Statically Typed Reality: A Type-Preserving ABI for Machine Intelligences

Anonymous

February 7, 2026

Abstract

We present a statically typed ABI and abstract syntax tree (AST) that bridges observation and computation: sensors, symbolic knowledge, and probabilistic models are unified as typed tensors, fused through type-preserving primitives, and lowered to any runtime (ONNX, GPU kernels, hardware accelerators, or domain-specific interpreters). Each tensor is a triple—*DType, Shape, Semantics*—that grounds computation in measurable reality and abstract knowledge. This enables deterministic interpretation of the world, where perception (vision, audio, language), reasoning (distributed, probabilistic, symbolic), and action (motor, communication, control) are formally connected. Test and train graphs are identical in structure and semantics, enabling reproducible evaluation and formal validation. Compile-time verification rejects type-unsafe programs; where checks remain dynamic, explicit assertions are generated and elided when provably safe.

Contents

1 Introduction

Machine intelligence operates at the boundary between perception and abstraction. Sensors observe the physical world; models compute over symbolic and probabilistic abstractions; actuators effect change in reality. Yet current frameworks treat tensors as untyped arrays, creating a semantic gap: the mapping from observations to computation, through reasoning, to action remains implicit. Runtime errors, shape mismatches, and silent semantic drift are endemic.

The core problem: How do we ground computation in reality while fusing heterogeneous knowledge domains (visual, linguistic, probabilistic, symbolic) into coherent reasoning?

We present a statically typed ABI (Application Binary Interface) and AST (Abstract Syntax Tree) for machine intelligence that bridges this gap. The ABI maps observations—from sensors, symbolic data, and probabilistic models—onto a unified, type-safe representation. Each tensor is ground-truth linking reality to computation:

- **DType**: numeric or categorical precision
- **Shape**: constant, symbolic, or ranged dimensions
- **Semantics**: measurement (physical reality), distribution (probabilistic knowledge), enum (symbolic knowledge), or structured (composite)

The AST comprises typed primitives (Linear, Attention, Softmax) that preserve semantics through composition. The ABI lowers to *any* runtime—ONNX, custom GPU kernels, CPU optimizers, or specialized accelerators—ensuring semantic preservation regardless of execution target. This enables deterministic interpretation of reality where every sensor read, every reasoning step, and every actuator command is formally verified and auditable.

2 Type System

Traditional type systems (Hindley-Milner [Hindley, 1969], System F [Girard, 1972]) encode properties of computation. Dependent types [Team, 2023b,a] allow types to encode propositions about values. We extend this paradigm: types encode properties of *meaning*. A Measurement type is not merely a numeric constraint but a statement about grounding in physical reality.

2.1 Core Tensor Type

A tensor is a triple:

$$\text{Tensor}\langle \text{DType}, \text{Shape}, \text{Semantics} \rangle$$

where DType $\in \{\texttt{float32}, \texttt{int32}, \texttt{bool}, \ldots\}$ (numeric precision), Shape is determined by dimension list (constant, symbolic, or ranged), and Semantics grounds meaning in observable reality or abstract knowledge.

2.2 Semantic Domains

- **Atomic**: unstructured numeric substrate; generic latent vectors without semantic commitment
- **Measurement**: grounds tensors in physical reality via IRIs—radiance (`qudt:Radiance`), pressure (`qudt:Pressure`), temperature (`qudt:Temperature`)—with unit, valid range, and observational frame (sensor IRI). Follows SOSA/SSN [Haller et al., 2017] and QUDT [Hodgson et al., 2023] standards.

- **Distribution**: encodes probabilistic knowledge: unnormalized (logits over support), normalized (probabilities), or samples from a stochastic process (IRI reference to Gaussian, Poisson, etc.)
- **Enum**: grounds tensors in symbolic knowledge via SKOS concept schemes [Miles and Bechhofer, 2009]: discrete vocabulary (BPE tokens, class labels, entity IRIs) with fixed cardinality
- **Structured**: composite types combining Measurements, Distributions, and Enums; enables fusing heterogeneous knowledge (e.g., visual features + textual embedding + pose measurement)

Key principle: Semantics are *not* erased. They flow through the computation graph and are enforced at type check and lowering time. A tensor's semantic grounding is invariant: a Measurement remains grounded in physical reality; a Distribution preserves probabilistic constraints; an Enum maintains symbolic vocabulary. This enables the system to detect semantic mismatches (e.g., attempting to average class indices) that would silently corrupt untyped systems like PyTorch [Paszke et al., 2019] or TensorFlow [Abadi et al., 2016].

