pooling is a natural transformation  $\text{Sequence} \Rightarrow \text{Identity}$  whose component at  $A$  reduces an `Array(128, A)` to an `A` (from `09_natural_transformation.morph`):

```
category Vect

endofunctor Sequence(128) on Vect

-- Mean pooling is a natural transformation from Sequence to Identity
-- For every object A: eta_A : Sequence(A) -> A
natural mean_pool : Sequence(128) => Identity {
  component(A) : Array(128, A) -> A = mean_over_dimension_zero
}
```

Monads capture group actions and read-write memory, as in `10_monad.morph`:

```
category Vect

-- Group action monad: the foundation of geometric deep learning
-- Algebras for this monad = group representations
-- Algebra homomorphisms = equivariant neural networks
monad TranslationAction on Vect {
  endofunctor(A) = product(Integers, product(Integers, A))
  unit(A) = wrap_identity(A)
  multiplication(A) = compose_shifts(A)
}

-- State monad: computation with read-write memory
monad StateMemory(512) on Vect {
  endofunctor(A) = function(Vector(512), product(A, Vector(512)))
  unit(A) = pass_through(A)
  multiplication(A) = chain_state(A)
}
```

Attention and pooling are given their first-principles universal definitions in `25_limit_attention.morph`: attention is a `limit` (universal cone over the key-value diagram), and mean/max pooling are `colimits` (universal cocones):

```
category Vect

endofunctor Sequence(512) on Vect
```

```

-- Attention as a limit: universal cone over key-value diagram
limit AttentionLimit over Sequence on Vector(256) {
  cone(d) = weighted_aggregate(d)
}

-- Mean pooling as a colimit: universal cocone over feature diagram
colimit MeanPooling over Sequence on Vector(256) {
  cocone(d) = mean_pool(d)
}

-- Max pooling as another colimit
colimit MaxPooling over Sequence on Vector(256) {
  cocone(d) = max_pool(d)
}

parametric TransformerBlock(dimension: Natural)
  : Vector(256) -> Vector(256)
  homomorphism over Sequence
{
  pullback Residual(Attention(256, 4))
  pullback Residual(FeedForward(256, 1024))
}

```

The limit/colimit duality is visible in the diagram: a limit  $L$  receives projections  $\pi_j$  into the diagram objects  $D(j)$ , a colimit  $C$  receives injections  $\iota_j$  from them.



### 3.7 Losses and models

Loss functions are first-class morphisms  $\text{Product}(\text{predicted}, \text{target}) \rightarrow \text{Scalar}$ , each inducing an algebra on the prediction space (from `26_loss_functions.morph`):

```

category Vect

loss mean_squared_error(predicted, expected) = mse(predicted, expected)
loss cross_entropy_loss(predicted, expected) = nll(predicted, expected)
loss hinge_loss(predicted, expected) = hinge(predicted, expected)
loss huber_loss(predicted, expected) = smooth_l1(predicted, expected)

algebra loss_algebra for identity on Vector(10) {
  structure_map(prediction) = nll(prediction)
}

```

A **model** names a full composition with an optional fold algebra and step modifiers (**repeat**  $N$  for distinct-parameter repetition, **recurse**  $N$  for weight-tied repetition, **share** for weight tying, **parallel** for product morphisms). We return to concrete models in Section 5.

## 4 The Compiler

The Morpheus compiler is written in Rust—chosen for memory safety without a garbage collector, fast compilation (important for architecture-search loops), and a type system that is itself algebraic. The crate root declares the pipeline’s modules:

```

///! Morpheus -- The Categorical Deep Learning Language
///!
///! A compiled, statically-typed language where category theory IS the type system.
///! Every program that type-checks is a mathematically valid neural architecture.

pub mod lexer;
pub mod parser;
pub mod types;
pub mod ir;
pub mod codegen;
pub mod lean;
pub mod imports;
pub mod testing;
pub mod search;
pub mod symbolic_search;
pub mod error;
pub mod span;

```

The flow is: source  $\rightarrow$  lexer  $\rightarrow$  parser (categorical AST)  $\rightarrow$  type checker (composition + homomorphism checks)  $\rightarrow$  optional Lean proof obligations  $\rightarrow$  Categorical IR  $\rightarrow$  diagram optimisation  $\rightarrow$  backend codegen (PyTorch, and others). We describe the load-bearing stages.

