Transformer Algebra

Compositional Semantics and Correct Compilation of Transformer Architectures

Adrian Diamond

AAD Systems

adrian@aadsystems.com

https://aadsystems.com

September 2026

Table of Contents

Abstract

We develop a compositional semantics and compilation framework for transformer architectures. Transformer computations are represented by a formal algebra of architectural operations whose compositions define a Transformer Intermediate Representation (TIR). The representation separates abstract architecture from backend implementation, permitting transformer programs to be normalized, analyzed, transformed, and compiled into executable target representations.

A denotational semantics assigns mathematical meaning to TIR programs, while an operational rewriting layer specifies normalization and certified transformation. Backend interpretations model the lowering of well-formed programs into executable target representations. The central correctness problem is semantic preservation under compilation: under stated assumptions, execution of a compiled target program must agree with the denotation of its source architecture. This framework provides a basis for compositional reasoning about transformer architectures, semantics-preserving optimization, architecture equivalence, and verified compiler construction.

Keywords: transformer architectures, compiler correctness, intermediate representations, compositional semantics, denotational semantics, operational semantics, formal verification, deep learning.

1 Introduction

Transformer architectures are ordinarily specified as compositions of tensor operations: embeddings, query–key–value projections, attention, feedforward maps, normalization, residual connections, and repeated layer composition. Although these operations admit precise mathematical descriptions, an architecture description alone does not provide a formal account of how the same transformer program may be represented, transformed, and compiled across different executable implementations.

This paper develops *Transformer Algebra*, a compositional formalism for representing transformer architectures independently of any particular execution backend. Programs in the algebra admit a Transformer Intermediate Representation (TIR), which serves as the boundary between abstract architecture and concrete implementation. TIR programs may be composed, normalized, analyzed, transformed, and interpreted by backend-specific compilers.

The central requirement is semantic preservation. If

$$\Theta \vdash P : A \to D$$

is a well-formed TIR program and $C_B(P)$ is its compilation into a backend B, exact compiler correctness is expressed by the commuting relation

$$[\![C_B(P)]\!]_{B,\lambda_B(\rho)} \circ e_A^B = e_D^B \circ [\![P]\!]_\rho.$$

Here e_A^B and e_D^B relate source semantic values to their backend representations, while λ_B lowers source parameter environments. Thus source and backend representations need not be literally identical for compilation to preserve architectural meaning.

This shifts the role of category theory in the framework. Categorical structure supplies the compositional semantics of transformer programs, while the compiler layer supplies their executable realization. The objective is therefore not merely to describe transformer architectures categorically, but to establish a mathematical basis for correct compilation between an abstract transformer language and concrete executable representations.

Motivation

A transformer compiler should separate architectural meaning from implementation detail. Embedding, attention, feedforward evolution, normalization, residual assembly, head composition, and layer stacking should have abstract meanings that do not depend on whether their executable realizations are expressed in a tensor library, an accelerator-oriented graph, or an interchange format.

Such a separation permits three questions to be stated precisely: when two architecture programs have the same meaning; when an architecture transformation preserves that meaning; and when compilation into a concrete backend preserves the semantics of the source program. These questions motivate the algebraic, operational, denotational, and compiler semantics developed below.

Contributions

The principal contributions of this paper are a typed cartesian intermediate representation for transformer architectures; parameter-indexed denotational semantics and certified architecture transformations; a representation-aware compositional compiler-correctness theorem; an end-to-end preservation theorem covering source and backend optimization; and a compositional numerical-error analysis for finite-precision realizations under explicit local assumptions.

2 The Transformer Algebra

Transformer Algebra separates transformer architecture from any particular tensor library, execution engine, or hardware backend. Its programs form a typed architecture language whose compiler representation is the Transformer Intermediate Representation (TIR).

The formal language must represent not only sequential composition but also dataflow branching. The same sequence representation may feed query, key, and value projections; a residual input may be consumed both directly and through a transformed branch; and one representation may feed multiple attention heads. TIR therefore includes explicit cartesian product structure.

2.1 TIR Types

Definition 2.1 (TIR Types)**.** The types of Transformer Algebra are generated by

$$A, B ::= \mathbf{1} \mid X_{\mathrm{tok},n} \mid X_{n,d} \mid H_{n,d_h} \mid A_n \mid A \times B.$$

Here:

- $\mathbf{1}$ is the unit type;
- $X_{\mathrm{tok},n}$ is the type of token sequences of length n;
- $X_{n,d}$ is the type of length-n sequences of d-dimensional model representations;
- H_{n,d_h} is the type of length-n representations associated with one attention head of dimension d_h;
- A_n is the type of $n \times n$ attention-score matrices; and
- $A \times B$ is a product type representing simultaneous availability of values of types A and B.

For $m \geq 1$, define the multi-head product type

$$H_{n,d_h}^{(m)} = \underbrace{H_{n,d_h} \times \cdots \times H_{n,d_h}}_{m \text{ factors}}.$$

Unless otherwise stated, finite product expressions are right-associated.

The symbol $\times$ denotes product structure in the architecture language. It should not be confused with the tensor product of vector spaces. Operationally, a product value is a typed tuple of simultaneously available intermediate values.

2.2 Parameter Signatures

Architecture structure and numerical parameter values are distinct components of Transformer Algebra.

Definition 2.2 (Parameter Signature)**.** A parameter signature Θ is a finite collection of symbolic parameter names

$$\theta$$

together with a required parameter type

$$\mathrm{Par}(\theta)$$

for each symbol.

For example, a parameter signature for an attention head may contain

$$\theta_Q, \qquad \theta_K, \qquad \theta_V, \qquad \theta_O,$$

representing query, key, value, and output-projection parameters.

A compiler representation stores such parameter symbols or handles as part of the architecture program; concrete numerical values are supplied separately.

2.3 Primitive Architectural Operations

For a fixed parameter signature Θ, the primitive operations of TIR are typed architectural generators.

Definition 2.3 (Embedding Operation)**.** A parameterized embedding operation has type

$$\mathrm{Embed}_{\theta_E} : X_{\mathrm{tok},n} \to X_{n,d}.$$

Definition 2.4 (Projection Operations)**.** For each attention head,

$$Q_{\theta_Q}, K_{\theta_K}, V_{\theta_V} : X_{n,d} \to H_{n,d_h}$$

denote the query, key, and value projections.

Definition 2.5 (Score Operation)**.** The score operation has type

$$\mathrm{Score} : H_{n,d_h} \times H_{n,d_h} \to A_n.$$

Definition 2.6 (Attention Normalization)**.** The attention-normalization operation has type

$$\mathrm{Norm} : A_n \to A_n.$$

Definition 2.7 (Attention Aggregation)**.** The attention-aggregation operation has type

$$\mathrm{Attend} : A_n \times H_{n,d_h} \to H_{n,d_h}.$$

Definition 2.8 (Head Assembly)**.** For m attention heads, a parameterized assembly operation has type

$$\mathrm{Merge}_{\theta_O} : H_{n,d_h}^{(m)} \to X_{n,d}.$$

Definition 2.9 (Feedforward Operation)**.** A parameterized feedforward operation has type

$$\mathrm{FF}_{\theta_F} : X_{n,d} \to X_{n,d}.$$

Definition 2.10 (Model Normalization)**.** A model-normalization operation has type

$$\mathrm{LN}_{\theta_N} : X_{n,d} \to X_{n,d},$$

where θ_N may contain learned scale and shift parameters when the represented architecture uses them.

Definition 2.11 (Residual Assembly)**.** Residual assembly has type

$$\mathrm{Res} : X_{n,d} \times X_{n,d} \to X_{n,d}.$$

Additional transformer primitives may be added to the signature provided that their source and target types, parameter requirements, and denotational semantics are specified.

Throughout the core development, the primitive vocabulary is fixed and is suppressed from the notation $\mathrm{Term}_\Theta(A, B)$ and $\mathcal{T}_{\mathrm{IR}}(\Theta)$. Any later extension of that vocabulary, such as the certified QKV fusion operation, is stated explicitly.

When parameter labels are irrelevant to a derivation, we suppress their subscripts and write simply

$$Q,\ K,\ V,\ \mathrm{Merge},\ \mathrm{FF},\ \mathrm{LN}.$$

2.4 Raw TIR Syntax

TIR is first defined as a typed syntax. Categorical equations are imposed only after raw program structure has been constructed.

Definition 2.12 (Raw TIR Terms)**.** For TIR types A and B, the set

$$\mathrm{Term}_\Theta(A, B)$$

of raw TIR terms is generated by the following constructors:

1. every primitive operation
$$g : A \to B$$
is a term;

2. every type A has an identity term
$$\mathrm{id}_A : A \to A;$$

3. every type A has a terminal morphism
$$!_A : A \to \mathbf{1};$$

4. if
$$P : A \to B \qquad \text{and} \qquad Q : B \to C,$$
then
$$Q \circ P : A \to C;$$

5. for every product type $A \times B$ there are projection terms
$$\pi_1 : A \times B \to A, \qquad \pi_2 : A \times B \to B;$$

6. if
$$P : C \to A \qquad \text{and} \qquad Q : C \to B,$$
then the pairing
$$\langle P, Q \rangle : C \to A \times B$$
is a term.

Pairing is the fundamental fan-out operation of TIR. It permits one program value to be consumed by several downstream operations without introducing an informal copying convention.

Definition 2.13 (Diagonal)**.** For every TIR type A, define
$$\Delta_A = \langle \mathrm{id}_A, \mathrm{id}_A \rangle : A \to A \times A.$$

Thus copying is expressible explicitly in the language.