2.3 Subtyping and Compatibility

Edge compatibility enforces the fusion of knowledge realms. When connecting two primitives, the system checks:

1. **DType match**: no implicit numeric casts; float32 fuses with float32, not int32
2. **Shape unification**: symbolic dimensions refined, ranges narrowed; no shape conflict reaches runtime
3. **Semantic subsumption**: Measurements match in frame and unit; Distributions normalize correctly; Enums share vocabulary or use explicit mappings; Structured types align field-by-field

Example of semantic fusion: A vision encoder outputs `image_emb : Tensor<float32,[B,512],Atomic»`. A language model outputs `text_emb : Tensor<float32,[B,512],Enum vocab:BPE50k »`. Direct concatenation is rejected by the type system because Atomic and Enum semantics are incompatible. An explicit bridge primitive must be used: `Embed(text) -> Tensor<float32,[B,512],Atomic»`, which projects symbolic knowledge (tokens) into latent space. Only after this projection can the embeddings be fused.

Implicit unit conversion, enum widening, or dimension mismatch is rejected. This preserves auditability: every knowledge realm crossing is explicit and auditable.

3 Sensors as Data Sources

Sensors are typed producers. The type specifies what semantic information the data carries, enabling deterministic mapping to computation.

- **CameraRGB**: Tensor<float32,[B,C,H,W],Measurement{radiance,unit:srgb_8bit,frame:camera}> Converts pixel radiance to normalized $[0, 1]$ RGB.

- **Microphone**: Tensor<float32,[B,F,T],Measurement{amplitude,unit:log_mel,frame:acoustic}> Mel-spectrogram: log-energy features over frequency/time.
- **Tokenizer**: Tensor<int32,[B,T],Enum{vocab:BPE_50k}> Text to token indices.
- **ClassificationLabel**: Tensor<int32,[B],Enum{vocab:ImageNet1k}> Train-time: ground-truth class index.

Sensor bindings are explicit: `bind CameraRGB.output → ConvStem.input` enforces that the downstream primitive expects a Measurement with matching unit and frame. Types guarantee *semantic coherence*—every operation respects the data's origin and meaning—but not physical truth. A malfunctioning sensor will produce a tensor of the correct type with incorrect content; the type system ensures correct composition regardless.

4 Typed Primitives

Primitives are type-preserving transformations. Each primitive declares its input and output types; the type checker ensures compatibility at every edge.

Linear Transformation

```
primitive Linear {
  input:  X : Tensor<float32, [B,T,D], Atomic>
  output: Y : Tensor<float32, [B,T,D_out], Atomic>
  weight: W : Tensor<float32, [D, D_out], Atomic>
}
```

Multi-Head Attention

```
primitive Attention {
  input:  Q,K,V : Tensor<float32, [B,H,T,D], Atomic>
  output: O : Tensor<float32, [B,H,T,D], Atomic>
  constraint: dims_match(Q,K,V) and H*D = emb_dim
}
```

Distribution-Aware Softmax

```
primitive Softmax {
  input:  logits : Tensor<float32, [B,C], Atomic>
  output: probs : Tensor<float32, [B,C],
           Distribution{categorical, range:[0,1]}>
  constraint: sum of probs[b,c] over c = 1.0
}
```

Key insight: semantics are explicit. Softmax output is *not* merely normalized values—it is *typed* as a Distribution, enabling downstream primitives (Sample, KL divergence, cross-entropy loss) to enforce semantic correctness. This makes stochastic and supervised pipelines verifiable.

5 Fusing Knowledge Realms: From Observation to Action

A complete pipeline from observation to action is a DAG where every edge carries a tensor type. The graph fuses three distinct realms of intelligence: *perception* (grounded measurement), *knowledge* (symbolic and probabilistic), and *reasoning* (deterministic and stochastic computation).

Perception Realm: Grounding in Physical Reality

- **CameraRGB** → `Tensor<float32,[B,3,H,W],Measurement radiance,unit:srgb,frame:camera`
  »—raw pixels are physical measurements
- **Microphone** → `Tensor<float32,[B,F,T],Measurement amplitude,unit:log_mel,frame:acoustic`
  »—audio is acoustic measurement
- **Tactile Sensor** → `Tensor<float32,[B,N],Measurement pressure,unit:pascals,[0,100k]`
  »—touch is force measurement

All perception inputs are Measurement types: they ground the computation in physical reality with explicit units, frames, and valid ranges. A malfunctioning sensor produces a Measurement of the correct type with wrong content—the type system ensures this is processed consistently and fails explicitly if constraints are violated.