## 4.1 Lexer and parser

The lexer tokenises the surface syntax (`--` line comments, `;` composition, keywords such as `category`, `parametric`, `homomorphism`, `over`). The parser produces the AST of Section 3. Expressions capture the categorical operators directly—composition, blocks, repeat/recurse, pullback sugar, parallel branches, and identity/inverse:

```

pub enum Expr {
  Ident(String),
  Call { func: String, args: Vec<CallArg> },
  Compose { left: Box<Expr>, right: Box<Expr> },           // f ; g
  Block(Vec<Expr>),                                       // { f \n g \n h }
  Repeat { expr: Box<Expr>, count: i64 },                 // block repeat 6
  Recurse { expr: Box<Expr>, depth: i64 },                // block recurse 8
  IdentityExpr(Box<TypeExpr>),                           // identity(Type)
  InverseExpr(Box<Expr>),                                 // inverse(f)
  PullbackExpr { name: String, inner: Box<Expr> },       // pullback Residual(f)
  ParallelExpr { branches: Vec<(String, Box<Expr>)> },    // parallel { ... }
  // ... BinOp (incl. MatMul '@'), Lambda, ScanExpr, FoldExpr, Tuple
}

```

A parametric definition records its source and target objects, an optional homomorphism annotation, its parameter declarations, and its forward body. The homomorphism annotation is where 2-categorical strictness lives:

```

pub enum Strictness { Strict, Lax, OpIax }

pub struct HomomorphismAnnotation {
  pub strictness: Strictness,
  pub endofunctor: String,
  pub monad: Option<String>,
}

```

## 4.2 The categorical type checker

Type checking is categorical checking. The environment seeds the built-in categories and the polymorphic activation morphisms, and knows how to resolve built-in parametric layers to their source/target objects and parameter shapes. For instance, `Linear(i, o)` becomes a parametric morphism  $\text{Vector}(i) \rightarrow \text{Vector}(o)$  with a weight of shape  $[o, i]$  and a bias of shape  $[o]$ ; `Attention(d, h)` an endomorphism on  $\text{Vector}(d)$  with four  $d \times d$  projection matrices; `Embedding(v, d)` a morphism  $\text{Discrete}(v) \rightarrow \text{Vector}(d)$ :

```
fn resolve_builtin_call(&self, name: &str, args: &[i64]) -> Option<MorphismType> {
  match (name, args) {
    ("Linear", [i, o]) | ("linear", [i, o]) => Some(MorphismType::parametric(
      ObjectType::Vector(*i as usize), ObjectType::Vector(*o as usize),
      vec![ParamInfo { name: "weight".into(), shape: vec![*o as usize, *i as
        usize] },
        ParamInfo { name: "bias".into(), shape: vec![*o as usize] }],
    )),
    ("Attention", [d, h]) | ("attention", [d, h]) => Some(MorphismType::
      parametric(
        ObjectType::Vector(*d as usize), ObjectType::Vector(*d as usize),
        vec![ParamInfo { name: "weight_query".into(), shape: vec![*d as usize,
          *d as usize] },
          ParamInfo { name: "weight_key".into(), shape: vec![*d as usize,
            *d as usize] },
          ParamInfo { name: "weight_value".into(), shape: vec![*d as usize,
            *d as usize] },
          ParamInfo { name: "weight_output".into(), shape: vec![*d as usize,
            *d as usize] }],
        )),
    ("Embedding", [v, d]) | ("Embed", [v, d]) => Some(MorphismType::parametric(
      ObjectType::Discrete(*v as usize), ObjectType::Vector(*d as usize),
      vec![ParamInfo { name: "table".into(), shape: vec![*v as usize, *d
        as usize] },
        ParamInfo { name: "position".into(), shape: vec![2048, *d as usize
          ] }],
    )),
    // FeedForward, StateSpaceModel, GatedRecurrentUnit, Convolution2D,
    // LanguageModelHead, MessagePassing, ...
    _ => None,
  }
}
```