More generally, if
$$P_i : C \to A_i, \qquad 1 \le i \le m,$$
then
$$\langle P_1, \ldots, P_m \rangle : C \to A_1 \times \cdots \times A_m$$
denotes iterated pairing.

2.5 Derived Parallel Composition

Independent computations on independent inputs are represented by a derived product operation.

If
$$P : A \to B \qquad \text{and} \qquad Q : C \to D,$$
define
$$P \times Q = \langle P \circ \pi_1, Q \circ \pi_2 \rangle : A \times C \to B \times D.$$

This distinguishes two operations that must not be conflated:
$$\langle P, Q \rangle : C \to A \times B$$
branches from one common input, whereas
$$P \times Q : A \times C \to B \times D$$
applies independent computations to independent components.

This distinction is required for query–key–value fan-out, residual connections, and multi-head computation.

2.6 Typing and Well-Formedness

Typing is written

$$\Theta \vdash P : A \to B.$$

Definition 2.14 (Well-Formed TIR Program)**.** A raw TIR term is well formed when every syntax constructor satisfies its typing rule and every primitive operation satisfies the sequence-length, dimension, head-count, parameter-type, and shape constraints associated with its signature.

For example,

$$\Theta \vdash \langle Q, K, V \rangle : X_{n,d} \to H_{n,d_h} \times H_{n,d_h} \times H_{n,d_h}$$

is well formed because the three projections have a common source.

Likewise, whenever

$$\Theta \vdash P : X_{n,d} \to X_{n,d},$$

the residual program

$$\mathrm{Res} \circ \left\langle \mathrm{id}_{X_{n,d}}, P \right\rangle : X_{n,d} \to X_{n,d}$$

is well formed.

2.7 Structural Equivalence

Raw TIR syntax records program representation. Distinct syntax trees are not identified merely because they satisfy the same structural laws.

Let

$$\equiv_{\mathrm{str}}$$

be the least typed congruence containing the following equations.

The category laws are

$$P \circ \mathrm{id} \equiv_{\mathrm{str}} P, \qquad \mathrm{id} \circ P \equiv_{\mathrm{str}} P,$$

and

$$(R \circ Q) \circ P \equiv_{\mathrm{str}} R \circ (Q \circ P).$$

Terminality requires that every well-typed

$$R : A \to \mathbf{1}$$

satisfy

$$R \equiv_{\mathrm{str}} !_A.$$

The product equations are

$$\pi_1 \circ \langle P, Q \rangle \equiv_{\mathrm{str}} P,$$

$$\pi_2 \circ \langle P, Q \rangle \equiv_{\mathrm{str}} Q,$$

and

$$\langle \pi_1 \circ R, \pi_2 \circ R \rangle \equiv_{\mathrm{str}} R$$

whenever the terms are well typed.

Definition 2.15 (Transformer IR Category)**.** For parameter signature Θ, define

$$\mathcal{T}_{\mathrm{IR}}(\Theta)$$

to have TIR types as objects and structural-equivalence classes

$$[P]_{\mathrm{str}}$$

of well-formed raw terms

$$P : A \to B$$

as morphisms.

Proposition 2.16 (Cartesian Structure of TIR)**.** *For every parameter signature* Θ*,*

$$\mathcal{T}_{\mathrm{IR}}(\Theta)$$

is a cartesian category generated by the transformer primitive signature.

Proof. Composition and identities are inherited from raw TIR syntax, and the category equations hold in the quotient by $\equiv_{\mathrm{str}}$.

The object $\mathbf{1}$ is terminal because every morphism into it is identified with the corresponding terminal morphism $!_A$.

For $A \times B$, the projection morphisms are induced by π_1 and π_2. Given morphisms represented by

$$P : C \to A \qquad \text{and} \qquad Q : C \to B,$$

their pairing is represented by

$$\langle P, Q \rangle : C \to A \times B.$$

The projection and uniqueness equations follow from the defining product equations of $\equiv_{\mathrm{str}}$. Hence $A \times B$ satisfies the categorical product universal property. □

The distinction between raw syntax and

$$\mathcal{T}_{\mathrm{IR}}(\Theta)$$

is important for compilation. A compiler may retain a concrete syntax tree, SSA-style representation, or directed acyclic graph, while mathematical reasoning may operate on the corresponding structural-equivalence class.

In particular, pairing describes shared use of a value; it does not require a runtime to physically duplicate that value.

2.8 Parameter Environments

Definition 2.17 (Parameter Environment)**.** For a parameter signature Θ, define

$$\mathrm{Env}(\Theta) = \prod_{\theta \in \Theta} \mathrm{Par}(\theta).$$

An element

$$\rho \in \mathrm{Env}(\Theta)$$

assigns a concrete parameter value of the required type to every symbolic parameter in Θ.

For example,

$$\rho(\theta_Q) = W_Q, \qquad \rho(\theta_K) = W_K, \qquad \rho(\theta_V) = W_V, \qquad \rho(\theta_O) = W_O.$$

The same TIR architecture may therefore be interpreted under different parameter environments without changing its syntactic architecture.

2.9 Architectural Factorization

The cartesian structure makes the standard attention factorization fully typed.

First,

$$\langle Q, K\rangle : X_{n,d} \to H_{n,d_h} \times H_{n,d_h}.$$

Therefore

$$\mathrm{Score} \circ \langle Q, K\rangle : X_{n,d} \to A_n,$$

and

$$\mathrm{Norm} \circ \mathrm{Score} \circ \langle Q, K\rangle : X_{n,d} \to A_n.$$

Pairing the normalized score branch with the value projection gives

$$\langle \mathrm{Norm} \circ \mathrm{Score} \circ \langle Q, K\rangle, V\rangle : X_{n,d} \to A_n \times H_{n,d_h}.$$

Finally,

$$\boxed{\mathrm{Head} = \mathrm{Attend} \circ \langle \mathrm{Norm} \circ \mathrm{Score} \circ \langle Q, K\rangle, V\rangle : X_{n,d} \to H_{n,d_h}.}$$

Thus standard single-head attention is expressible as a well-typed TIR program without implicit duplication or informal branching.

3 Transformer Blocks as Compositional Programs

The typed constructors of TIR permit complete transformer blocks to be written as genuine programs rather than as schematic tensor expressions. In particular, pairing represents fan-out from a shared input, while sequential composition represents data dependence between successive computations.

Throughout this section, fix a parameter signature Θ.

3.1 Single-Head Attention Programs

For one attention head, let

$$Q, K, V : X_{n,d} \to H_{n,d_h}$$

denote parameterized projection primitives, with parameter labels suppressed when no ambiguity arises.

The query and key branches are paired as

$$\langle Q, K\rangle : X_{n,d} \to H_{n,d_h} \times H_{n,d_h}.$$

Hence the normalized score branch is

$$S = \mathrm{Norm} \circ \mathrm{Score} \circ \langle Q, K\rangle : X_{n,d} \to A_n.$$

Pairing S with the value branch gives

$$\langle S, V\rangle : X_{n,d} \to A_n \times H_{n,d_h}.$$

Definition 3.1 (Single-Head Attention Program)**.** The TIR program for one attention head is

$$\boxed{\mathrm{Head} = \mathrm{Attend} \circ \langle \mathrm{Norm} \circ \mathrm{Score} \circ \langle Q, K\rangle, V\rangle : X_{n,d} \to H_{n,d_h}.}$$

This term is well typed entirely within the TIR language. No implicit duplication of the source representation is required.

3.2 Multi-Head Composition

Let

$$\mathrm{Head}_i : X_{n,d} \to H_{n,d_h}, \qquad 1 \le i \le m,$$

be well-formed attention-head programs.

Their common-input fan-out is represented by iterated pairing:

$$\langle \mathrm{Head}_1, \ldots, \mathrm{Head}_m \rangle : X_{n,d} \to H_{n,d_h}^{(m)}.$$

Definition 3.2 (Multi-Head Attention Program)**.** The corresponding multi-head attention program is

$$\boxed{\mathrm{MHA} = \mathrm{Merge} \circ \langle \mathrm{Head}_1, \ldots, \mathrm{Head}_m \rangle : X_{n,d} \to X_{n,d}.}$$

Proposition 3.3 (Multi-Head Typing)**.** *If every*

$$\mathrm{Head}_i : X_{n,d} \to H_{n,d_h}$$

is well formed and the resulting head-product type satisfies the input requirements of Merge*, then* MHA *is a well-formed TIR program of type*

$$X_{n,d} \to X_{n,d}.$$

Proof. Iterated pairing produces

$$\langle \mathrm{Head}_1, \ldots, \mathrm{Head}_m \rangle : X_{n,d} \to H_{n,d_h}^{(m)}.$$

The codomain of this term is the domain of Merge. Type-correct sequential composition therefore gives

$$\mathrm{MHA} : X_{n,d} \to X_{n,d}.$$

□

3.3 Transformer Blocks

A transformer block combines attention, residual assembly, normalization, and feedforward computation. TIR records the ordering explicitly, allowing different transformer conventions to be represented without changing the underlying language.

For a post-normalization block, first define the attention residual

$$R_{\mathrm{att}} = \mathrm{Res} \circ \left\langle \mathrm{id}_{X_{n,d}}, \mathrm{MHA} \right\rangle : X_{n,d} \to X_{n,d}.$$

Let

$$\mathrm{LN}_1 : X_{n,d} \to X_{n,d}$$

denote the first normalization operation and define

$$S_1 = \mathrm{LN}_1 \circ R_{\mathrm{att}}.$$

The feedforward residual map is

$$R_{\mathrm{ff}} = \mathrm{Res} \circ \left\langle \mathrm{id}_{X_{n,d}}, \mathrm{FF} \right\rangle : X_{n,d} \to X_{n,d}.$$

Let

$$\mathrm{LN}_2 : X_{n,d} \to X_{n,d}$$

denote the second normalization operation.

