Cognitive Compiler: Lowering Reasoning to Machine-Native IR

Anonymous

February 8, 2026

Abstract

As machine intelligence moves from monolithic models to complex reasoning pipelines, the gap between high-level cognitive intent and low-level machine execution (silicon) becomes a critical bottleneck. We introduce the *Cognitive Compiler*, a system that transforms high-level reasoning specifications into optimized, machine-native intermediate representations (IR), specifically targeting the ONNX standard. By treating reasoning as a first-class compilation target, we enable the formal fusion of heterogeneous compute graphs directly into verifiable, executable structures. This paper details the architecture of the Fuse-based cognitive IR, the graph-fusion strategies that preserve semantic invariants, and demonstrates the system's utility through the compilation of complex scoring and decision-making logic into performant silicon-ready code.

Contents

1 Introduction

The transition from "learning from data" to "reasoning from principles" requires a new layer in the computing stack: the Cognitive Compiler. While traditional compilers bridge source code to assembly, and neural compilers bridge model graphs to kernels, the Cognitive Compiler bridges *intent* to *operation*.

Current AI deployment relies on the manual translation of algorithmic logic into deep learning frameworks. This translation is prone to error, difficult to audit, and often results in opaque binaries where the original "reasoning" is erased. We propose a methodology where reasoning is expressed as the *fusion* of compute graphs, defined in a domain-specific language (DSL) called Fuse.

The compiler's core task is to take these disparate graph specifications and fuse them into machine-native representations, primarily the Open Neural Network Exchange (ONNX) format. This choice ensures that fused cognitive logic can run on any silicon target with a standard executor, from cloud GPUs to edge-based NPUs, while remaining bit-accurate across environments.

2 Cognitive Intermediate Representation (IR)

At the heart of the compiler is a multi-level Intermediate Representation. Unlike standard tensor IRs that focus on operator fusion and memory layout, the Cognitive IR prioritizes *semantic provenance*.

Fused Subgraphs as Units A fused subgraph is a self-contained unit of reasoning that preserves its identity through compilation. Whether it is a distance calculation, a Bayesian update, or a symbolic search, it is represented as a coherent graph structure.

Type-Safe Graph Fusion The IR maintains strict type safety across the fusion boundaries, referencing the DType and Shape of inputs/outputs throughout the process. By the time the IR reaches ONNX level, every operator is bound to a specific opset version, ensuring deterministic behavior across fused domains.

Contextual Metadata The IR carries rich contextual metadata, encoded via the '@' pragma syntax, which links each computational unit to its semantic origin, provenance, and intended use. This metadata enables bidirectional traceability between the IR and the source cognitive algorithm, supporting tasks such as debugging, auditing, and interpretability. By embedding information such as motif lineage, data source, and transformation history, the IR facilitates transparent reasoning about how each fused subgraph was constructed and why specific design choices were made.

3 Fuse Syntax: Expressing Intent

The Fuse DSL provides a human-readable interface to the cognitive compiler. It combines the declarative nature of a graph specification with the imperative logic of localized transformations.

Declarations Programs begin with metadata declarations ('@fuse', '@opset') that configure the compiler's backend.

Graph Definition Graphs define the macro-structure of the reasoning pipeline. They use a typed-edge syntax reflecting the dataflow between internal components and external ports.

Constants and Parameters The language distinguishes between ‘const‘ (fixed at compile time) and ‘param‘ (dynamically bound or trained), allowing the compiler to perform aggressive constant folding on reasoning logic (e.g., pre-calculating distance references).

