Cognitive Closure: Formally Bound Learning

Anonymous

February 8, 2026

Abstract

We present *Cognitive Closure*, the terminal unification of the machine intelligence stack. Building upon the static grounding of *Typed Reality* and the computational fusion of *Compiled Cognition*, we extend the cognitive compiler to support a formally bound backward pass. By introducing high-level training pragmas—namely `@training`, `@frozen`, and `@train`—we enable the differentiable optimization of the *Fused Fabric* under strict ontological constraints. This system allows for the emergence of a "Formally Bound Latent Space": a shared, interpretable domain where human engineers, autonomous AI agents, and emerging AGI can collaborate, comprehend, and co-train with mathematical guarantees of semantic preservation. We demonstrate that by closing the loop between observation, reasoning, and learning, we achieve a stable state of machine comprehension where the "black-box" nature of neural updates is replaced by verifiable state transitions in a grounded reality fabric.

Contents

0.1 Introduction

The arc of this research has moved from the static to the dynamic: from the grounding of raw sensors (*Typed Reality*) to the fusion of computational intent (*Compiled Cognition*), and finally to their topological unification (*Fused Fabric*). However, a system that can only execute what is currently known is incomplete. True intelligence requires the ability to update its internal representation of the world while maintaining the formal integrity of that world.

We introduce *Cognitive Closure*. In this phase, the Cognitive Compiler is no longer restricted to the forward-pass execution of fixed graphs. It now encompasses the *backward-pass*: the differentiable optimization of parameters under the oversight of ontological constraints.

By closing this loop, we move from "Machine Learning" (unconstrained optimization) to "Targeted Comprehension" (constrained refinement). The result is a system where learning is not a stochastic drift in a vacuum, but a verifiable navigation of the grounded fabric of reality.

0.2 Training Pragmas: Formalizing the Backward Pass

To enable closure, we extend the Fuse-IR with three foundational pragmas that control the optimization of the Fused Fabric.

`@training` This pragma declares the objective function and initialization states. It binds the compute graph to a differentiable context, signalling to the compiler to generate the adjoint (gradient) nodes for the targeted silicon runtime.

`@frozen` The Fused Fabric contains many nodes that are "closed to change" (e.g., physical laws, fundamental ontological definitions, or pre-verified reasoning motifs). The `@frozen` pragma prevents the backward pass from modifying these subgraphs, ensuring that core grounding is never corrupted by stochastic updates.

`@train` This pragma marks specific parameter tensors within the graph as mutable targets for optimization. Unlike untyped systems where all "weights" are mutable, `@train` allows for precision learning: updating only the specific components of a reasoning chain that require refinement.

0.3 Typed Differentiation and Semantic Preservation

The central innovation of Cognitive Closure is *Typed Differentiation*. In traditional backpropagation, gradients are untyped updates applied to latent weights. In our system, the gradient itself is a typed object in the Reality Fabric $\mathcal{R}$.

Axiom 1 (Gradient Grounding)**.** *For any update $\Delta\theta$ applied to a parameter $\theta \in \mathcal{R}$, the resulting state $\theta' = \theta + \Delta\theta$ must satisfy the semantic invariants $\Sigma(\theta)$ defined in the initial grounding.*

If a parameter is grounded as a "Normalized Distribution," the optimizer is formally prevented from applying an update that would lead to a non-normalized state. By embedding constraints (normalization, physical ranges, symbolic boundaries) directly into the backward pass, we achieve a form of "Self-Correcting Reasoning" that is mathematically impossible to exit the bounds of reality.

0.4 Constrained Optimization in the Fused Fabric

We formalize the constrained update rule that enforces semantic invariants during learning.

Projection-Based Updates

Let θ be a parameter tensor with semantics Σ. Let $g = \nabla_\theta \mathcal{L}$ be the gradient produced by the adjoint graph. The update in a generic optimizer (e.g., SGD with step-size η) becomes:

$$\theta' = \Pi_\Sigma \left(\theta - \eta g\right)$$

where Π_Σ is a projection operator onto the set of semantically valid states for θ (e.g., simplex projection for probability distributions, clipping for bounded measurements, vocabulary-preserving projection for enums).