Knowledge Realm: Symbolic and Probabilistic Abstraction

- **Tokenizer** → `Tensor<int32,[B,T],Enum vocab:BPE50k`
  »—raw text as symbolic vocabulary
- **Ontology/KG** → `Tensor<int32,[B,E],Enum entities:DBpedia`
  »—structured knowledge as symbols

Symbolic knowledge is typed as Enum: discrete, finite vocabulary. Probabilistic knowledge (language model output, belief states) is typed as Distribution: normalized over a support space.

Reasoning Realm: Fusing Knowledge Through Type-Preserving Primitives **Embedding projection**: Bridge symbolic knowledge into latent space:

- `TextEmbed`: Enum
  vocab
  → Atomic (project tokens to embeddings)
- `EntityEmbed`: Enum
  entities
  → Atomic (project knowledge graph entities to embeddings)
- `MeasurementNorm`: Measurement (with frame/unit) → Atomic (normalize physical quantities to standard ranges)

Fusion across realms:

$$\text{X} = \text{Concat}([\text{text\_emb}, \text{vision\_emb}, \text{entity\_emb}, \text{tactile\_emb}], \text{axis} = 1)$$

All inputs to Concat are now Atomic (latent, dimensionless). The type system ensures shape alignment and semantic compatibility. Mixing a Measurement directly with a symbolic Enum is impossible—all knowledge realms must first be projected to a common latent space.

Integrated reasoning: Apply N transformer blocks to fused representation. Each block composes primitives that preserve semantic structure:

$$\text{X'} = \text{Attention(X)} \rightarrow \text{Atomic}$$

$$\text{X"} = \text{Linear(LayerNorm(X'))} \rightarrow \text{Atomic}$$

$$\text{logits} = \text{Linear(X")} \rightarrow \text{Atomic}$$

The output logits are still Atomic—pure latent vectors. No semantic information flows from this layer; it is deterministic (no Distribution) because reasoning over fused knowledge is deterministic until explicitly stochastic.

Decision and action: Bridge reasoning output back to action in physical and symbolic realms:

$$\text{action\_logits} = \text{Linear(X")} \rightarrow \text{Atomic}$$

$$\text{action\_probs} = \text{Softmax(action\_logits)} \rightarrow \text{Distributioncategorical}$$

Actuators bind to Distribution or Measurement outputs:

- **Motor**: Sample from Distribution, e.g., action $\sim$ action_probs[“left”, “straight”, “right”]
- **Controller**: Project back to physical Measurement, e.g., `SpeedControl(speed_logits) -> Measurement speed,unit:m/s,[0,10]`, then enforce via Clip assertion

Semantic invariant: Every edge in the graph is typed. Perception (Measurement) grounds input. Knowledge (Enum, Distribution) enters via symbolic/probabilistic projections. Reasoning fuses and transforms Atomic representations. Decision outputs (Distribution) drive stochastic action; Motor control (Measurement) enforces physical safety. The type system rejects (at compile time) any crossing of knowledge realms without explicit type bridge primitives.

6 Semantic-Preserving Lowering to Runtime

The typed AST is lowered to any target runtime (ONNX IR, custom GPU kernels, CPU code, hardware accelerators, or domain-specific interpreters) while preserving semantic invariants. Assertions are explicit; runtime checks are elided only when provably safe; modality semantics are preserved in metadata for downstream auditing and validation.

Type → Runtime Mapping

Typed Tensor	Runtime Encoding	Assertions
Measurement	Tensor + metadata	Clip, Assert on range and unit
Distribution	float32/float64 tensor	*normalization*; constraint is semantic
Enum	int32 tensor	Assert bounds and vocabulary
Structured	Multi-tensor or packed encoding	Coordinate semantics

ONNX as default target: ONNX IR provides an extensible format for intermediate representation. Tensors are lowered to ONNX tensors with semantic metadata stored as attributes or external schema references. Assertions (on Measurement ranges, Enum bounds, Distribution normalization) are encoded as ONNX op graphs (Clip, Greater, Assert nodes).

Alternative runtimes: The same typed AST can lower to GPU-optimized code (CUDA kernels with semantic metadata in comments), CPU code (with explicit runtime checks inlined), or specialized accelerators (custom instruction sequences with semantic annotations).

Assertion Elision and Symbolic Analysis Compile-time shape and range analysis determines which runtime assertions are redundant:

- If range $[0, 1]$ is guaranteed by prior Softmax, subsequent Clip is elided.
- If Enum bounds are verified statically, vocabulary check is removed.
- If symbolic dimension is never refined, shape Assert remains at runtime.