4 Case Study: Compiled LOF Scoring

To demonstrate the compiler's capability, we examine the Local Outlier Factor (LOF) inspired scoring mechanism, a classic reasoning pattern for novelty detection.

```
# LOF-inspired score comparing sample distance to a reference
# Ops: Sub, Abs, ReduceSum, Div, Reshape

@fuse 0.7
@opset onnx 18
@version 0.7.0
@domain jupyter.cookbook

const axes0: i64[1] = [0]

@type "urn:jupyter.cookbook.lof_score"

graph lof_score(x: f32[3]) -> f32[1] {
  center: f32[3] = [0.0, 0.0, 0.0]
  reference: f32[3] = [0.1, -0.1, 0.0]
  shape_one: i64[1] = [1]
  dist = ReduceSum(Abs(Sub(x, center)), axes0, keepdims@=0)
  ref_dist = ReduceSum(Abs(Sub(reference, center)), axes0, keepdims@=0)
  ratio = Div(dist, Add(ref_dist, 1.0))
  Reshape(ratio, shape_one)
}

@proof graph test_lof_score() {
  sample = [1.0, 0.0, 0.0]
  out = lof_score(sample)
  assert out == [0.8333333]
}
```

Analysis of compiled logic The compiler identifies that ‘ref$_d$ist‘*involvesonly*‘*const*‘*inputs*(‘*reference*‘, ‘*center*‘

5 Lowering Logic: From Graph to Node

The lowering process is a recursive transformation that maintains the identity of cognitive units.

Graph Fusion and Inlining Each graph reference is fused into the caller's graph. If a reasoning step refers to another graph, the compiler performs inlining or subgraph generation based on the target runtime's capabilities, optimizing away redundant intermediate tensors.

Operator Mapping Standard operators (‘Sub‘, ‘Abs‘, ‘Div‘) are mapped directly from the fused high-level representation to their ONNX equivalents. The compiler ensures that attributes (such as ‘keepdims‘ or ‘axis‘) are correctly formatted according to the targeted ‘@opset‘.

Training Lowering When a training context is declared, the lowering pass generates an adjoint graph. This graph explicitly models gradient propagation, optimizer state (e.g., momentum, Adam moments), and semantic projection steps. The compiler emits specialized primitives such as ‘Project$_{S}igma$‘$toensurethatupdatesremainsemanticallyvalidpost-update$.

Formal Guarantees We formally assert that the lowering preserves semantic invariants: if the fused graph satisfies the ‘FUSE‘ predicate with grounding σ, then both the primal and adjoint graphs preserve σ under optimization steps that include ‘Project$_{S}igma$‘.

Verification Generation The ‘@proof‘ blocks are lowered into a side-car validation graph. This graph executes alongside the model during testing to ensure the compiled silicon accurately reflects the intended high-level logic.

6 Training Extensions to the Cognitive Compiler

The Cognitive Compiler is extended to support training contexts via structured pragmas and generation of adjoint graphs.

6.1 Pragma Semantics

We formalize three pragmas that influence the compiler's behavior when building training graphs:

- `@training{loss: expr, opt: alg}`: declares a training context, desired loss and optimization algorithm (e.g., SGD, Adam). Compilers instantiate gradient nodes and optimizer subgraphs.
- `@frozen`: marks subgraphs and constants that the adjoint graph must not update; enforced both at compile and runtime.
- `@train param`: annotates parameter tensors as optimization targets. The compiler emits update nodes guarded by type-safe projection steps.

6.2 Adjoint Generation and Semantic Projection

The compiler performs automatic differentiation on the fused graph to generate an adjoint (backward) graph. Crucially, each gradient is accompanied by a *semantic projection* that ensures the update respects the parameter's semantic constraints (normalization, range, vocabulary bounds). Practically, this generates an extra operator chain following the gradient: `Grad -> Project_Semantics -> Update`.

6.3 Verification, Assertion Graphs, and Artifacts

Training graphs are accompanied by assertion graphs that validate semantic invariants post-update (e.g., probability normalization, physical range adherence). These assertion graphs can be run as checks during training iterations or compiled into continuation checks for on-device updates.