Projected Gradient Descent (PGD) When Σ defines a convex feasible set, the above projection yields classical projected gradient descent. We present the generic algorithm below.

```
# Pseudo-code: Typed Projected Gradient Descent
initialize theta_0 respecting Sigma
for t in 0..T:
  g_t = AD(Primal, theta_t)         # adjoint graph returns gradient
  theta_temp = theta_t - eta * g_t  # gradient step
  theta_{t+1} = Project_Sigma(theta_temp)  # semantic projection
end

author: converge_if_convex(Sigma, eta)
```

Convergence Guarantees

Theorem 1. *If Σ is convex and the loss $\mathcal{L}$ is Lipschitz-smooth, then with a sufficiently small step-size η, Typed PGD converges to a stationary point of $\mathcal{L}$ restricted to Σ.*

Sketch. Standard PGD convergence proofs apply ([?]). The novelty here is that each projection is semantics-aware (the projection operator implements type-specific constraints). Convergence properties follow from convexity of Σ and smoothness of $\mathcal{L}$. □

Non-Convex and Discrete Semantics

When Σ is non-convex (structured combinatorial constraints) or contains discrete components (e.g., Enums), we combine projection with approximation (soft-relaxations) or with specialized combinatorial solvers injected as optimizer primitives. For discrete enums, the system learns an embedding and maps updates back with an argmax or Gumbel-softmax relaxation during training.

0.5 The Formally Bound Latent Space

The unification of observation, reasoning, and learning creates a new kind of computational environment: the *Formally Bound Latent Space.*

Definition 1 (Praxis). *A* praxis *is a bounded cognitive configuration described by:*

```
praxis <Name> {
  input: [Sensor, ...]
  output: [Actuator, ...]
  transform: [Transform, ...]     // primative computational transforms
  model: Graph                     // self-model of computation and observation
  constraint: [Constraint, ...]
  reconfigure: {                    // optional: safe reconfiguration policy
    scope: [S|A|C],                 // sensors, actions, computation
    policy: IRI                     // reference to policy & invariants
  }
}
```

where each field is annotated with ontology IRIs. A praxis is the minimal unit that can be fused, trained, and audited in the Fused Fabric.

Praxis Checklist (safe reconfiguration) *To enable practical, auditable self-modification we recommend the following checklist for every praxis that exposes reconfiguration:*

1. *Declare the reconfiguration `scope` (S/A/C) and the invariants that must hold post-update.*
2. *Provide a verification artifact (assertion graph or test-suite) that can be executed automatically after any update.*
3. *Sandboxed trial: allow candidate updates to be evaluated in an isolated environment with conservative time/resource budgets.*
4. *Commit policy: only atomically commit updates that pass verifiers and whose reconfiguration cost is within acceptable bounds.*
5. *Provenance audit: record the update, its rationale, and verification outcome in the Praxis provenance graph.*

These pragmatic steps align praxis with real-world deployment constraints and make self-modifying behavior auditable and reversible.

In untyped deep learning, the latent space is an uninterpretable "black box." In the Fused Fabric the latent space is a topological map whose coordinates are associated with semantic descriptors from the Reality Fabric. Below we provide a formal treatment that supports provable invariants, projection-based updates, and human-participant interaction.

Definition 2 (Formally Bound Latent Space). *Let $\mathcal{X}$ be the observation space and $\mathcal{L} = \mathbb{R}^d$ a finite-dimensional latent space. A* Formally Bound Latent Space *is a tuple $(\mathcal{L}, E, \phi)$ where*

- *$E : \mathcal{X} \to \mathcal{L}$ is an embedding induced by a fused graph, and*
- *$\phi : \mathcal{L} \to \mathcal{S}$ maps latent points to semantic descriptors $s \in \mathcal{S}$ (OWL/RDF IRIs and associated constraints).*

We require that for any observation x, assertions($E(x)$) $\models \phi(E(x))$ (semantic fidelity).