Definition 3.4 (Post-Normalization Transformer Block)**.** A post-normalization transformer block is the TIR program

$$\boxed{B = \mathrm{LN}_2 \circ R_{\mathrm{ff}} \circ \mathrm{LN}_1 \circ R_{\mathrm{att}} : X_{n,d} \to X_{n,d}.}$$

Equivalently,

$$B = \mathrm{LN}_2 \circ \mathrm{Res} \circ \langle \mathrm{id}, \mathrm{FF} \rangle \circ \mathrm{LN}_1 \circ \mathrm{Res} \circ \langle \mathrm{id}, \mathrm{MHA} \rangle ,$$

with identity subscripts suppressed.

Other block organizations, including pre-normalization variants, are represented by changing the typed composition order rather than by changing the TIR language.

3.4 Depth Composition

A transformer architecture of depth L is a finite sequential composition

$$P_L = B_L \circ B_{L-1} \circ \cdots \circ B_1,$$

where every block has type

$$B_i : X_{n,d} \to X_{n,d}.$$

Proposition 3.5 (Depth Typing)**.** *For every finite $L \geq 1$, if*

$$B_i : X_{n,d} \to X_{n,d} \qquad (1 \leq i \leq L)$$

are well-formed TIR programs, then

$$P_L : X_{n,d} \to X_{n,d}$$

is well formed.

Proof. The result follows by induction on L using the typing rule for sequential composition. □

3.5 Parameter-Indexed Denotational Semantics

TIR syntax describes architecture independently of one particular parameter assignment. Its mathematical meaning is therefore indexed by a compatible parameter environment.

Fix

$$\rho \in \mathrm{Env}(\Theta).$$

Definition 3.6 (Denotational Interpretation)**.** A denotational interpretation assigns to every TIR type A a semantic domain

$$[\![A]\!]$$

and to every well-formed raw TIR term

$$\Theta \vdash P : A \to B$$

a mathematical map

$$[\![P]\!]_\rho : [\![A]\!] \to [\![B]\!].$$

The unit and product types are interpreted by

$$[\![\mathbf{1}]\!] = \{*\},$$

and

$$[\![A \times B]\!] = [\![A]\!] \times [\![B]\!].$$

Parameterized primitive operations are interpreted using the values supplied by ρ. Nonparameterized primitives have fixed mathematical interpretations.

The structural constructors are interpreted by

$$[\![\mathrm{id}_A]\!]_\rho = \mathrm{id}_{[\![A]\!]},$$

$$[\![!_A]\!]_\rho(x) = *,$$

$$[\![Q \circ P]\!]_\rho = [\![Q]\!]_\rho \circ [\![P]\!]_\rho,$$

$$[\![\pi_1]\!]_\rho(a, b) = a, \qquad [\![\pi_2]\!]_\rho(a, b) = b,$$

and

$$[\![\langle P, Q\rangle]\!]_\rho(x) = ([\![P]\!]_\rho(x), [\![Q]\!]_\rho(x)).$$

Consequently, for the derived product of programs,

$$P \times Q : A \times C \to B \times D,$$

we obtain

$$[\![P \times Q]\!]_\rho(a, c) = ([\![P]\!]_\rho(a), [\![Q]\!]_\rho(c)).$$

Proposition 3.7 (Structural Soundness)**.** *If two well-formed raw TIR terms satisfy*

$$P \equiv_{\mathrm{str}} Q,$$

then for every compatible parameter environment

$$\rho \in \mathrm{Env}(\Theta),$$

$$[\![P]\!]_\rho = [\![Q]\!]_\rho.$$

Proof. It is sufficient to check the generating equations of $\equiv_{\mathrm{str}}$.

The identity and associativity equations hold because function composition satisfies the category laws. Every map into $[\![\mathbf{1}]\!] = \{*\}$ is the unique constant map to $*$, giving terminality.

For pairing,

$$[\![\pi_1 \circ \langle P, Q\rangle]\!]_\rho(x) = [\![P]\!]_\rho(x),$$

and similarly for π_2. Finally,

$$([\![\pi_1 \circ R]\!]_\rho(x), [\![\pi_2 \circ R]\!]_\rho(x)) = [\![R]\!]_\rho(x).$$

Closure under typed congruence then gives the result. □

Corollary 3.8 (Well-Defined Semantics on TIR)**.** *For every parameter environment ρ, denotation descends from raw syntax to a well-defined interpretation of*

$$\mathcal{T}_{\mathrm{IR}}(\Theta).$$

Thus categorical reasoning on structural-equivalence classes and execution of concrete TIR syntax share the same denotational semantics.

3.6 Compositionality

Proposition 3.9 (Compositionality of Denotation)**.** *The denotation of every well-formed TIR term is determined compositionally by the denotations of its primitives and its syntax constructors.*

Proof. The claim follows directly from the inductive definition of $[\![-]\!]_\rho$ on raw TIR terms. □

In particular,

$$[\![\mathrm{Head}]\!]_\rho = [\![\mathrm{Attend}]\!]_\rho \circ \langle [\![\mathrm{Norm} \circ \mathrm{Score} \circ \langle Q, K \rangle]\!]_\rho, [\![V]\!]_\rho \rangle \,.$$

The concrete tensor interpretation of these maps is given in Section 4.

3.7 Semantic Equivalence

Architecture-level equivalence must not depend on one accidental numerical parameter assignment.

Definition 3.10 (Semantic Equivalence)**.** Let

$$\Theta \vdash P, Q : A \to B.$$

The programs P and Q are semantically equivalent, written

$$P \simeq Q,$$

when

$$\forall \rho \in \mathrm{Env}(\Theta), \qquad [\![P]\!]_\rho = [\![Q]\!]_\rho.$$

A parameter-specific equality may also be written

$$P \simeq_\rho Q$$

when

$$[\![P]\!]_\rho = [\![Q]\!]_\rho$$

for one fixed environment ρ.

The distinction is important: architecture transformations claimed correct in TIR will normally be required to preserve denotation for every compatible parameter environment.

Proposition 3.11 (Semantic Equivalence is a Typed Congruence)**.** *If*

$$P \simeq Q,$$

then replacing P by Q inside any well-typed TIR context preserves semantic equivalence.

Proof. For every compatible environment ρ,

$$[\![P]\!]_\rho = [\![Q]\!]_\rho.$$

Sequential composition preserves equality of functions, and pairing preserves equality componentwise. The projection and terminal constructors are fixed. Structural induction on the surrounding typed TIR context therefore preserves the equality for every ρ. □

3.8 Dataflow Interpretation

The cartesian syntax specifies sharing semantically without prescribing a physical copying strategy.

For example,

$$\langle Q, K, V \rangle : X_{n,d} \to H^{(3)}_{n,d_h}$$

may be represented by a compiler as one source value with three consumers:

$$x \longmapsto \big(Q(x), K(x), V(x)\big).$$

An SSA-style or directed-acyclic-graph implementation may therefore store one definition of x and references from the three projection nodes. The architecture semantics determines data dependence; the backend determines the concrete storage and execution strategy.

4 Concrete Tensor Semantics

The preceding sections define the syntax and compositional semantics of TIR. We now instantiate those definitions with the standard real-valued tensor semantics of transformer computation.

Throughout this section, fix a parameter signature Θ and a compatible environment

$$\rho \in \mathrm{Env}(\Theta).$$

The semantics defined here is an ideal real-arithmetic reference semantics. Finite-precision backend execution is treated separately in Section 8.

4.1 Semantic Domains

Let $\mathcal{V}$ denote the token vocabulary. We interpret the principal TIR types by

$$[\![X_{\mathrm{tok},n}]\!] = \mathcal{V}^n,$$

$$[\![X_{n,d}]\!] = \mathbb{R}^{n \times d},$$

$$[\![H_{n,d_h}]\!] = \mathbb{R}^{n \times d_h},$$

and

$$[\![A_n]\!] = \mathbb{R}^{n \times n}.$$

Product types are interpreted cartesianly:

$$[\![A \times B]\!] = [\![A]\!] \times [\![B]\!].$$

Thus the syntactic product structure of TIR corresponds directly to tuples of intermediate tensor values.

4.2 Embedding Semantics

Suppose

$$\rho(\theta_E) = (E, P),$$

where

$$E \in \mathbb{R}^{|\mathcal{V}| \times d}$$

is an embedding table and

$$P \in \mathbb{R}^{n \times d}$$

is a positional representation.

For a token sequence

$$t = (t_1, \ldots, t_n) \in \mathcal{V}^n,$$

define

$$[\![\mathrm{Embed}_{\theta_E}]\!]_\rho(t) = \begin{bmatrix} E_{t_1} \\ \vdots \\ E_{t_n} \end{bmatrix} + P.$$

Architectures that represent positional information separately may take $P = 0$ here and introduce an additional typed positional primitive.

4.3 Projection Semantics

For one attention head, suppose

$$\rho(\theta_Q) = W_Q, \qquad \rho(\theta_K) = W_K, \qquad \rho(\theta_V) = W_V,$$

with

$$W_Q, W_K, W_V \in \mathbb{R}^{d \times d_h}.$$

For

$$X \in \mathbb{R}^{n \times d},$$

define

$$[\![Q_{\theta_Q}]\!]_\rho(X) = XW_Q,$$
$$[\![K_{\theta_K}]\!]_\rho(X) = XW_K,$$

and

$$[\![V_{\theta_V}]\!]_\rho(X) = XW_V.$$

When parameter subscripts are suppressed, these are written simply as

$$[\![Q]\!]_\rho, \qquad [\![K]\!]_\rho, \qquad [\![V]\!]_\rho.$$

Affine projection variants may be represented by extending the corresponding parameter type to include bias vectors. The compositional results below are unchanged.