We provide accompanying artifacts for reproducing and verifying these behaviors:

- A Jupyter notebook (`notebooks/lof_training.ipynb`) demonstrating forward, adjoint, and typed projected SGD on th

6.4 LOF Example (Training Variant)

We extend the LOF example to show a training scenario where the reference vector is refined from observed benign samples while keeping sensor-grounding frozen.

- ```
  @fuse 0.7
  @opset onnx 18
  @version 0.7.0
  @domain jupyter.cookbook

  const axes0: i64[1] = [0]
  @train weight  reference: f32[3] [0.1, -0.1, 0.0]  # learnable reference

  @type "urn:jupyter.cookbook.lof_score"

  graph lof_score(x: f32[3]) -> f32[1] {
    center: f32[3] = [0.0, 0.0, 0.0]
    shape_one: i64[1] = [1]
    dist = ReduceSum(Abs(Sub(x, center)), axes0, keepdims@=0)
    ref_dist = ReduceSum(Abs(Sub(reference, center)), axes0, keepdims@=0)
    ratio = Div(dist, Add(ref_dist, 1.0))
    Reshape(ratio, shape_one)
  }

  @training { loss: out, optimizer: Adam, lr: 1e-3 }
  @proof graph train_lof() {
    sample = [0.9, 0.0, 0.0]
    label = [0.0]  # training to reduce false-positive on benign sample
    out = lof_score(sample)
    assert out <= 0.5
  }
  ```

7 Silicon-Native Execution

By targeting the machine-native ONNX format, the Cognitive Compiler bridges the gap between reasoning and hardware.

Universal Backend Support The compiled artifacts can be executed on any standard runtime, including ONNX Runtime, OpenVINO, and CoreML. This eliminates the need for specialized "cognitive hardware" and allows reasoning to scale with standard GPU/NPU performance.

Weights as Governed Assets Weights and learned parameters are valuable assets: they can be costly to produce, proprietary, or subject to regulatory or contractual use constraints. The Cognitive Compiler treats weights as external artifacts referenced by IRIs with attached metadata (provenance, licensing, cost, and access-control policies). Before composing or deploying a fused fabric, the compiler can (1) query availability and license compatibility, (2) plan alternative
  ```

computation when weights are unavailable (e.g., untrained transforms, distilled approximations, or hardware-safe replacements), and (3) annotate ModelProto with weight-asset metadata for runtime enforcement and auditing.

Deterministic Low-Level Logic Because every operation is statically typed and lowered to exact machine ops, the execution is bit-deterministic. This is critical for industrial applications where "black box" behavior is unacceptable and every decision must be repeatable.

Human Oversight and Machine Reasoning The design separates algorithmic structure (the transform graph) from the parameter assets (weights), enabling independent governance: humans can audit and approve computation pipelines, while automated reasoners can verify license constraints, check semantic compatibility, and suggest alternative compositions when necessary. Together, this enables mixed human-machine workflows where policy, provenance, and performance trade-offs are explicitly modeled in the compilation pipeline.

8 Formal Grammar of the Fuse-IR

The syntax for the cognitive compiler is defined by a strict grammar that enforces the separation of metadata and logic.

```
File ::= { Pragma } { Decl }

Pragma ::= "@" Ident [ Value ]
Decl   ::= ConstDecl | GraphDef | ProofGraph

ConstDecl ::= "const" Ident ":" Type "=" Value

GraphDef ::= "graph" Name "(" [ Params ] ")" "->" Type "{"
               { Statement }
             "}"

Statement ::= Binding | OpCall
Binding   ::= Ident ":" Type "=" Value
OpCall    ::= Ident "=" Operator "(" [ Args ] ")"

Operator ::= "Sub" | "Abs" | "ReduceSum" | "Div" | "Reshape" | ...

ProofGraph ::= "@proof graph" Name "(" ")" "{"
                 { Statement }
                 "assert" Expression
               "}"
```

9 Conclusion

The Cognitive Compiler represents a foundational shift in how we build and deploy machine intelligence. By elevating reasoning to a first-class citizen of the compilation pipeline, we achieve the level of auditability and performance required for the next generation of AI systems. High-level

intent is no longer "black-box" weights; it is a verifiable sequence of fused compute graphs that lower directly to silicon via standard, machine-native IRs.

Moving forward, this framework enables the creation of a "fused fabric" of intelligence, where formal reality models (Typed Reality) and the compilers that map fused compute to hardware (Compiled Cognition) work in concert to build fully auditable, performant, and grounded AI.