Semantic Projection and Typed Updates *Let $\Sigma(s)$ be the constraint set associated with a semantic descriptor s. The semantic projection operator*

$$\Pi_\Sigma : \mathbb{R}^d \to \Sigma$$

maps an unconstrained vector to the nearest semantically-feasible point. Typed updates are then defined by

$$z_{t+1} = \Pi_\Sigma \left(z_t - \eta \nabla_z \mathcal{L}(z_t) \right).$$

This update rule guarantees that iterates remain feasible at every step and that the projection is applied atomically at the parameter commit boundary.

Quantitative Metrics *We measure the quality of a bound latent space using:*

- ***Semantic Fidelity Rate (SFR)****: fraction of test latents satisfying their RDF constraints.*
- ***Axis Interpretability (AI)****: accuracy of probes that predict ontology labels from coordinates.*
- ***Projection Stability (PS)****: expected change in model outputs under projection cycles.*

Guarantees *Under standard smoothness and convexity assumptions on $\mathcal{L}$ and Σ, Typed Projected Gradient Descent converges to stationary points constrained to Σ (see Section 10). The lowering pass emits assertion graphs and Project$_{s}$igmaoperationssuchthatruntimeenforcementensuresanycommittedstateresp*

Human-Participant Interaction *The map ϕ gives human participants the ability to label, query, and constrain latent regions. Collaboration primitives include axis labeling, human-reviewed projection policies, and provenance-based audits that trace latent regions back to praxis and fused motifs.*

0.6 Implementation: Silicon Closure

The Cognitive Compiler lowers this closed loop to silicon-native representations.

Adjoint Graph Generation *When the compiler encounters the `@training` context, it perform Automatic Differentiation (AD) on the fused graph. It translates high-level reasoning intents into a dual-graph structure: 1. The **Primal Graph** (forward pass/inference). 2. The **Adjoint Graph** (backward pass/gradients).*

ONNX Training Extensions *We target ONNX Training IR, which supports the execution of gradient graphs on standard hardware. The compiler emits the dual graph (primal + adjoint) as a single ModelProto with separate graph entries for inference and training. Pragma metadata (`@training`, `@frozen`, `@train`) is stored in `metadata_props` and `doc_string` fields. During on-device updates, runtime checkers validate post-update invariants via the assertion graphs and perform semantic projections as part of the update kernel when required.*

Hardware-Side Enforcements *For edge or secure environments, we provide optional hardware-enforced guards that refuse to commit weight updates violating semantic constraints. These guards can be expressed as microcode or runtime policies in the device's firmware, enabling certified updates where regulatory or safety concerns demand absolute guarantees on post-update semantics.*

Verification Workflows *Formal verification is integrated into the build pipeline: the compiler emits assertion graphs and proof obligations alongside ModelProto artifacts. We use a layered strategy:*

1. *Lightweight runtime assertions encoded in ONNX graph nodes (Clip, Assert) that are executed during training and inference.*
2. *Post-update assertion graphs that verify semantic invariants and trigger rollback when violations are detected.*
3. *Off-line mechanized proofs (Coq) for projection operators and key optimizer invariants.*

Encoding Pragmas in ModelProto *Pragmas are encoded as follows:*

- `@training{loss:.., opt:..}` $\rightarrow$ *ModelProto.metadata_props[*`"training"`*] = serialized configuration.*
- `@train` *parameters* $\rightarrow$ *TensorProto attributes labeled* `trainable=1` *and recorded in* `metadata_props`*.*
- `@frozen` *subgraphs* $\rightarrow$ *Marked with* `frozen=1` *and excluded from adjoint generation and optimizer updates.*

Adjoint & Optimizer State *Optimizer state (moments for Adam, momentum buffers) is represented as auxiliary tensors in the ModelProto. Update kernels incorporate* `Project_Sigma` *operators immediately prior to write-back to parameter storage. This design minimizes the window during which invalid states exist and enables atomic commits guarded by hardware or software checks.*