The central judgement is composition. When checking `f ; g`, the checker verifies that `f.target` is compatible with `g.source`; if not, it raises a *categorical* error (not a shape error), reporting which object failed to match. On success it builds the composite morphism, tensoring the parameter lists—exactly the **Para** composition rule:

```
Expr::Compose { left, right } => {
  let f = self.check_expr(left, span)?;
  let g = self.check_expr(right, span)?;
  if !f.target.compatible(&g.source) {
    return Err(MorpheusError::TypeError {
      span,
      msg: format!("cannot compose:\n {f}\n ; {g}\n because {} is not \
        compatible with {}", f.target, g.source),
    });
  }
  let source = if matches!(f.source, ObjectType::Poly(_)) { g.source.clone() }
```

```

        else { f.source.clone() };
let target = if matches!(g.target, ObjectType::Poly(_)) { f.target.clone() }
        else { g.target.clone() };
let mut params = f.params.clone();
params.extend(g.params.iter().cloned());
Ok(MorphismType {
    source, target, is_parametric: f.is_parametric || g.is_parametric,
    params, repeat_count: 1, recursion_depth: 1, homomorphism: None,
})
}

```

The modifiers `repeat` and `recurse` are only well-typed on endomorphisms—their source and target objects must be compatible—since repeated or weight-tied application requires  $A \rightarrow A$ . The checker enforces this and records the repeat count or recursion depth on the morphism type:

```

Expr::Repeat { expr, count } => {
    let mt = self.check_expr(expr, span)?;
    if !mt.source.compatible(&mt.target) {
        return Err(MorpheusError::TypeError {
            span, msg: format!("repeat requires endomorphism, got: {mt}"),
        });
    }
    Ok(MorphismType { repeat_count: *count as usize, ..mt })
}

```

A `MorphismType` thus records source, target, whether it is parametric, its parameters, repeat count, recursion depth, and any homomorphism annotation. Its `Display` instance prints morphisms the way a category theorist would read them—tagging the category (**Para** or **Vect**), the source/target objects, and any `repeat/recurse/homomorphism` over decorations.

### 4.3 Categorical IR and PyTorch codegen

After checking, the program is lowered to a Categorical IR (CIR): a diagram of typed nodes (embeddings, linear maps, attention, feed-forward, SSM, GRU, convolution, message passing, head, pullback) carrying their dimensions, plus weight-tying edges and a recursion depth. The CIR is the categorical “assembly language”; it can be serialised, diagram-rewritten (fuse adjacent morphisms, factor shared parameters, eliminate dead morphisms), verified, and compiled to any backend.

The PyTorch backend walks the CIR and emits a `torch.nn.Module`. Each node contributes an `__init__` fragment (parameter construction) and a `forward` fragment. A residual attention sublayer, for example, emits a `MultiheadAttention` with a post-residual `LayerNorm`; a feed-forward sublayer emits the post-norm residual GELU-MLP:

```

IRNode::Attention { id, d_model, n_heads } => (
    format!("self.attention_{id} = nn.MultiheadAttention({d_model}, {n_heads}, \
        batch_first=True)\n\
        self.attention_normalization_{id} = nn.LayerNorm({d_model})"),
    format!("_residual, _ = self.attention_{id}(x, x, x, need_weights=False)\n\
        x = self.attention_normalization_{id}(x + _residual)"),
),
IRNode::FeedForward { id, d_model, d_ff } => (
    format!("self.feedforward_up_{id} = nn.Linear({d_model}, {d_ff})\n\
        self.feedforward_down_{id} = nn.Linear({d_ff}, {d_model})\n\
        self.feedforward_normalization_{id} = nn.LayerNorm({d_model})"),
    format!("x = self.feedforward_normalization_{id}(x + \
        self.feedforward_down_{id}(F.gelu(self.feedforward_up_{id}(x))))"),
)

```

```
),
```

Both sublayers normalise *after* the residual addition, computing  $\text{LayerNorm}(x + f(x))$ —the post-norm (post-residual) convention. Weight tying—a 2-morphism in **Para**—is emitted literally as a parameter-aliasing assignment, so that the embedding table and the language-model head share storage:

```
for (lhs, rhs, via) in &ir.ties {
  let via_note = via.as_ref().map(|v| format!(" (via {v})")).unwrap_or_default();
  code.push_str(&format!("          # Weight tying (2-morphism){via_note}\n"));
  code.push_str(&format!("          self.{l} = self.{r}\n", strip(lhs), strip(rhs)));
}
```