4.4 Score Semantics

Definition 4.1 (Scaled Score Semantics)**.** For

$$U, K' \in \mathbb{R}^{n \times d_h},$$

define

$$[\![\mathrm{Score}]\!]_\rho(U, K') = \frac{U(K')^\mathsf{T}}{\sqrt{d_h}}.$$

The score operation is nonparameterized, so its interpretation does not depend numerically on ρ; the subscript is retained for notational uniformity.

Its output lies in

$$\mathbb{R}^{n\times n}.$$

The factor

$$\frac{1}{\sqrt{d_h}}$$

is part of the standard scaled dot-product attention specification [1].

4.5 Attention Normalization

Definition 4.2 (Row-Wise Softmax)**.** For

$$Z \in \mathbb{R}^{n\times n},$$

define

$$[\![\mathrm{Norm}]\!]_\rho(Z)_{ij} = \frac{\exp(Z_{ij})}{\sum_{k=1}^{n}\exp(Z_{ik})}.$$

Thus each row of the score matrix is normalized independently.

4.6 Value Aggregation

Definition 4.3 (Attention Aggregation Semantics)**.** For

$$A \in \mathbb{R}^{n\times n}$$

and

$$V' \in \mathbb{R}^{n\times d_h},$$

define

$$[\![\mathrm{Attend}]\!]_\rho(A, V') = AV'.$$

4.7 Single-Head Attention Semantics

Recall the TIR program

$$\mathrm{Head} = \mathrm{Attend} \circ \langle \mathrm{Norm} \circ \mathrm{Score} \circ \langle Q, K\rangle, V\rangle\,.$$

Proposition 4.4 (Single-Head Semantic Reconstruction)**.** *For every compatible environment ρ satisfying*

$$\rho(\theta_Q) = W_Q, \qquad \rho(\theta_K) = W_K, \qquad \rho(\theta_V) = W_V,$$

and every

$$X \in \mathbb{R}^{n\times d},$$

the TIR denotation is

$$\boxed{[\![\mathrm{Head}]\!]_\rho(X) = \mathrm{softmax}_{\mathrm{row}}\left(\frac{(XW_Q)(XW_K)^{\mathsf{T}}}{\sqrt{d_h}}\right)(XW_V).}$$

Proof. By the semantics of pairing,

$$[\![\langle Q, K\rangle]\!]_\rho(X) = (XW_Q, XW_K).$$

Applying Score yields

$$\frac{(XW_Q)(XW_K)^\mathsf{T}}{\sqrt{d_h}}.$$

Applying Norm gives the row-wise softmax of this matrix. Pairing that result with

$$[\![V]\!]_\rho(X) = XW_V$$

and applying Attend gives the stated expression. □

This proposition verifies that the typed TIR construction recovers standard scaled dot-product attention.

4.8 Multi-Head Semantics

For

$$1 \le i \le m,$$

let

$$\mathrm{Head}_i : X_{n,d} \to H_{n,d_h}$$

have its own projection parameters supplied by ρ.

The denotation of their common-input pairing is

$$[\![\langle \mathrm{Head}_1, \ldots, \mathrm{Head}_m\rangle]\!]_\rho(X) = ([\![\mathrm{Head}_1]\!]_\rho(X), \ldots, [\![\mathrm{Head}_m]\!]_\rho(X)).$$

Suppose

$$\rho(\theta_O) = W_O, \qquad W_O \in \mathbb{R}^{(md_h)\times d}.$$

Define

$$[\![\mathrm{Merge}_{\theta_O}]\!]_\rho(H_1, \ldots, H_m) = \mathrm{Concat}(H_1, \ldots, H_m)W_O.$$

Proposition 4.5 (Multi-Head Semantic Reconstruction)**.** *For the TIR program*

$$\mathrm{MHA} = \mathrm{Merge} \circ \langle \mathrm{Head}_1, \ldots, \mathrm{Head}_m\rangle,$$

we have

$$\boxed{[\![\mathrm{MHA}]\!]_\rho(X) = \mathrm{Concat}\,([\![\mathrm{Head}_1]\!]_\rho(X), \ldots, [\![\mathrm{Head}_m]\!]_\rho(X))\,W_O.}$$

Proof. The denotation of iterated pairing forms the tuple of head outputs. The denotation of Merge concatenates those outputs and applies the output projection W_O. Composition gives the stated expression. □

4.9 Residual Semantics

For

$$X, Y \in \mathbb{R}^{n \times d},$$

define

$$[\![\mathrm{Res}]\!]_\rho(X, Y) = X + Y.$$

Consequently, if

$$P : X_{n,d} \to X_{n,d},$$

then

$$[\![\mathrm{Res} \circ \langle \mathrm{id}, P \rangle]\!]_\rho(X) = X + [\![P]\!]_\rho(X).$$

Thus the TIR pairing structure reproduces the ordinary residual connection without requiring implicit duplication of X.

4.10 Feedforward Semantics

Suppose

$$\rho(\theta_F) = (W_1, b_1, W_2, b_2),$$

where

$$W_1 \in \mathbb{R}^{d \times d_{\mathrm{ff}}}, \qquad b_1 \in \mathbb{R}^{d_{\mathrm{ff}}},$$
$$W_2 \in \mathbb{R}^{d_{\mathrm{ff}} \times d}, \qquad b_2 \in \mathbb{R}^{d}.$$

Let

$$\phi : \mathbb{R} \to \mathbb{R}$$

be the element-wise activation associated with the represented architecture.

For

$$X \in \mathbb{R}^{n \times d},$$

define

$$[\![\mathrm{FF}_{\theta_F}]\!]_\rho(X) = \phi\left(XW_1 + \mathbf{1}_n b_1^{\mathsf{T}}\right) W_2 + \mathbf{1}_n b_2^{\mathsf{T}},$$

where $\mathbf{1}_n$ denotes the length-n all-ones column vector and ϕ is applied elementwise.

This formulation permits ReLU, GELU, or another explicitly specified activation to be associated with the feedforward primitive.

4.11 Layer-Normalization Semantics

Suppose

$$\rho(\theta_N) = (\gamma, \beta, \varepsilon),$$

where

$$\gamma, \beta \in \mathbb{R}^d, \qquad \varepsilon > 0.$$

For each row i of

$$X \in \mathbb{R}^{n \times d},$$

define

$$\mu_i = \frac{1}{d} \sum_{j=1}^{d} X_{ij},$$

and

$$\sigma_i^2 = \frac{1}{d}\sum_{j=1}^{d}(X_{ij} - \mu_i)^2.$$

Then

$$[\![\mathrm{LN}_{\theta_N}]\!]_\rho(X)_{ij} = \gamma_j \frac{X_{ij} - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}} + \beta_j.$$

Different normalization primitives may be added to TIR when an architecture uses a different normalization convention.

4.12 Masked Attention

An attention mask should be specified without relying on an informal "sufficiently negative" real number.

Let

$$M \in \{0,1\}^{n\times n}$$

be an admissibility mask such that every row contains at least one allowed position:

$$\forall i, \qquad \sum_{j=1}^{n} M_{ij} \geq 1.$$

Define the masked row-wise normalizer by

$$\mathrm{msoftmax}(Z, M)_{ij} = \frac{M_{ij}\exp(Z_{ij})}{\sum_{k=1}^{n} M_{ik}\exp(Z_{ik})}.$$

If

$$\rho(\theta_M) = M,$$

an additional TIR primitive

$$\mathrm{NormMask}_{\theta_M} : A_n \to A_n$$

may therefore be interpreted by

$$[\![\mathrm{NormMask}_{\theta_M}]\!]_\rho(Z) = \mathrm{msoftmax}(Z, M).$$

The corresponding masked attention head is

$$\mathrm{Head}_M = \mathrm{Attend} \circ \langle \mathrm{NormMask}_{\theta_M} \circ \mathrm{Score} \circ \langle Q, K\rangle, V\rangle .$$

Its denotation is

$$[\![\mathrm{Head}_M]\!]_\rho(X) = \mathrm{msoftmax}\left(\frac{(XW_Q)(XW_K)^{\mathsf{T}}}{\sqrt{d_h}}, M\right)(XW_V).$$

This representation covers causal and padding masks while remaining entirely within real-valued semantics.

4.13 Transformer Block Semantics

Because all block primitives now have specified denotations, the transformer block defined in Section 3 has a concrete tensor interpretation.

For the post-normalization block

$$B = \mathrm{LN}_2 \circ R_{\mathrm{ff}} \circ \mathrm{LN}_1 \circ R_{\mathrm{att}},$$

we have

$$[\![R_{\mathrm{att}}]\!]_\rho(X) = X + [\![\mathrm{MHA}]\!]_\rho(X),$$

and

$$[\![R_{\mathrm{ff}}]\!]_\rho(Y) = Y + [\![\mathrm{FF}]\!]_\rho(Y).$$

Therefore

$$[\![B]\!]_\rho = [\![\mathrm{LN}_2]\!]_\rho \circ [\![R_{\mathrm{ff}}]\!]_\rho \circ [\![\mathrm{LN}_1]\!]_\rho \circ [\![R_{\mathrm{att}}]\!]_\rho.$$

Thus complete finite-depth transformer architectures constructed from these blocks inherit a concrete tensor semantics by composition.