Praxis Checklist: Safe Self-Modification *When a praxis allows reconfiguration, runtimes should implement the following minimal workflow to ensure safe deployable updates:*

1. ***Declare**: encode the reconfiguration* `scope` *and invariant set in the ModelProto metadata.*
2. ***Sandbox**: run candidate updates in an isolated environment using the same runtime checks and data distributions.*
3. ***Verify**: execute assertion graphs and test-suites; apply semantic projection operators where necessary.*
4. ***Audit**: log update artifacts and verification outcomes into the provenance graph for human review.*
5. ***Commit / Rollback**: atomically commit only when all checks pass; otherwise rollback and raise alerts.*

Safe Self-Modify (pseudocode)

```
candidate = propose_update(praxis, candidate_params)
trial_state = sandbox_run(candidate, budget)
if verify(trial_state, invariants):
    commit(candidate)
    record_provenance(candidate, outcome='committed')
```

```
else:
    rollback(trial_state)
    record_provenance(candidate, outcome='rejected')
```

0.7 Self-Modeling and On-Device Learning

We include a short reproducible example of on-device self-modeling where an agent learns a sensor calibration parameter online. The example demonstrates stability of the projection operators and the role of runtime assertion graphs in detecting and recovering from invalid updates. Detailed experiments and notebooks are available in the repository under ***papers/cognitive-closure/experiments/self_modeli***

Praxis for Self-Modeling *Every self-modifying praxis should explicitly declare a* ***reconfigure*** *policy and verification artifact. The* ***self_modeling_sensor*** *experiment follows the Praxis Checklist: candidate updates are evaluated in a sandboxed trial, verified with assertion graphs and a simple prediction-error reduction test, and recorded with provenance (figure + npz). The canonical loop used is:*

```
for t in timesteps:
  s = sense(g)
  pred = model.predict(s)
  loss = L(pred, target)
  model.update(loss)
  g = propose_and_verify_update(g, model, invariants)
```

This pattern generalizes to sensor (S), action (A), and computation (C) reconfiguration by extending the verification artifacts and sandbox budgets accordingly.

0.8 Formal Proofs and Formalization Roadmap

This section outlines the formalization artifacts accompanying Cognitive Closure and presents proof sketches for the principal claims.

Projection Properties

We include a Coq proof skeleton ***proofs/typed***$_p$*rojection.vthatestablishesthefollowinglemmasfortheboxprojectio*
(1)*boundedness*(*post*−*projectionlieswithinthedefinedinterval*), *and*(2)*non*−*expansiveness*(*projectionisnon*−
increasingindistancetoanyfeasiblepoint).*Theselemmasarecriticalbuildingblocksforprovingconvergenceof Proj*

Theorem 2 (Typed PGD Convergence Sketch)**.** *Let* Σ *be a closed, convex semantic constraint set and let* $\mathcal{L}$ *be a L-smooth function on the parameter space. Consider the typed projected gradient descent updates:*

$$\theta_{t+1} = \Pi_\Sigma(\theta_t - \eta \nabla \mathcal{L}(\theta_t)).$$

If $0 < \eta < 1/L$, *then the sequence* $\{\theta_t\}$ *satisfies*

$$\min_{0 \le t < T} \|\nabla_\Sigma \mathcal{L}(\theta_t)\|^2 = O\left(\frac{\mathcal{L}(\theta_0) - \mathcal{L}^*}{\eta T}\right),$$

where ∇_Σ *denotes the projected gradient and* $\mathcal{L}^*$ *is the infimum of* $\mathcal{L}$ *on* Σ.

Sketch. The proof follows standard PGD analyses (Nocedal Wright [?]). Key observations: projections onto closed convex sets are non-expansive, and the descent lemma for L-smooth functions yields a sufficient decrease per step when the step-size satisfies $\eta < 1/L$. Combining these with the non-expansiveness of Π_Σ yields the asymptotic bound above. Crucially, our Coq formalization targets the projection lemmas and the descent lemma to mechanize this sketch. □

Adjoint Correctness

We state a correctness criterion for the adjoint graph generated by the compiler.