Semantic Preservation and Auditability Regardless of target runtime, the lowered code carries semantic annotations for every tensor:

- **Auditing**: trace back any output to its sensor input and all intermediate semantic types.
- **Test/Train Consistency**: graph topology and type constraints are deterministic and runtime-agnostic; train and test graphs share identical lowered semantics.
- **Model Debugging**: when output is unexpected, the semantic types and assertions narrow culprit to a specific primitive, knowledge realm crossing, or measurement constraint violation.
- **Runtime Flexibility**: same ABI/AST can be deployed on different hardware (edge device with lightweight interpreter, cloud GPU with ONNX Runtime, custom accelerator with custom lowering) without changing the semantic contract.

7 EBNF Grammar of the AST

The type-safe ABI is defined by a formal grammar. Tensors, primitives, and graphs are syntactic objects subject to type checking before lowering.

7.1 Type Grammar

```
Type ::= TensorType | PrimitiveType | GraphType

TensorType ::= Tensor< DType, Shape, Semantics >
```

```
DType ::= float16 | float32 | float64 | int8 | int32 | int64 | bool

Shape ::= Dimension+
Dimension ::= Const(n) | Symbol(name) | Range(min, max)

Semantics ::= Atomic
           | Measurement{phys_qty, unit, frame, [min, max]}
           | Distribution{type, support}
           | Enum{vocab_name, cardinality}
           | Structured{field_1: Type, ..., field_n: Type}

PrimitiveType ::= primitive PrimName {
  input: name_1: Type, ..., name_m: Type
  output: name_1: Type, ..., name_k: Type
  constraint: BoolExpr+
}

GraphType ::= DAG with typed edges;
  each edge: (output_port, input_port) typed as Type
```

7.2 Primitive Grammar

Primitives are morphisms with declared input/output types and constraints:

```
Primitive ::= primitive Name {
  input: [var: Type, ...]
  output: [var: Type, ...]
  weight: [var: TensorType, ...]
  constraint: [expr, ...]
}

Constraint ::= shape_match(var1, var2)
            | dtype_match(var1, var2)
            | semantic_subsumption(sem1, sem2)
            | range_fit(var, [min, max])
            | norm_check(var)  // Distribution normalization
```

7.3 Graph Grammar

```
Graph ::= name: {
  inputs: [port: Type, ...]
  outputs: [port: Type, ...]
  nodes: [Node, ...]
  edges: [Edge, ...]
}

Node ::= id: Primitive | Graph

Edge ::= source=(node_id, port) -> target=(node_id, port)
```

```
       : Type  // Type annotation on edge

// Typing rule: source output type must be compatible with
// target input type under subtyping relation
```

8 Semantic Foundations and RDF Grounding

Semantics are not merely informal annotations—they are first-class objects in the type system. We ground semantics in RDF (Resource Description Framework) and IRIs (Internationalized Resource Identifiers), establishing a global namespace for meaning.

8.1 Semantic Types as RDF Entities

Each semantic class (Measurement, Distribution, Enum, Structured) is instantiated as an RDF class, enabling interoperability and formal reasoning:

```
Measurement rdf:type owl:Class
  rdfs:subClassOf Semantic
  dcterms:hasPhysicalQuantity IRI
  qudt:hasUnit IRI
  sosa:hasResultTime xsd:dateTime
  sosa:isObservedBy IRI  // sensor/observer IRI
  sosa:madeBySensor IRI
```

Example instantiation:

```
ex:camera_radiance rdf:type Measurement
  ex:quantityOf ex:Radiance
  qudt:unit qudt:Watt-Per-Steradian-Per-Meter_Squared
  sosa:madeBySensor ex:FrontCamera
  ex:frame ex:CameraFrame_320x240
```

8.2 Enum Semantics: Symbolic Knowledge as Controlled Vocabularies

Enum types map to SKOS (Simple Knowledge Organization System) concept schemes:

```
ex:BPE50kVocab rdf:type skos:ConceptScheme
  dcterms:title "BPE 50k Vocabulary"
  skos:hasTopConcept [multiple BPE token IRIs]

ex:ImageNet1kClasses rdf:type skos:ConceptScheme
  dcterms:title "ImageNet 1000 Classes"
  skos:hasTopConcept [1000 class IRIs]
```

Each token/class is an IRI pointing to semantic meaning in a knowledge graph (WordNet, DBpedia, or domain-specific ontologies).