4.14 Semantic Independence from Backend Representation

The equations in this section specify what TIR primitives mean. They do not specify how a backend must execute them.

A backend may combine projection matrices, fuse attention kernels, alter memory layouts, schedule independent branches differently, or represent the same architecture with a graph structurally different from the source TIR.

Such implementation choices are correct only when they preserve the semantic specification

$$[\![P]\!]_\rho$$

for the relevant source program and parameter environment.

This separation between source denotation and target execution is the basis for the compiler-correctness results developed below.

5 Operational Semantics and Certified Normalization

The denotational semantics of the preceding sections assigns mathematical meaning to TIR programs. We now specify how concrete TIR syntax may be canonicalized and transformed before backend lowering.

The operational layer acts on well-formed raw TIR terms rather than directly on morphisms of

$$\mathcal{T}_{\mathrm{IR}}(\Theta).$$

This distinction is essential. Structural equivalence identifies terms for mathematical reasoning, while a compiler manipulates concrete syntax trees, SSA values, or graph nodes.

Throughout this section, fix a parameter signature Θ.

5.1 Rewriting on Raw TIR

The syntax subject to transformation is exactly the raw TIR syntax defined in Section 2. Thus rewrite rules may occur inside terms generated from primitives, identities, terminal maps, composition, projections, and pairing.

Definition 5.1 (Typed TIR Rewrite)**.** A typed TIR rewrite is a directed transformation

$$P \longrightarrow Q$$

between well-formed raw TIR terms satisfying

$$\Theta \vdash P : A \to B \qquad \text{and} \qquad \Theta \vdash Q : A \to B.$$

We write

$$P \longrightarrow^* Q$$

for the reflexive transitive closure of the rewrite relation.

Because the source and target of a rewrite are required to have the same TIR type, every admissible rewrite preserves typing by construction.

Definition 5.2 (Uniformly Semantics-Preserving Rewrite)**.** A typed rewrite

$$P \longrightarrow Q$$

is uniformly semantics preserving when

$$\forall \rho \in \mathrm{Env}(\Theta), \qquad [\![P]\!]_\rho = [\![Q]\!]_\rho.$$

Equivalently,

$$P \simeq Q.$$

When compilation is specialized to one fixed parameter environment ρ, a rewrite may instead be certified relative to that environment by proving

$$[\![P]\!]_\rho = [\![Q]\!]_\rho.$$

Unless stated otherwise, the transformations considered below are uniform in ρ.

Definition 5.3 (Certified Rewrite)**.** A certified TIR rewrite is a typed rewrite together with a proof obligation establishing its semantic preservation under the stated preconditions.

Since semantic equivalence is a typed congruence, a certified local rewrite remains correct when performed inside any well-typed TIR context.

Proposition 5.4 (Contextual Rewrite Soundness)**.** *Suppose*

$$P \longrightarrow Q$$

is uniformly semantics preserving and $C[-]$ is any well-typed TIR context admitting both terms. Then

$$C[P] \simeq C[Q].$$

Proof. Uniform semantic preservation gives

$$[\![P]\!]_\rho = [\![Q]\!]_\rho$$

for every compatible ρ. The congruence property established in Section 3 then gives

$$[\![C[P]]\!]_\rho = [\![C[Q]]\!]_\rho$$

for every such environment. □

5.2 Structural Canonicalization

Structural equivalence

$$\equiv_{\mathrm{str}}$$

is symmetric, whereas compiler normalization requires a deterministic choice of representation. We therefore distinguish structural equality from a particular canonicalization algorithm.

Definition 5.5 (Structural Normalization)**.** Define

$$N_{\mathrm{str}}$$

recursively on well-formed raw TIR terms as follows.

1. Primitive operations, projections, terminal maps, and standalone identities are retained.
2. The components of every pairing
$$\langle P, Q \rangle$$
are recursively normalized.
3. Inside every maximal sequential-composition spine, all component terms are recursively normalized, identity factors are removed, and the remaining factors are rebuilt in a fixed right-associated order.
4. If removal of identity factors leaves no nonidentity factor in a sequential spine, the appropriately typed identity term is retained.

For example,

$$(R \circ \mathrm{id}) \circ (Q \circ P)$$

is structurally normalized to

$$R \circ (Q \circ P),$$

assuming the displayed composition is well typed.

The normalization procedure preserves the order of computational operations; it changes only structural presentation.

Theorem 5.6 (Structural Normalization Correctness)**.** *For every well-formed raw TIR term*

$$\Theta \vdash P : A \to B,$$

the structural normalization $N_{\mathrm{str}}(P)$ *satisfies:*

1.
$$\Theta \vdash N_{\mathrm{str}}(P) : A \to B;$$

2.
$$P \equiv_{\mathrm{str}} N_{\mathrm{str}}(P);$$

3. for every
$$\rho \in \mathrm{Env}(\Theta),$$
$$[\![N_{\mathrm{str}}(P)]\!]_\rho = [\![P]\!]_\rho;$$

4. and

$$N_{\mathrm{str}}\big(N_{\mathrm{str}}(P)\big) = N_{\mathrm{str}}(P).$$

Proof. Proceed by structural induction on P.

Primitive operations, projections, terminal maps, and identities preserve their typing and are unchanged except when an identity occurs as a removable factor of a composition.

For a pairing

$$\langle P, Q\rangle,$$

the induction hypothesis gives well-typed normalized components having the same source types and respective target types. Pairing therefore remains well typed. Structural equivalence and semantic equality follow componentwise.

For sequential composition, recursive normalization preserves the types of all factors. Removing an identity factor preserves the source and target of the composite, and reassociation preserves well-typed composition. The identity and associativity equations belong to $\equiv_{\mathrm{str}}$, so the normalized term is structurally equivalent to the original.

Structural soundness from Section 3 therefore gives

$$[\![N_{\mathrm{str}}(P)]\!]_\rho = [\![P]\!]_\rho$$

for every ρ.

Finally, after one application, every sequential spine contains no removable identity factor and already has the prescribed association; all subterms have also been recursively normalized. A second application therefore makes no change. □

This theorem gives the compiler a deterministic structural pass without requiring structural equality itself to be oriented as an arbitrary rewrite system.

5.3 Architecture-Level Rewrites

Structural normalization changes only representation. Architecture-level rewrites may replace one computational construction with another, for example by introducing a fused operation.

Such rewrites require semantic proofs rather than merely categorical structural laws.

A particularly important example is fusion of query, key, and value projections.

5.4 QKV Fusion

Let

$$W_Q, W_K, W_V \in \mathbb{R}^{d\times d_h}.$$

Define their horizontal concatenation by

$$W_{\mathrm{QKV}} = [W_Q \;\; W_K \;\; W_V] \in \mathbb{R}^{d\times 3d_h}.$$

Let

$$\mathrm{Split}_3 : \mathbb{R}^{n\times 3d_h} \to \mathbb{R}^{n\times d_h} \times \mathbb{R}^{n\times d_h} \times \mathbb{R}^{n\times d_h}$$

split the final matrix dimension into three consecutive blocks of width d_h.

Extend the available TIR primitive signature with the derived fused operation

$$\mathrm{QKVFuse}_{\theta_Q,\theta_K,\theta_V} : X_{n,d} \to H^{(3)}_{n,d_h},$$

whose denotation is

$$[\![\text{QKVFuse}_{\theta_Q,\theta_K,\theta_V}]\!]_\rho(X) = \text{Split}_3\left(X\,[\rho(\theta_Q)\ \ \rho(\theta_K)\ \ \rho(\theta_V)]\right).$$

The fused operation refers to the same three parameter symbols as the unfused projections. It therefore introduces no new source-level parameter environment.

Theorem 5.7 (QKV Fusion Correctness)**.** *Let*

$$Q_{\theta_Q}, K_{\theta_K}, V_{\theta_V} : X_{n,d} \to H_{n,d_h}$$

be the three projection primitives. Then

$$\boxed{\text{QKVFuse}_{\theta_Q,\theta_K,\theta_V} \simeq \langle Q_{\theta_Q}, K_{\theta_K}, V_{\theta_V}\rangle.}$$

Equivalently, for every compatible parameter environment ρ and every

$$X \in \mathbb{R}^{n\times d},$$

$$[\![\text{QKVFuse}_{\theta_Q,\theta_K,\theta_V}]\!]_\rho(X) = [\![\langle Q_{\theta_Q}, K_{\theta_K}, V_{\theta_V}\rangle]\!]_\rho(X).$$

Proof. Fix ρ and write

$$W_Q = \rho(\theta_Q), \qquad W_K = \rho(\theta_K), \qquad W_V = \rho(\theta_V).$$

By block-matrix multiplication,

$$X\,[W_Q\ \ W_K\ \ W_V] = [XW_Q\ \ XW_K\ \ XW_V].$$

Applying Split_3 therefore gives

$$\text{Split}_3\left(X\,[W_Q\ \ W_K\ \ W_V]\right) = (XW_Q, XW_K, XW_V).$$

By the projection semantics of Section 4,

$$XW_Q = [\![Q_{\theta_Q}]\!]_\rho(X),$$

and analogously for K_{θ_K} and V_{θ_V}.

By the denotation of TIR pairing,

$$(XW_Q, XW_K, XW_V) = [\![\langle Q_{\theta_Q}, K_{\theta_K}, V_{\theta_V}\rangle]\!]_\rho(X).$$

The equality holds for every compatible ρ and every X, establishing semantic equivalence. □

Hence the compiler rewrite

$$\boxed{\langle Q_{\theta_Q}, K_{\theta_K}, V_{\theta_V}\rangle \longrightarrow \text{QKVFuse}_{\theta_Q,\theta_K,\theta_V}}$$

is a certified architecture-level optimization.