Definition 3 (Adjoint Equivalence)**.** *Let G be the primal fused graph and G^* be the adjoint graph produced by the compiler via AD. G^* is adjoint-equivalent to G if, for any parameter perturbation $\Delta\theta$, the finite-difference approximation of the primal perturbation equals the adjoint evaluation:*

$$\lim_{\epsilon\to 0} \frac{\mathcal{L}(\theta + \epsilon\Delta\theta) - \mathcal{L}(\theta)}{\epsilon} = \langle \nabla_\theta \mathcal{L}(\theta), \Delta\theta \rangle = G^*(\Delta\theta).$$

Verification strategy *We verify adjoint correctness by combining symbolic differentiation of small motifs (e.g., LOF) with execution equivalence tests on the adjoint graph. Mechanized proofs will proceed by induction on the operator set and use equational reasoning to show that the AD rules preserve the inner-product characterization of directional derivatives.*

Complexity and Overhead

Generation of adjoint graphs increases IR size (roughly by a factor depending on operator arity and statefulness). Projection operations add a modest computational overhead (typically linear in parameter size for box/simplex projections). We argue that this overhead is justified by improved semantic safety and often reduced iteration counts due to constrained optimization.

Formal Artifacts

Present artifacts include:

- *`proofs/typed_projection.v`: Coq skeleton proving box projection lemmas and non-expansiveness.*
- *`proofs/adjoint_sketches.md`: notes and invariants for adjoint correctness proofs.*
- *RDF snippets encoding common semantic constraints for verifier consumption.*

0.9 Governance, Safety, and Human-in-the-Loop

Closing the cognitive loop introduces responsibilities. We outline governance patterns and safety mechanisms that ensure the Fused Fabric is auditable, safe, and aligned with human values.

Auditing and Traceability

Every update cycle preserves provenance information in RDF: which data samples were used, what optimizer produced the update, which projection operator was applied, and which assertion checks were executed. This enables ex-post-facto inspection and automated auditing pipelines.

Safety Policies

We recommend policy tiers for update acceptance:

- ***Soft Guard**: runtime assertion graphs execute post-update; violations trigger logging and alert but allow rollbacks.*
- ***Hard Guard**: firmware or secure enclave refuses updates that violate critical safety constraints (e.g., actuator bounds).*
- ***Human-in-the-Loop**: parameter updates flagged as semantically significant require manual approval before commit; this is enacted via a review API.*

Ethical Considerations

Because the latent space is now interpretable and manipulable, we define ethical guidelines for data provenance, bias auditing (ensure fairness constraints are expressible in RDF), and mechanisms for redress when models produce harmful outputs. The project provides templates for policy encoding in RDF to ensure machine-readable governance.

0.10 Conclusion: Reaching Cognitive Closure

Cognitive Closure marks the point where machine intelligence becomes a self-consistent, grounded, and auditable participant in reality. By closing the loop between the "What" (Typed Reality), the "How" (Compiled Cognition), and the "Why" (Learning in the Fused Fabric), we move beyond the era of untyped, ungrounded modeling.

The resulting "Formally Bound Latent Space" is not just a tool for optimization, but a medium for shared comprehension. It is the fabric upon which humans and machines will co-author the future of intelligence—where every decision is grounded, every reasoning step is fused, and every update is verifiably true to the fabric of reality.

We have moved from raw data to grounded cognition; from compiled intent to verified silicon; and finally, from stochastic drift to cognitive closure.

Broader Impacts *Cognitive Closure is intended to enable safer, more auditable AI systems. By constraining learning with ontological semantics and formal verification, we reduce the risk of unintended behaviors and make models more amenable to regulatory review. This work is a step toward building AI systems that can be deployed in safety-critical domains with strong guarantees about their behavior and provenance.*