8.3 Distribution Semantics: Probabilistic Grounding

Distribution types carry semantic information about what is being modeled:

```
Distribution rdf:type owl:Class
  ex:overId IRI          // IRI of entity being modeled
  ex:supports IRI        // support space IRI
  ex:isNormalized xsd:boolean
  ex:hasStochasticProcess IRI  // Poisson, Gaussian, etc.
```

Example:

```
ex:action_distribution rdf:type Distribution
  ex:overId ex:DiscreteActions
  ex:supports ex:ActionSpace_3ways
  ex:isNormalized true
  // Softmax output over {left, straight, right}
```

8.4 The Fabric of Meaning: Semantic Graph Structure

Semantics form a graph structure (the "fabric of meaning") orthogonal to the computation graph:

- **Nodes**: RDF entities (measurements, symbols, distributions, physical quantities)
- **Edges**: RDF predicates ("unit of", "instance of", "hasStochasticProcess", "observedBy")
- **Namespace**: Global IRIs enable linking across systems and domains

When a primitive transforms a tensor, it must preserve semantic coherence: if input is a Measurement with "unit: Kelvin", output must either remain Kelvin or use an explicit conversion semantic (e.g., to Celsius via owl:hasConversion).

This fabric enables:

- **Semantic interoperability**: two systems using same IRIs can verify semantic compatibility
- **Ontology-driven verification**: reason over semantic constraints (e.g., cannot mix Celsius and Kelvin without conversion)
- **Formal grounding**: semantics are not strings but globally resolvable IRIs with formal definitions

9 Formal Semantics and Graph Theory

9.1 Denotational Semantics of Typed Tensors

A typed tensor is interpreted as a pair: value and semantic grounding.

$$[\![\text{Tensor}\langle D, S, \Sigma\rangle]\!] = (v, \sigma)$$

where $v \in \mathbb{R}^D$ (the value) and $\sigma \in$ SemanticSpace (the meaning).

Semantic spaces:

- Atomic : $\emptyset$ (no semantic constraint)

- Measurement : $(\text{PhysicalQuantity}, \text{Unit}, \text{Frame}) \times [a, b]$ (grounded in physical reality with bounds)
- Distribution : $(\text{Entity}, \text{StochasticProcess}, \text{Support})$ (probabilistic model of an entity)
- Enum : $\text{ConceptScheme} \times \mathbb{N}$ (symbolic vocabulary with cardinality)
- Structured : $\prod_i \text{SemanticSpace}_i$ (product of semantic spaces)

A primitive P is a morphism that preserves semantic structure:

$$P : (v_1, \sigma_1) \times \cdots \times (v_n, \sigma_n) \to (v'_1, \sigma'_1) \times \cdots \times (v'_m, \sigma'_m)$$

Semantic preservation constraint: P must satisfy $\sigma'_j \sqsubseteq \sigma_j$ (semantic subsumption) or involve an explicit type bridge.

9.2 Graph-Theoretic Model of Computation

A computation graph $G = (N, E, \tau)$ where:

- N is a set of primitive nodes
- $E \subseteq N \times N$ are directed edges (data flow)
- $\tau : E \to \text{Type}$ assigns a type to each edge

The graph must satisfy:

1. **Well-typedness**: for each edge $(u, v) \in E$, output type of u is compatible with input type of v
2. **Acyclicity**: no cycles (data flow is a DAG)
3. **Semantic closure**: semantic types form a lattice under subsumption; each edge type is a lattice element

9.3 Three Orthogonal Graph Layers

A statically typed system actually operates over three intertwined graphs:

1. Computation Graph The data flow DAG: $G_{\text{comp}} = (N_{\text{prim}}, E_{\text{data}})$ where edges carry tensor types. This is what executes at runtime.

2. Type Graph The constraint system over types: $G_{\text{type}} = (N_{\text{type}}, E_{\text{subtype}})$ where nodes are types and edges are subsumption relations. The system checks that every data edge respects type constraints.

3. Semantic Graph The RDF/ontology layer: $G_{\text{sem}} = (N_{\text{iri}}, E_{\text{predicate}})$ where nodes are IRIs (physical quantities, knowledge concepts, stochastic processes) and edges are RDF predicates. This ground-truth graph describes what meanings the computation is over.

Invariant: For every edge $e \in E_{\text{data}}$ with type $\tau(e)$, the semantics of $\tau(e)$ must resolve via G_{sem}. If $\tau(e) = \text{Measurement}\{\text{velocity}, \text{unit:m/s}\}$, then velocity and m/s must be IRIs resolvable in G_{sem}.

9.4 The Fabric of Reality, Knowledge, and Computation

These three graphs form a unified fabric:

- **Reality Fabric** (G_{sem} layer): the ontology of physical quantities, measured by sensors
- **Knowledge Fabric** (G_{sem} layer): the vocabulary of symbols and concepts (tokens, entities, relations)
- **Computation Fabric** (G_{comp} layer): the neural network DAG that transforms observations
- **Type Fabric** (G_{type} layer): the constraints ensuring safe composition

These fabrics are not separate—they are woven together. A Measurement tensor carries both: - A value (computation fabric): the pixel values from a camera - A semantic IRI (reality fabric): the physics quantity "radiance" with unit "sRGB"

A Distribution tensor carries: - A value (computation fabric): probabilities summing to 1 - A semantic IRI (knowledge fabric): the decision space it models

When primitives compose, all three fabrics must align. This is why semantic typing is powerful: it enforces alignment across all three simultaneously.