Weight-tied recursion (`recurse N`) compiles to an explicit loop over the same submodules—an effective depth of  $N$  with the parameters of depth one:

```
if recursion {
  code.push_str(&format!("          # Recursive application (weight-tied, depth={})\n",
    ir.recursion_depth));
  code.push_str(&format!("          for _recursion in range({}): \n", ir.
    recursion_depth));
  // ... emit the forward body indented inside the loop ...
}
```

The generated module exposes a `param_count` helper and, when a language-model head is present, folds a cross-entropy loss into `forward` when labels are supplied. Every generated file is headed with the provenance comment “Every composition was verified by the categorical type checker.”

#### 4.4 The Lean 4 verification backend

Beyond dimension checking, Morpheus can emit *proof obligations* to Lean 4 [8]. The design rests on Mathlib’s existing category-theory library (`CategoryTheory.Category.Basic`, `Monad.Basic`, `Limits`, `Adjunction`); Morpheus provides a bridge library, `MorpheusProof`, that defines the concrete categories (**Vect**, **Para(Vect)**, **Set**), translates Morpheus types and morphisms to Lean, and supplies the automation tactic `morpheus_verify`.

The obligation extractor walks the AST; each construct that requires a proof contributes an `ObligationKind`. A parametric morphism with a homomorphism annotation yields an `AlgebraHomomorphism` obligation; functors yield functor-law obligations; monads yield monad-law obligations; naturals, adjunctions, (co)algebras, (co)limits, isomorphisms each yield their characteristic obligation:

```
for stmt in &program.statements {
  match stmt {
    Statement::ParametricDef(p) if p.homomorphism.is_some() => {
      let hom = p.homomorphism.as_ref().unwrap();
      obligations.push(ProofObligation {
        name: p.name.clone(),
        kind: ObligationKind::AlgebraHomomorphism {
          layer: p.name.clone(),
          endofunctor: hom.endofunctor.clone(),
        },
      });
    }
    Statement::FunctorDef(f) => obligations.push(ProofObligation {
      name: f.name.clone(),
```

```

        kind: ObligationKind::FunctorLaws { functor: f.name.clone() },
    }},
    // MonadDef, NaturalDef, AdjunctionDef, AlgebraDef, CoalgebraDef,
    // LimitDef, ColimitDef, IsomorphismDef, EndofunctorDef, ...
    _ => {}
}
}

```

The generator then writes a single `MorpheusGenerated.lean` importing `MorpheusProof`, with no `sorry`—every theorem is to be machine-checked. The algebra-homomorphism obligation for a layer over the sequence endofunctor is emitted as a theorem about the last-element fold, discharged by `morpheus_verify`:

```

ObligationKind::AlgebraHomomorphism { layer, endofunctor } => {
  code.push_str(&format!(
r#"/-- Proof: {layer} is an algebra homomorphism over {endofunctor}.
  Verified: processing then folding = folding then processing. -/
theorem {layer}_is_algebra_homomorphism
  (n d1 d2 : Nat) (hn : n > 0)
  (h : MorpheusMorphism d1 d2) :
  SequenceAlgebraHomomorphism n d1 d2
    (lastElementAlgebra n d1 hn)
    (lastElementAlgebra n d2 hn)
  h := by
    morpheus_verify
"#));
}

```

The bridge library defines the `Vect` category, composition and identity, the proofs of the identity and associativity laws, the sequence endofunctor with its functor-law proofs, and the  $F$ -algebra and algebra-homomorphism structures. The core definitions, quoted from the spec:

```

/-- An F-algebra: carrier type A + structure map a : F(A) -> A -/
structure FAlgebra (F : Type -> Type) (A : Type) where
  structure_map : F A -> A

/-- An algebra homomorphism h : (A,a) -> (B,b) is a morphism h : A -> B
such that h . a = b . F(h).
This IS the equivariance equation for neural network layers. -/
structure AlgebraHomomorphism
  (F : Type -> Type) [Functor F]
  (algA : FAlgebra F A) (algB : FAlgebra F B)
  (h : A -> B) : Prop where
  commutes : h . algA.structure_map = algB.structure_map . Functor.map h

```

The `morpheus_verify` tactic is a macro that unfolds the `Morpheus` definitions, applies function extensionality, simplifies with `Mathlib`'s category-theory simp set, tries algebraic solvers (`ring`, `omega`), and falls back to reflexivity. Verification is invoked by copying the generated file into the Lean project and running `lake build`; on failure, the compiler extracts the Lean `error`: lines and translates them into a human-readable explanation of which categorical law was violated. If Lean is not installed the compiler prints an install hint and continues without the formal guarantee, so verification is opt-in rather than mandatory.