For affine projections, the same proof applies after horizontally concatenating the three weight matrices and concatenating their bias vectors.

The theorem establishes correctness of the fusion transformation; it does not by itself assert a performance improvement. Runtime benefit depends on the backend execution model, memory traffic, kernel-launch cost, and hardware characteristics.

5.5 Other Fused Operations

A backend or TIR extension may introduce additional operations such as fused attention, fused normalization, or combined feedforward kernels.

For a proposed replacement

$$P \longrightarrow F,$$

the optimization is admitted only after establishing

$$\forall \rho, \qquad [\![P]\!]_\rho = [\![F]\!]_\rho$$

under its stated preconditions.

Unlike QKV fusion, no blanket semantic identity is assumed for such operations: each fused primitive carries its own correctness obligation.

5.6 Rewrite Sequences

Proposition 5.8 (Certified Rewrite Sequence Preservation)**.** *Suppose*

$$P = P_0 \longrightarrow P_1 \longrightarrow \cdots \longrightarrow P_k = Q$$

is a finite sequence of uniformly semantics-preserving TIR rewrites. Then

$$P \simeq Q.$$

Proof. For every compatible environment ρ, each rewrite gives

$$[\![P_i]\!]_\rho = [\![P_{i+1}]\!]_\rho.$$

Transitivity yields

$$[\![P]\!]_\rho = [\![Q]\!]_\rho.$$

Since this holds for every ρ,

$$P \simeq Q.$$

□

This is the operational compiler invariant: a finite pipeline assembled from certified transformations remains semantics preserving.

5.7 Normal Forms and Pass Scheduling

Definition 5.9 (Normal Form Relative to a Rule Set)**.** For a rewrite system $\mathcal{R}$, a well-formed raw TIR term N is in $\mathcal{R}$-normal form when no rule of $\mathcal{R}$ applies to N.

An implementation need not obtain normal forms by unrestricted rewriting. A compiler may instead use an ordered sequence of deterministic passes.

For example, let

$$T_1, \ldots, T_k$$

be terminating compiler passes, each constructed from certified rewrites. Define

$$N(P) = T_k \circ \cdots \circ T_1\big(N_{\mathrm{str}}(P)\big).$$

Corollary 5.10 (Certified Normalization Preservation)**.** *If every transformation performed by*

$$T_1, \ldots, T_k$$

is uniformly semantics preserving, then

$$\boxed{N(P) \simeq P.}$$

Proof. Structural Normalization Correctness gives

$$N_{\mathrm{str}}(P) \simeq P.$$

Certified Rewrite Sequence Preservation applies to the subsequent finite pass sequence. Transitivity of semantic equivalence gives the result. □

This formulation closely matches an executable compiler: normalization is an ordered pass pipeline whose individual transformations have local correctness obligations.

5.8 Termination and Confluence Scope

Termination and confluence are properties of a particular oriented rewrite system, not consequences of semantic preservation. Standard term-rewriting theory supplies the relevant general criteria [10, 11].

Transformer Algebra therefore makes no unconditional claim that every possible collection of TIR optimization rules is terminating or confluent.

For the deterministic structural normalization

$$N_{\mathrm{str}},$$

termination follows directly from structural recursion on a finite TIR term, and idempotence was established above.

For additional compiler passes, termination must be proved from the concrete pass definition or guaranteed operationally by a finite traversal strategy. If an implementation instead permits unrestricted nondeterministic rewriting, confluence or another strategy-independence result must be established for that particular rule system before uniqueness of resulting normal forms may be claimed.

This distinction prevents generic rewriting metatheory from being presented as a transformer-specific result.

5.9 From Normalization to Lowering

Normalization and backend lowering are distinct compiler stages.

Normalization operates within TIR:

$$P \longmapsto N(P),$$

while backend lowering translates the resulting TIR program into a target representation:

$$N(P) \longmapsto C_B(N(P)).$$

The compiler pipeline is therefore

$$\boxed{P \quad \longrightarrow \quad N_{\mathrm{str}}(P) \quad \longrightarrow \quad N(P) \quad \longrightarrow \quad C_B(N(P)).}$$

The first stage selects a deterministic structural representation. The second applies certified architecture transformations. The third lowers the semantically equivalent TIR program into a backend.

Section 6 formalizes this final lowering stage and the semantic obligations required for compiler correctness.

6 Backend Compilation and Semantic Preservation

A Transformer Intermediate Representation separates the meaning of an architecture from the concrete operations used to execute it. We now formalize the second half of this separation: compilation from TIR into an executable backend.

The central requirement is not that a backend reproduce the syntax or physical representation of a TIR program. A backend may fuse operations, alter storage layouts, change parameter formats, select specialized kernels, or otherwise change representation. Correctness instead requires a commuting relationship between source denotation and backend execution.

Throughout this section, fix a parameter signature Θ.

6.1 Backend Models

Definition 6.1 (Backend Model)**.** A backend model B consists of:

1. a collection of backend types;
2. well-formed backend programs between those types;
3. backend identity and sequential-composition operations;
4. backend unit and product structure sufficient to interpret the structural constructors of TIR;
5. a backend parameter environment
$$\mathrm{Env}_B(\Theta);$$
and
6. an execution semantics
$$[\![-]\!]_{B,\rho_B}$$
assigning a mathematical map to each well-formed backend program under $\rho_B \in \mathrm{Env}_B(\Theta)$.

The backend model is abstract. Concrete realizations may include tensor libraries, computational graphs, interchange representations, accelerator runtimes, or hardware-oriented target languages.

6.2 Type Lowering and Representation Maps

Compilation first assigns a backend representation to each TIR type.

Definition 6.2 (Type Lowering)**.** For backend B, a type-lowering map
$$\tau_B$$
assigns to every TIR type A a corresponding backend type
$$\tau_B(A).$$

For example,

$$X_{n,d}$$

may lower to a tensor type carrying shape (n, d) together with backend-specific layout, device, precision, batching, or memory information.

The source and backend semantic domains need not be literally identical.

Definition 6.3 (Representation Map)**.** For every TIR type A, a backend representation map is a function

$$e_A^B : [\![A]\!] \to [\![\tau_B(A)]\!]_B$$

that embeds or represents source semantic values in the backend semantic domain.

The representation maps are required to respect the structural products used by TIR. In particular, under the backend product interpretation,

$$e_{\mathbf{1}}^B(*) = *_B,$$

and

$$e_{A\times D}^B(a, d) = \big(e_A^B(a), e_D^B(d)\big).$$

This requirement is semantic rather than physical: a backend may store a logical product using tuples, graph edges, registers, buffers, structured values, or another representation, provided its semantics agrees with the declared product interface.

6.3 Parameter Lowering

Source parameter symbols are interpreted by

$$\rho \in \mathrm{Env}(\Theta).$$

A backend may require different layouts, dtypes, packing conventions, or parameter objects.

Definition 6.4 (Parameter Lowering)**.** A backend parameter lowering is a map

$$\lambda_B : \mathrm{Env}(\Theta) \to \mathrm{Env}_B(\Theta)$$

that converts a compatible source parameter environment into the parameter environment consumed by the backend.

Parameter lowering may therefore change representation without changing the source-level parameter signature.

6.4 Primitive Lowering

Definition 6.5 (Primitive Lowering)**.** For every primitive TIR operation

$$g : A \to D,$$

the compiler assigns a well-formed backend program

$$C_B(g) : \tau_B(A) \to \tau_B(D).$$

Primitive lowering need not preserve syntactic decomposition. One primitive may lower to several backend operations, and a backend-specific primitive implementation may itself use a fused kernel.

The semantic obligation is the following commuting square.

Definition 6.6 (Locally Correct Primitive Lowering)**.** A primitive lowering

$$C_B(g) : \tau_B(A) \to \tau_B(D)$$

is locally correct when, for every compatible source parameter environment

$$\rho \in \mathrm{Env}(\Theta),$$

$$\boxed{[\![C_B(g)]\!]_{B,\lambda_B(\rho)} \circ e_A^B = e_D^B \circ [\![g]\!]_\rho.}$$

Thus the target implementation need not operate on the same representation as the source denotation. It must agree after accounting explicitly for input, output, and parameter representation.

6.5 Compilation of Raw TIR Syntax

Compilation is defined on the raw syntax introduced in Section 2.

For a primitive,

$$C_B(g)$$

is its declared primitive lowering.

For identities,

$$C_B(\mathrm{id}_A) = \mathrm{id}^B_{\tau_B(A)}\,.$$

For terminal maps,

$$C_B(!_A) = !^B_{\tau_B(A)}.$$

For projections,

$$C_B(\pi_1) = \pi_1^B, \qquad C_B(\pi_2) = \pi_2^B.$$

For sequential composition,

$$C_B(Q \circ P) = C_B(Q) \circ_B C_B(P).$$

For pairing,

$$C_B(\langle P, Q\rangle) = \langle C_B(P), C_B(Q)\rangle_B\,.$$

The derived product operation is therefore compiled through its TIR definition:

$$C_B(P \times Q) = C_B\left(\langle P \circ \pi_1, Q \circ \pi_2\rangle\right).$$

This distinction is important. Common-input fan-out is compiled from pairing; independent-input product computation is a derived construction rather than an additional source constructor.

Definition 6.7 (Compositional Compiler)**.** A backend compiler C_B is compositional when it is defined on every raw TIR constructor by the clauses above and the corresponding backend structural operations have their declared denotational meaning.

6.6 Compositional Compiler Correctness

We now state the principal exact compiler theorem.