10 Related Work

10.1 Type Systems and Dependent Types

Type systems in programming languages (Hindley-Milner [Hindley, 1969], System F [Girard, 1972]) provide shape and kind discipline. Recent work on dependent types (Coq [Team, 2023b], Agda [Team, 2023a]) enables richer specifications where types encode propositions. Our semantic types extend this paradigm: instead of properties of computation, they encode properties of *meaning*. A Measurement type is not a computability statement but a grounding statement: "this tensor represents physical reality."

Array languages (Futhark [Henriksen et al., 2016], Dex [Paszke et al., 2022]) provide shape polymorphism and compile-time shape inference, but treat arrays as untyped numeric substrates. They lack semantic distinction for physical quantities, distributions, or symbolic data. Julia's type system [Bezanson et al., 2017] is flexible but permissive; no constraint on semantic meaning flows through composition.

10.2 Tensor Type Systems and Neural Language Design

TensorFlow's shape inference [Abadi et al., 2016] and PyTorch's type hints [Paszke et al., 2019] provide partial static guarantees but remain permissive: dynamic shapes, implicit semantic assumptions, no end-to-end verification. Glow [Rotem et al., 2020], a compiler for neural networks, performs shape inference and code generation but lacks semantic types.

More recent work on certified neural networks (VNN Abstr. [Cohen and Welling, 2020], DeepTest [Tian et al., 2018]) focuses on robustness properties (adversarial bounds) rather than semantic grounding. Our semantic types are orthogonal to robustness but enable semantic verification at a higher level than floating-point bounds.

10.3 RDF, Ontologies, and Semantic Web

RDF [Cyganiak et al., 2014], OWL [Group, 2012], and SKOS [Miles and Bechhofer, 2009] provide formal frameworks for describing meaning via IRIs. QUDT [Hodgson et al., 2023] (Quantities, Units, Dimensions, Types) defines ontologies for physical units and dimensions. SOSA/SSN [Haller et al., 2017] (Sensor, Observation, Sample, and Actuator) is a W3C standard for sensor data and observations.

Our contribution is to embed this semantic infrastructure *into the tensor type system*. Rather than treating semantics as external metadata, we make them first-class type properties that drive compilation and verification. This bridges formal ontology languages with neural computation.

10.4 Domain-Specific Languages for Neural Computation

Halide [Ragan-Kelley et al., 2013] provides scheduling and optimization for image pipelines, but lacks neural primitives. Darkroom [Hegarty et al., 2013] uses dependent types for image processing but is not neural-focused. TVM [Chen et al., 2018] is a compiler for diverse backends (CPU, GPU, TPU) but operates on untyped tensor IRs.

MLIR [Lattner et al., 2021] provides a framework for intermediate representations and lowering. Our work can be viewed as a domain-specific dialect of MLIR, where the semantic type system is a core abstraction.

10.5 Knowledge Graphs and Neurosymbolic Reasoning

Knowledge graphs (Freebase [Bollacker et al., 2008], DBpedia [Lehmann et al., 2015], Wikidata [Vrandečić and Krötzsch, 2014]) enable structured reasoning over symbolic knowledge. Neurosymbolic AI (Mao et al. [Mao et al., 2019], Garnelo and Shanahan [Garnelo and Shanahan, 2019]) attempts to integrate neural and symbolic reasoning.

Our semantic fabric (Section 9) connects neural tensors directly to KG IRIs, enabling compositional verification that fuses neural and symbolic reasoning without ad-hoc glue code. When an Enum tensor references a SKOS vocabulary, the type system can verify that operations respect vocabulary constraints.

10.6 Formal Semantics of Machine Learning Systems

Recent work on formal specification of ML systems (Uchitel et al. [Uchitel and Zussman, 2021], Heil et al. [Heil et al., 2022]) proposes formal models of training dynamics and generalization. Our approach is complementary: we focus on *graph-level* semantics (what computation means) rather than *statistical* semantics (what generalization guarantees hold).

10.7 Verified Compilers and Certified Code Generation

CompCert [Leroy, 2023], a fully verified C compiler, proves correctness of code generation. CakeML [Kumar et al., 2014] extends this to functional languages. Our work borrows the philosophy—semantics preservation through lowering is verified or at least auditable—but applies it to neural computation and semantic (not just syntactic) correctness.