## 5 Worked Examples

We now assemble the constructs into full models. Each is a real program from the language’s example suite, and each exercises a different endofunctor.

### 5.1 An image classifier over the grid endofunctor

Image classification processes grid-structured data. The example `19_model_image_classifier.morph` declares the  $224 \times 224$  grid endofunctor, a global-average-pooling algebra (the grid’s fold), a spatial layer that respects the grid structure, and a model that maps a grid of pixels to a class:

```
category Vect

type Pixel    : Vect = Vector(3)
type Feature  : Vect = Vector(64)
type Class    : Set  = Discrete(1000)

endofunctor Grid(224, 224) on Vect

algebra global_average for Grid on Vector(64) {
  structure_map(grid) = mean_spatial(grid)
}

-- spatial layer preserves Grid structure
parametric SpatialLayer(dimension: Natural)
  : Vector(64) -> Vector(64)
  homomorphism over Grid
{
  Linear(64, 64)
}

model ImageClassifier : Grid[Pixel] -> Class
  with global_average
{
  Linear(3, 64)
  Linear(64, 64) repeat 4
  Linear(64, 1000)
}
```

The `with global_average` clause names the fold used to collapse the grid before classification. The `Linear(64, 64) repeat 4` stage is four distinct-parameter layers; each `Linear` type-checks against the next because their objects agree. The classification head maps `Vector(64)` to the thousand-way class object.

### 5.2 A graph attention network

Molecular property prediction processes graphs. The example `advanced_graph_transformer.morph` reads graph attention as an algebra homomorphism over the molecular-graph endofunctor, with a sum readout as the graph’s fold. The accompanying comment is precise about the categorical content: the attention is a natural transformation between graph algebras, and the layer is permutation-equivariant because permuting nodes then attending equals attending then permuting.

```
category Vect

type AtomFeature    : Vect = Vector(64)
type MolProperty    : Vect = Vector(1)
```

```

endofunctor MolecularGraph(128) on Vect

algebra sum_readout for MolecularGraph on Vector(64) {
  structure_map(graph) = sum_nodes(graph)
}

-- Graph attention layer: attention-weighted message passing
-- This is an algebra homomorphism over MolecularGraph
parametric GraphAttentionLayer(dimension: Natural)
  : Vector(64) -> Vector(64)
  homomorphism over MolecularGraph
{
  MessagePassing(64)
  layer_normalization
  Linear(64, 64)
  gaussian_error_linear
}

-- Full graph attention network
model GraphAttentionNet : MolecularGraph[AtomFeature] -> MolProperty
  with sum_readout
{
  Linear(64, 64)
  GraphAttentionLayer(64) repeat 6
  Linear(64, 1)
}

```

The permutation equivariance is exactly the homomorphism square with  $F = \text{MolecularGraph}$  and the sum aggregation as structure map; sum is permutation-invariant, so the readout genuinely commutes with node processing.

### 5.3 A VAE in the Kleisli category of a Gaussian monad

A variational autoencoder is, categorically, a morphism in the Kleisli category of the Gaussian-distribution monad: the encoder maps data to a distribution over latents  $X \rightarrow T(Z)$  and the decoder maps latent samples to a distribution over data  $Z \rightarrow T(X)$ . The ELBO is reconstruction plus KL divergence, and the reparameterisation trick  $z = \mu + \sigma\varepsilon$  makes sampling differentiable. The example `advanced_vae_full.morph`:

```

category Vect

type Input  : Vect = Vector(784)
type Latent : Vect = Vector(64)

monad GaussianDistribution on Vect {
  endofunctor(A) = product(A, A)
  unit(x) = dirac(x)
  multiplication(d) = reparameterize(d)
}

-- Encoder: maps input to PARAMETERS of a distribution (mean + log_variance)
-- This is a Kleisli morphism: X -> T(Z)
parametric Encoder(input_dim: Natural, latent_dim: Natural)
  : Vector(784) -> Vector(128)
{
  Linear(784, 512)
}