Theorem 6.8 (Compositional Compiler Correctness)**.** *Suppose that:*

1. *τ_B assigns a well-formed backend type to every TIR type;*
2. *the representation maps e_A^B satisfy the unit and product coherence conditions above;*
3. *λ_B is defined for every compatible source parameter environment;*
4. *every primitive lowering is locally correct;*
5. *backend identity, terminal maps, projections, composition, and pairing have the corresponding mathematical denotations.*

Then for every well-formed raw TIR program

$$\Theta \vdash P : A \to D$$

and every

$$\rho \in \mathrm{Env}(\Theta),$$

$$\boxed{[\![C_B(P)]\!]_{B,\lambda_B(\rho)} \circ e_A^B = e_D^B \circ [\![P]\!]_\rho.}$$

Proof. Proceed by structural induction on the raw syntax of P.

If P is a primitive g, the result is exactly local primitive correctness.

If

$$P = \mathrm{id}_A,$$

then

$$[\![C_B(P)]\!]_{B,\lambda_B(\rho)} \circ e_A^B = \mathrm{id} \circ e_A^B = e_A^B = e_A^B \circ [\![\mathrm{id}_A]\!]_\rho.$$

If

$$P = !_A,$$

both sides are the unique represented map from

$$[\![A]\!]$$

to the backend interpretation of the unit object, by unit coherence.

For a projection

$$P = \pi_i : A_1 \times A_2 \to A_i,$$

product coherence gives

$$[\![\pi_i^B]\!]\left(e_{A_1 \times A_2}^B(a_1, a_2)\right) = e_{A_i}^B(a_i),$$

which is the required commuting equation.

Suppose

$$P = Q \circ R$$

with

$$R : A \to C, \qquad Q : C \to D.$$

Then

$$\begin{aligned}\llbracket C_B(P)\rrbracket_{B,\lambda_B(\rho)} \circ e_A^B &= \llbracket C_B(Q)\rrbracket_{B,\lambda_B(\rho)} \circ \llbracket C_B(R)\rrbracket_{B,\lambda_B(\rho)} \circ e_A^B \\ &= \llbracket C_B(Q)\rrbracket_{B,\lambda_B(\rho)} \circ e_C^B \circ \llbracket R\rrbracket_\rho \\ &= e_D^B \circ \llbracket Q\rrbracket_\rho \circ \llbracket R\rrbracket_\rho \\ &= e_D^B \circ \llbracket Q \circ R\rrbracket_\rho,\end{aligned}$$

using the induction hypotheses for R and Q.

Finally suppose

$$P = \langle R, Q\rangle : A \to C \times D.$$

For every

$$x \in \llbracket A\rrbracket,$$

the backend semantics of pairing and the two induction hypotheses give

$$\begin{aligned}\llbracket C_B(P)\rrbracket_{B,\lambda_B(\rho)}\big(e_A^B(x)\big) &= \big(e_C^B(\llbracket R\rrbracket_\rho(x)), e_D^B(\llbracket Q\rrbracket_\rho(x))\big) \\ &= e_{C\times D}^B\left(\llbracket R\rrbracket_\rho(x), \llbracket Q\rrbracket_\rho(x)\right) \\ &= e_{C\times D}^B\left(\llbracket \langle R, Q\rangle\rrbracket_\rho(x)\right).\end{aligned}$$

These cases exhaust the raw TIR constructors. □

The theorem reduces whole-program exact correctness to local primitive obligations plus the structural contracts of the backend.

6.7 Compilation and Structural Equivalence

Structural equivalence is sound for source denotation, so correct compilation cannot distinguish structurally equivalent programs on represented source inputs.

Proposition 6.9 (Compilation Respects Structural Equivalence)**.** *If*

$$\Theta \vdash P, Q : A \to D$$

and

$$P \equiv_{\mathrm{str}} Q,$$

then for every compatible ρ,

$$\llbracket C_B(P)\rrbracket_{B,\lambda_B(\rho)} \circ e_A^B = \llbracket C_B(Q)\rrbracket_{B,\lambda_B(\rho)} \circ e_A^B.$$

Proof. Structural Soundness gives

$$\llbracket P\rrbracket_\rho = \llbracket Q\rrbracket_\rho.$$

Apply the Compositional Compiler Correctness Theorem to both programs. □

6.8 Correctness After Source Normalization

Let

$$N(P) : A \to D$$

be any well-typed source normalization or certified transformation satisfying

$$N(P) \simeq P.$$

Corollary 6.10 (Normalized Compilation Correctness)**.** *For every compatible ρ,*

$$\boxed{[\![C_B(N(P))]\!]_{B,\lambda_B(\rho)} \circ e_A^B = e_D^B \circ [\![P]\!]_\rho.}$$

Proof. Compiler correctness gives the commuting equation for $N(P)$, and

$$N(P) \simeq P$$

replaces its source denotation by that of P. □

Thus structural normalization, QKV fusion, and other certified source passes may precede backend lowering without changing source meaning.

6.9 Backend-Specific Optimization

Backend optimization occurs after source compilation.

Write

$$F \longrightarrow_B F'$$

for a target-level transformation.

Definition 6.11 (Backend Optimization Correctness)**.** A backend optimization is semantics preserving when, for every compatible backend parameter environment ρ_B,

$$[\![F']\!]_{B,\rho_B} = [\![F]\!]_{B,\rho_B}.$$

Proposition 6.12 (Target Optimization Preservation)**.** *Suppose*

$$F_0 = C_B(P)$$

and

$$F_0 \longrightarrow_B F_1 \longrightarrow_B \cdots \longrightarrow_B F_r$$

is a finite sequence of semantics-preserving backend transformations. Then for every compatible source environment ρ,

$$[\![F_r]\!]_{B,\lambda_B(\rho)} \circ e_A^B = e_D^B \circ [\![P]\!]_\rho.$$

Proof. The target transformations preserve backend denotation, so

$$[\![F_r]\!]_{B,\lambda_B(\rho)} = [\![C_B(P)]\!]_{B,\lambda_B(\rho)}.$$

The result follows from Compositional Compiler Correctness. □

6.10 Cross-Backend Semantic Agreement

Correct compilation gives a representation-aware form of cross-backend agreement.

Let B_1 and B_2 be backends satisfying the compiler theorem. Then for every source input

$$x \in [\![A]\!]$$

and every compatible ρ,

$$[\![C_{B_i}(P)]\!]_{B_i,\lambda_{B_i}(\rho)} \left(e_A^{B_i}(x)\right) = e_D^{B_i}\left([\![P]\!]_\rho(x)\right), \qquad i \in \{1,2\}.$$

This does not assert literal equality between values living in different backend domains.

If a backend additionally supplies an observation or decoding map

$$d_D^B : [\![\tau_B(D)]\!]_B \to [\![D]\!]$$

satisfying

$$d_D^B \circ e_D^B = \mathrm{id}_{[\![D]\!]},$$

then

$$d_D^B \left([\![C_B(P)]\!]_{B,\lambda_B(\rho)}(e_A^B(x)) \right) = [\![P]\!]_\rho(x).$$

Thus decoded outputs of correct backends agree through their common source denotation.

6.11 Exact and Finite-Precision Semantics

The theorem of this section is an exact theorem about the declared mathematical semantics of source and backend models.

Physical execution may instead use floating-point, quantized, approximate, or otherwise finite-precision arithmetic. In that setting, exact commuting equalities need not hold bitwise.

Section 8 therefore treats numerical execution separately and derives a compositional error specification under explicit local assumptions.

6.12 Compiler Correctness as the Central Invariant

The central exact invariant of Transformer Algebra is

$$\boxed{[\![C_B(P)]\!]_{B,\lambda_B(\rho)} \circ e_A^B = e_D^B \circ [\![P]\!]_\rho.}$$

The remaining sections investigate semantic equivalence, certified transformations, end-to-end verified pipelines, concrete backend instantiations, and finite-precision execution.

7 Architecture Equivalence and Verified Transformation

Sections 2–6 separate transformer syntax, denotation, normalization, and backend lowering. We now combine these components into a theory of verified architecture transformation.

A transformation may substantially change the representation of a transformer program while remaining inside the same semantic architecture. This is the basis for normalization, fusion, canonicalization, compiler optimization, and other source-level passes.

Throughout this section, fix a parameter signature Θ.

7.1 Uniform Semantic Equivalence

Recall that for

$$\Theta \vdash P, Q : A \to D,$$

semantic equivalence is defined by

$$P \simeq Q$$

when

$$\forall \rho \in \mathrm{Env}(\Theta), \qquad [\![P]\!]_\rho = [\![Q]\!]_\rho.$$

Thus semantic equivalence is architecture-level rather than parameter-instance-specific.

Because Section 3 established that $\simeq$ is a typed congruence, equivalent subprograms may be exchanged inside every well-typed TIR context.

Proposition 7.1 (Semantic Equivalence Relation)**.** *For every pair of TIR types A, D, semantic equivalence on well-formed terms*

$$\Theta \vdash P : A \to D$$

is reflexive, symmetric, and transitive.

Proof. For every parameter environment ρ, the relation is equality of mathematical functions

$$[\![A]\!] \to [\![D]\!].$$

Reflexivity, symmetry, and transitivity therefore follow pointwise in ρ. □

7.2 Architecture Transformations

Definition 7.2 (Typed Architecture Transformation)**.** A typed architecture transformation is a partial map

$$T : P \mapsto T(P)$$

on well-formed raw TIR terms such that whenever

$$\Theta \vdash P : A \to D$$

lies in the domain of T,

$$\Theta \vdash T(P) : A \to D.$$

The transformation may change syntax, sharing structure, primitive selection, or internal representation, but it preserves the external TIR type.