10.8 Robotics and Autonomous Systems

ROS (Robot Operating System) [Quigley et al., 2009] provides typed message definitions and sensor abstractions. Gazebo [Koenig and Howard, 2020] simulates physical systems. Our work comple-

ments these by providing compile-time verification that perception, reasoning, and action are semantically aligned. A typical ROS node might operate on untyped sensor messages; our ABI/AST would provide static guarantees that sensor frames, units, and action ranges are respected throughout.

10.9 Our Distinguishing Contributions

1. **Semantic Types as First-Class**: unlike prior work, semantics (Measurement, Distribution, Enum) are not annotations but type-system objects that drive verification and lowering.
2. **RDF/IRI Grounding**: we explicitly tie tensor semantics to global RDF IRIs, enabling interoperability and ontology-driven reasoning.
3. **Three-Graph Model**: we formalize the interplay of computation, type, and semantic graphs, showing how all three must align for a well-formed system.
4. **Knowledge Realm Fusion**: we provide explicit type-safe bridges for crossing between perception (Measurement), knowledge (Enum/Distribution), and reasoning (Atomic), preventing silent semantic drift.
5. **Runtime Agnostic**: unlike ONNX-specific approaches, our ABI/AST lowers to any target while preserving semantics.

11 Type Checking and Guarantees

11.1 Compile-Time Checks

The type checker validates:

1. **Shape Unification**: symbols refined, ranges narrowed; no shape conflict reaches runtime.
2. **DType Compatibility**: no implicit numeric casts; primitives declare exact DType.
3. **Semantic Coherence**: Measurement units match; Distribution-aware ops are applied only to Distribution types; Enum vocabularies align.

Rejects ill-typed programs statically. Where checks cannot be discharged (e.g., dynamic sequence length), emits explicit runtime Assert.

11.2 Formal Properties

- **Progress**: *Well-typed graphs execute without shape or DType errors.* (Proof: by induction over type check, every primitive's output matches successor's input.)
- **Semantic Preservation**: *Tensor semantics flow through primitives.* Measurement frame is preserved; Distribution normalization is enforced; Enum bounds are maintained. No implicit unit conversion or categorical collapse.
- **Scope**: Types ensure internal compositional correctness, not physical ground truth. A sensor may malfunction, but the type system ensures the erroneous data is processed consistently and fails explicitly if constraints are violated.

12 Code Synthesis and Automation

A declarative ABI enables deterministic code generation by LLMs and automated tools. Sensor bindings, pipeline wiring, and safety wrappers (assertions, bounds checks) can be synthesized from high-level specifications. Example:

From spec to pipeline An LLM given a type schema (sensor outputs, model architecture, actuator constraints) can generate:

- Sensor binding code: `bind CameraRGB.output -> ConvStem.input` with type validation
- Graph interconnect: ensuring shape/semantic compatibility at each edge
- Safety wrappers: explicit assertions for dynamic ranges, with justification for which checks are elided

This reduces manual engineering effort and improves reproducibility: two teams using the same type schema will generate semantically equivalent pipelines.

13 Discussion

13.1 Scope and Ground Truth

Types guarantee *logical* consistency: every primitive respects its input types, and output types are verifiably correct. They do *not* guarantee *physical* truth. A camera sensor may be misaligned, low-light, or occluded; the resulting tensor will still have type `Measurement{radiance,frame:camera}` even if content is corrupted. Empirical validation (train/test splits, unit tests, sensor calibration) remains essential. The type system makes failures *explicit and auditable*: when a sensor or primitive produces surprising output, the type system preserves evidence (types, semantics, bounds) enabling root-cause diagnosis.

13.2 Extensibility

The ontology of Semantics (Atomic, Measurement, Distribution, Enum, Structured) is intentionally minimal but extensible. Rich unit systems (SI, custom), physical constraints, and domain-specific types (e.g., poses, graph structures) can be added without breaking the core type checker.

13.3 Performance

Assertion elision via symbolic analysis and range inference minimizes runtime overhead. For large models, the cost of metadata and type checks is amortized across thousands of operations. Benchmarking against untyped baselines (raw ONNX) is needed; preliminary evidence suggests overhead $< 5\%$ with aggressive elision.

13.4 Test/Train Reproducibility

The type system makes test and train graphs identical in structure and semantics. Determinism is enforced: same sensor input + same model weights $\Rightarrow$ same output (modulo floating-point rounding). This enables reproducible evaluation, ablation studies, and formal validation—critical for safety-critical AI.

14 Conclusion

We present a statically typed ABI and AST that grounds machine intelligence in reality by fusing heterogeneous knowledge realms. Sensors produce Measurement types (physical reality). Symbolic knowledge enters as Enum types. Probabilistic models output Distribution types. Type-preserving primitives enable safe composition and explicit projection between realms. The ABI lowers to any runtime—ONNX, custom kernels, specialized hardware—ensuring semantic invariants are preserved end-to-end.