```

```

    rectified_linear
    Linear(512, 256)
    rectified_linear
    Linear(256, 128)
}

-- Decoder: maps latent sample back to reconstruction (Kleisli: Z -> T(X))
parametric Decoder(latent_dim: Natural, output_dim: Natural)
  : Vector(64) -> Vector(784)
{
  Linear(64, 256)
  rectified_linear
  Linear(256, 512)
  rectified_linear
  Linear(512, 784)
  sigmoid
}

-- Full VAE pipeline (encode -> sample -> decode)
let vae = Encoder(784, 128) ; Linear(128, 64) ; Decoder(64, 784)

adjunction VAE {
  left Encoder : Vect -> Vect
  right Decoder : Vect -> Vect
}

```

Note the encoder’s target is `Vector(128)`: it emits the concatenated mean and log-variance ( $2 \times 64$ ), and the `Linear(128, 64)` is the reparameterisation projection to the sampled latent, which the decoder consumes. The `let vae = ... ; ... ; ...` composition type-checks precisely because  $128$  (encoder out) =  $128$  (projection in) and  $64$  (projection out) =  $64$  (decoder in).

## 5.4 An encoder–decoder adjunction

The autoencoder’s duality is made explicit in `24_adjunction_autoencoder.morph`, which declares the adjunction and asks the compiler to `check` laws—the triangle identities  $\varepsilon_F \circ F(\eta) = \text{id}_F$  and  $G(\varepsilon) \circ \eta_G = \text{id}_G$ :

```

category Vect

type Data    : Vect = Vector(784)
type Latent  : Vect = Vector(64)

adjunction AutoEncoder {
  left Encoder : Vect -> Vect
  right Decoder : Vect -> Vect
}

parametric EncoderNet(input_dim: Natural, latent_dim: Natural)
  : Vector(784) -> Vector(64)
{
  Linear(784, 256)
  rectified_linear
  Linear(256, 64)
}

parametric DecoderNet(latent_dim: Natural, output_dim: Natural)
  : Vector(64) -> Vector(784)
{

```

```

    Linear(64, 256)
    rectified_linear
    Linear(256, 784)
    sigmoid
}

-- The unit: Id -> Decoder.Encoder (reconstruction ~ identity)
let autoencoder = EncoderNet(784, 64) ; DecoderNet(64, 784)
check laws(AutoEncoder)

```

The adjunction diagram is the familiar unit/counit picture:

$$\begin{array}{ccc}
 & \xrightarrow{F=\text{Encoder}} & \\
 \mathbf{Vect}_{\text{data}} & & \mathbf{Vect}_{\text{latent}} \\
 & \xleftarrow{G=\text{Decoder}} & 
 \end{array}
 \quad F \dashv G, \quad \eta : \text{Id} \Rightarrow G \circ F, \quad \varepsilon : F \circ G \Rightarrow \text{Id}.$$

Reconstruction loss is the quantitative surrogate for “ $G \circ F$  is close to the identity”: minimising it tightens the unit  $\eta$ .

## 5.5 Type-directed architecture search

Architecture search is a language primitive, not a library. The example `21_architecture_search.morph` declares a search over dimension, head count, depth, and recursion, with a `choose/option` block enumerating candidate block patterns, and budget/target/optimiser/data clauses:

```

category Vect

type Token   : Set   = Discrete(32000)
type Hidden : Vect  = Vector(256)

endofunctor Sequence(256) on Vect

parametric SearchableBlock(dimension: Natural)
  : Vector(256) -> Vector(256)
  homomorphism over Sequence
{
  Linear(256, 256)
}

search OptimalLM : Sequence[Token] -> Sequence[Token] {
  let dimension in [128, 256, 384, 512]
  let heads in [4, 8]
  let depth in 2..12
  let recursion in [1, 2, 4, 8]

  Embedding(32000, dimension)

  choose repeat depth {
    option TransformerBlock(dimension, heads)
    option StateSpaceModelBlock(dimension, state = 128)
    option {
      StateSpaceModel(dimension, state = 128)
      FeedForward(dimension)
      Attention(dimension, heads)
      FeedForward(dimension)
    }
  }
} recurse recursion