Test and train graphs are identical in structure and semantics, enabling reproducible validation. Compile-time type checking rejects semantic unsafe programs; runtime assertions are explicit and elided when provably safe. By embedding DType, Shape, and Semantics into tensors, we achieve deterministic mapping of observations to computation to action, enabling machines to interpret the world with formal, verifiable reasoning grounded in measurable reality.

Future work includes richer semantic ontologies (physical unit systems, stochastic process types), certified compiler backends with formal proofs of semantic preservation, and integration with symbolic reasoning systems to bridge discrete and continuous knowledge.

References

M Abadi, A Agarwal, et al. Tensorflow: A system for large-scale machine learning, 2016. tensorflow.org.

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

Kurt Bollacker, Colin Evans, Praveen Paritosh, et al. Freebase: A collaboratively created graph database for structuring human knowledge. 2008.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, et al. Tvm: An automated end-to-end optimizing compiler for deep learning. *13th USENIX Symposium on Operating Systems Design and Implementation*, 2018.

Karsten Cohen and Max Welling. Verification of neural networks, 2020. NeurIPS 2020 Workshop.

Richard Cyganiak, David Wood, and Markus Lanthaler. Rdf 1.1 concepts and abstract syntax. *W3C Recommendation*, 2014.

Marta Garnelo and Murray Shanahan. Towards deep learning models resistant to large perturbations. 2019.

Jean-Yves Girard. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur. 1972.

W3C OWL Working Group. Owl 2 web ontology language. *W3C Recommendation*, 2012.

Armin Haller, Krzysztof Janowicz, Simon John Cox, et al. Semantic sensor network ontology. *W3C Recommendation*, 2017.

James Hegarty, John Brunhaver, Zachary DeVito, et al. Darkroom: Compiling high-level image processing code into hardware pipelines. 2013.

Paul Heil, Shahab Anjomshoae, and Davide Calvaresi. Towards formal semantics of machine learning. 2022.

Troels Henriksen, Niels GW Serup, Martin Elsman, Cosmin E Oancea, and Mikkel Thorup. Futhark: Practical functional high-performance computing. *arXiv preprint arXiv:1604.04335*, 2016.

Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 1969.

Ralph Hodgson et al. Qudt: Quantities, units, dimensions, and data types. 2023. https://qudt.org/.

Nathan Koenig and Andrew Howard. Gazebo: A dynamic engine for simulation and robotics. 2020.

Ramana Kumar, Magnus O Myreen, Michael Norrish, et al. Cakeml: A verified implementation of ml. *ACM SIGPLAN Notices*, 49(1):179–191, 2014.

Chris Lattner, Mehdi Amini, Dan Bonachea, et al. Mlir: A compiler infrastructure for the end of moore's law. 2021.

Jens Lehmann, Robert Isele, Max Jakob, et al. Dbpedia: A large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2):167–195, 2015.

Xavier Leroy. The compcert c compiler: Formally verified for real-world reliability. 2023. https://compcert.inria.fr/.

Jiayuan Mao, Chuang Gan, Pushmeet Kohli, et al. Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. 2019.

Alistair Miles and Sean Bechhofer. Skos: Simple knowledge organization system. *W3C Recommendation*, 2009.

Adam Paszke, Sam Gross, et al. Pytorch: An imperative style, high-performance deep learning library, 2019.

Adam Paszke, MJ Johnson, and David Duvenaud. Dex: A typed language for modular computational physics. 2022.

Morgan Quigley, Ken Conley, Brian Gerkey, et al. Ros: an open-source robot operating system. *ICRA workshop on open source software*, 3(3.2), 2009.

Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, et al. Halide: A language and compiler for optimizing parallelism, locality, and recomputation. *ACM SIGPLAN Notices*, 2013.

Nadav Rotem, Jordan Fix, Saleem Abdulrasool, et al. Glow: Graph lowering compiler techniques for neural networks. 2020.

Agda Development Team. Agda: A dependently typed programming language, 2023a. https://agda.readthedocs.io/.

Coq Development Team. The coq proof assistant, 2023b. https://coq.inria.fr/.

Yuchi Tian, Kexin Pei, Suman Seth, et al. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. 2018.

Sebastian Uchitel and Dor Zussman. A formal semantics for machine learning systems. 2021.

Denny Vrandečić and Markus Krötzsch. Wikidata: A free and open knowledge base that can be read and edited by both humans and machines. *Communications of the ACM*, 57(10):78–85, 2014.