```

Table 1: The Morpheus Rosetta stone: each category-theoretic concept is a language construct and a deep-learning concept.

| Category theory        | Morpheus keyword                          | Deep-learning concept                  |
|------------------------|-------------------------------------------|----------------------------------------|
| Category               | <code>category</code>                     | A universe of objects + arrows         |
| Object                 | <code>type</code>                         | Tensor space (shape + dtype)           |
| Morphism               | <code>morphism</code>                     | Pure function between spaces           |
| Parametric morphism    | <code>parametric</code>                   | Learnable layer in <b>Para(Vect)</b>   |
| Functor                | <code>functor</code>                      | Structure-preserving map               |
| Endofunctor            | <code>endofunctor</code>                  | Data-structure type (seq, grid, graph) |
| Natural transformation | <code>natural</code>                      | Polymorphic op (e.g. pooling)          |
| Monad                  | <code>monad</code>                        | Group action / state / memory          |
| $F$ -algebra           | <code>algebra</code>                      | Fold / readout structure map           |
| Algebra homomorphism   | <code>homomorphism</code> over            | <b>Neural network layer</b>            |
| Adjunction             | <code>adjunction</code>                   | Encoder–decoder duality                |
| Limit / colimit        | <code>limit</code> / <code>colimit</code> | Attention / pooling                    |
| Pullback               | <code>pullback</code>                     | Residual connection                    |
| 2-morphism             | <code>share</code> / <code>recurse</code> | Weight tying / sharing                 |

```

LanguageModelHead(dimension, 32000)

budget  params < 50_000_000
budget  memory < 4GB
target  min perplexity
using    AdamW4
data     WikiText2
}

```

The decisive advantage over conventional neural architecture search is that the type system prunes structurally invalid candidates *before* any training: a candidate whose block objects fail to compose, or which annotates a non-homomorphic layer over the sequence endofunctor, is rejected at compile time. The search space the optimiser actually explores is the far smaller space of *well-typed* diagrams. The `SearchDef` AST node even supports a second `SearchMode::Symbolic` that searches over typed compositions of primitive morphisms and emits the result as a Lean obligation—type-directed program synthesis in the same construct.

**One grammar, every domain.** Table 1 collects the Rosetta correspondence that the examples instantiate: each categorical concept is simultaneously a language keyword and a deep-learning concept. The point the example suite makes concrete is that only the `endofunctor` / `structure` declaration changes between domains; the composition grammar is invariant.

## 6 Discussion and Limitations

**What the type checker actually guarantees today.** The implemented type checker enforces composition compatibility object-by-object (Section 4) and the endomorphism requirement for `repeat/recurse`. These already turn a large class of shape mismatches from runtime crashes into compile-time categorical errors, reported in the vocabulary of objects and morphisms rather than tensor shapes. The homomorphism annotation is parsed, typed, and turned into a proof obligation, but its *semantic* discharge is delegated to the optional Lean backend; when Lean is absent the annotation is recorded and trusted, not proven. Thus the slogan “if

it compiles, it is structurally valid” holds unconditionally only for the composition/dimension fragment, and conditionally—upon successful Lean verification—for the equivariance fragment.

**Attention is not a strict fold homomorphism.** The most important honesty in the design, stated plainly in the Lean integration spec, is that self-attention is *not* a strict algebra homomorphism for the last-element fold: attention reads the entire sequence to produce each output, so it cannot commute with collapsing the sequence to a single element first. The correct verdict is that the proof *should fail*, and the compiler should explain that attention is a homomorphism for the sequence-to-sequence (identity) algebra, not for a reductive fold. This is exactly the kind of silent architectural error that the system is designed to surface. It also tempers the claim: the clean “layer = strict algebra homomorphism” story is literal for linear/convolutional/message-passing layers but requires the *lax* 2-categorical formulation—commutativity up to a reparameterisation 2-morphism—for attention and for the weight-tying patterns of recurrent and recursive networks, matching the scope of [1].

**Proof automation and cost.** Many obligations are structurally simple (identity and associativity laws, functor laws for map-based endofunctors, isomorphisms for reshape/transpose) and fall to `simp/ring/omega` inside `morpheus_verify`. Harder obligations (algebra homomorphism for specific mechanisms) need domain lemmas, and some are undecidable, for which the design provides a numerical-sampling fallback with an explicit “verified numerically, not formally proven” warning. Lean invocation adds seconds to tens of seconds of compile time—acceptable for a compiled language where correctness dominates, and cached across unchanged sources, but not free.

**Backends and scope.** The PyTorch backend is the one quoted here; the specification envisions JAX, ONNX, StableHLO, Rust-native inference, and a CPU-first GCPU backend, but these are at varying stages. The generated PyTorch is a faithful transcription of the CIR—e.g. the selective-scan loop emitted for the state-space-model node—rather than a maximally optimised kernel; diagram-level optimisations (morphism fusion, shared-parameter factoring) are specified as IR rewrites. The surface syntax has mild duplication (`parametric/homomorphism` over versus the showcase’s `transform/structure` sugar) reflecting its evolution across grammar versions.

**Relation to prior work.** Morpheus is a language-level realisation of a research program rather than a new theory. The compositional and parametric foundations are those of [3, 2, 6]; the algebra-homomorphism-as-layer thesis is [1], surveyed by [5]; the categorical background is standard [7, 4]; the verification leans on Lean 4 [8] and the runtime on PyTorch [9]. The novel element is the insistence that these ideas live in a *type system* with a compiler, so that architecture validity is decided by the same machinery that decides whether a program is well-formed.

## 7 Conclusion

Morpheus takes the thesis that neural network layers are algebra homomorphisms and makes it operational: a compiled, statically typed language in which objects are tensor spaces, morphisms are (parametric) layers in **Para(Vect)**, composition is a checked categorical operation, and the equivariance of a layer is a proof obligation over an endofunctor. The compiler lexes and parses a categorical AST, type-checks composition compatibility in the object vocabulary of the problem, lowers to a Categorical IR, and emits PyTorch, with an optional Lean 4 backend that machine-checks the categorical laws. Through the language’s real example programs—linear

maps, morphisms and composition, functors, natural transformations, monads, the algebra-homomorphism transformer block, residual pullbacks, attention-as-limit, an image classifier, a graph attention network, a Kleisli-category VAE, an encoder-decoder adjunction, and type-directed search—we have shown that one categorical grammar spans sequences, grids, graphs, sets, and products, with only the endofunctor changing.

The honest boundaries are equally part of the contribution: the unconditional guarantee today is the composition/dimension fragment, the equivariance fragment is conditional on Lean, and self-attention correctly *fails* to be a strict fold homomorphism—an error the system is built to expose rather than hide. What Morpheus offers is a shift of a large class of architectural mistakes from silent runtime bugs to compile-time theorems, and a cognitive interface in which practitioners reason in the abstraction space of categories rather than the syntax space of tensors. If the history of mathematics is a history of pushing semantics into syntax so that it can be automated, Morpheus is an attempt to do the same for the design of neural architectures.

## References

- [1] B. Gavranović, P. Lessard, A. Dudzik, T. von Glehn, J. G. M. Araújo, and P. Veličković. Position: Categorical Deep Learning is an Algebraic Theory of All Architectures. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024.
- [2] G. S. H. Cruttwell, B. Gavranović, N. Ghani, P. Wilson, and F. Zanasi. Categorical Foundations of Gradient-Based Learning. In *Programming Languages and Systems (ESOP)*, 2022.
- [3] B. Fong, D. I. Spivak, and R. Tuyéras. Backprop as Functor: A Compositional Perspective on Supervised Learning. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2019.
- [4] B. Fong and D. I. Spivak. *Seven Sketches in Compositionality: An Invitation to Applied Category Theory*. Cambridge University Press, 2019.
- [5] F. Crescenzi. Towards a Categorical Foundation of Deep Learning: A Survey. *arXiv preprint arXiv:2410.05353*, 2024.
- [6] D. I. Spivak. Learners’ Languages. In *Applied Category Theory (ACT)*, 2021.
- [7] S. Mac Lane. *Categories for the Working Mathematician*. Springer, 1978.
- [8] L. de Moura and S. Ullrich. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction (CADE)*, 2021.
- [9] A. Paszke et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.