Definition 7.3 (Correct Architecture Transformation)**.** A typed architecture transformation T is semantically correct when

$$T(P) \simeq P$$

for every P in its domain.

Equivalently,

$$\forall \rho \in \mathrm{Env}(\Theta), \qquad [\![T(P)]\!]_\rho = [\![P]\!]_\rho.$$

The transformation theory therefore quantifies over all compatible parameter assignments rather than validating a pass on one accidental set of weights.

7.3 Transformation Passes

Compiler passes are commonly assembled from many local rewrites.

Let

$$T_1, \ldots, T_k$$

be typed architecture transformations, and define

$$P_0 = P,$$

$$P_i = T_i(P_{i-1}), \qquad 1 \le i \le k,$$

whenever every transformation is defined.

Proposition 7.4 (Composition of Correct Transformation Passes)**.** *If every T_i is semantically correct, then*

$$P_k \simeq P.$$

Proof. For every i,

$$P_i \simeq P_{i-1}.$$

Transitivity of semantic equivalence therefore gives

$$P_k \simeq P_0 = P.$$

□

This proposition allows a compiler pipeline to be verified modularly. Each pass may carry its own semantic proof obligation, while correctness of the finite sequence follows compositionally.

The structural normalization and certified rewrites of Section 5 are special cases of this transformation model.

7.4 Transformation Certificates

A practical compiler requires a representation of the evidence associated with verified transformations.

Definition 7.5 (Transformation Certificate)**.** For well-formed terms

$$\Theta \vdash P, Q : A \to D,$$

a transformation certificate is evidence establishing

$$P \simeq Q$$

under explicitly stated side conditions.

A certificate may contain, for example:

- an invocation of a previously proved TIR rewrite theorem;
- an algebraic derivation;
- shape and typing evidence;
- parameter-layout side conditions;
- a mechanically checked equality; or
- another proof object accepted by the compiler's trusted verification layer.

The compiler implementation may therefore distinguish:

candidate transformation

from

evidence that the transformation is semantically valid.

A transformation is admitted to the verified pipeline only when its required certificate has been established or accepted by the trusted checker.

7.5 Verified QKV Fusion as a Compiler Pass

Section 5 proved

$$\mathrm{QKVFuse}_{\theta_Q,\theta_K,\theta_V} \simeq \langle Q_{\theta_Q}, K_{\theta_K}, V_{\theta_V} \rangle .$$

Therefore the rewrite

$$\langle Q_{\theta_Q}, K_{\theta_K}, V_{\theta_V} \rangle \longrightarrow \mathrm{QKVFuse}_{\theta_Q,\theta_K,\theta_V}$$

defines a correct architecture transformation.

By contextual congruence, the same replacement may be performed inside a larger attention or transformer program.

For example, replacing the QKV projection subterm inside a complete attention head preserves the denotation of the enclosing program for every compatible parameter environment.

This provides a concrete example of the intended verification pattern:

$$\text{local algebraic theorem} \Longrightarrow \text{certified local rewrite} \Longrightarrow \text{correct compiler pass.}$$

Additional transformations such as fused attention require their own semantic proofs and are not assumed correct merely because a backend provides an operation with a similar name.

7.6 Correctness and Optimization Objectives

Semantic correctness and performance improvement are distinct properties.

Let

$$\kappa_B(P)$$

be an implementation-oriented cost model associated with a target backend B. Depending on the backend, it may estimate quantities such as

- kernel-launch count;
- memory traffic;
- intermediate storage;
- graph size;
- execution depth;
- communication volume; or
- hardware-specific resource usage.

Definition 7.6 (Verified Optimization)**.** A transformation T is a verified optimization relative to κ_B on a specified class of programs when

$$T(P) \simeq P$$

and

$$\kappa_B(T(P)) \leq \kappa_B(P)$$

for every program in that class.

The semantic equality and cost inequality require separate justification.

In particular, the QKV Fusion Correctness Theorem proves semantic preservation but does not, by itself, prove that fusion is faster on every backend. Performance depends on the execution and hardware cost model.

7.7 Semantic Quotient of TIR

Semantic equivalence provides a notion of architecture identity more abstract than raw syntax or structural equivalence.

For a well-formed program P, define its semantic equivalence class by

$$[P]_{\simeq} = \{Q \mid Q \simeq P\}.$$

Proposition 7.7 (Semantic Quotient Category)**.** *Because semantic equivalence is a typed congruence, the quotient*

$$\mathcal{T}_{\mathrm{IR}}(\Theta)/{\simeq}$$

is a well-defined category whose morphisms are semantic-equivalence classes of TIR programs.

Proof. The objects are the TIR types.

If

$$[P]_{\simeq} : A \to D$$

and

$$[Q]_{\simeq} : D \to E,$$

define

$$[Q]_{\simeq} \circ [P]_{\simeq} = [Q \circ P]_{\simeq}.$$

Because $\simeq$ is a congruence, choosing different representatives of either equivalence class produces a semantically equivalent composite. Hence composition is well defined.

Identity classes are

$$[\mathrm{id}_A]_{\simeq},$$

and the category laws follow from the corresponding TIR laws. □

Thus a semantic architecture may be regarded as an equivalence class of program representations rather than one privileged syntax tree.

Compiler optimization moves between representatives of the same semantic class.

7.8 Compilation of Equivalent Architectures

Section 6 established that an exactly correct compiler satisfies

$$[\![C_B(P)]\!]_{B,\lambda_B(\rho)} \circ e_A^B = e_D^B \circ [\![P]\!]_\rho.$$

This immediately yields the correct representation-sensitive formulation of compiler invariance.

Proposition 7.8 (Compilation Respects Semantic Equivalence)**.** *Let*

$$\Theta \vdash P, Q : A \to D$$

satisfy

$$P \simeq Q.$$

If C_B satisfies the Compositional Compiler Correctness Theorem, then for every compatible ρ,

$$\boxed{[\![C_B(P)]\!]_{B,\lambda_B(\rho)} \circ e_A^B = [\![C_B(Q)]\!]_{B,\lambda_B(\rho)} \circ e_A^B.}$$

Proof. Compiler correctness gives

$$[\![C_B(P)]\!]_{B,\lambda_B(\rho)} \circ e_A^B = e_D^B \circ [\![P]\!]_\rho,$$

and

$$[\![C_B(Q)]\!]_{B,\lambda_B(\rho)} \circ e_A^B = e_D^B \circ [\![Q]\!]_\rho.$$

Since

$$P \simeq Q,$$

we have

$$[\![P]\!]_\rho = [\![Q]\!]_\rho.$$

Substitution gives the result. □

The proposition deliberately asserts equality on backend values that represent source inputs through

$$e_A^B.$$

It does not require the two compiled target programs to have identical syntax or identical behavior on arbitrary backend values lying outside the image of the source representation map.

7.9 Verified Source-to-Target Pipelines

We now combine source transformation with correct backend compilation.

Let

$$P_0 = P,$$

and let

$$P_i = T_i(P_{i-1}), \qquad 1 \le i \le k,$$

where every T_i is a correct architecture transformation.

Let

$$F_0 = C_B(P_k)$$

be the compiled backend program.

A backend may then apply a finite sequence of semantics-preserving target optimizations

$$F_0 \longrightarrow_B F_1 \longrightarrow_B \cdots \longrightarrow_B F_r.$$

The final executable representation is F_r.

Theorem 7.9 (End-to-End Verified Pipeline Correctness)**.** *Suppose:*

1. $$\Theta \vdash P : A \to D;$$

2. *every source transformation*

   $$T_1, \ldots, T_k$$

   is semantically correct;

3. *C_B satisfies the Compositional Compiler Correctness Theorem; and*

4. *every backend transformation*

$$F_j \longrightarrow_B F_{j+1}$$

preserves backend denotation.

Then for every compatible parameter environment

$$\rho \in \mathrm{Env}(\Theta),$$

the final backend program satisfies

$$\boxed{[\![F_r]\!]_{B,\lambda_B(\rho)} \circ e_A^B = e_D^B \circ [\![P]\!]_\rho.}$$

Equivalently, for every

$$x \in [\![A]\!],$$

$$\boxed{[\![F_r]\!]_{B,\lambda_B(\rho)}\big(e_A^B(x)\big) = e_D^B\big([\![P]\!]_\rho(x)\big).}$$

Proof. Correctness of the source transformations gives

$$P_k \simeq P.$$

Hence for every ρ,

$$[\![P_k]\!]_\rho = [\![P]\!]_\rho.$$

Compiler correctness applied to P_k gives

$$[\![F_0]\!]_{B,\lambda_B(\rho)} \circ e_A^B = e_D^B \circ [\![P_k]\!]_\rho.$$

Therefore

$$[\![F_0]\!]_{B,\lambda_B(\rho)} \circ e_A^B = e_D^B \circ [\![P]\!]_\rho.$$

Every backend optimization preserves backend denotation, so

$$[\![F_r]\!]_{B,\lambda_B(\rho)} = [\![F_0]\!]_{B,\lambda_B(\rho)}.$$

Substituting this equality yields

$$[\![F_r]\!]_{B,\lambda_B(\rho)} \circ e_A^B = e_D^B \circ [\![P]\!]_\rho.$$

□

This theorem is the end-to-end exact correctness result for the Transformer Algebra compilation pipeline.

It covers the sequence

$$\boxed{\begin{aligned}&\text{source TIR} \longrightarrow \text{structural normalization} \longrightarrow \text{certified architecture passes}\\ &\qquad\qquad \longrightarrow \text{backend lowering} \longrightarrow \text{backend optimization} \longrightarrow \text{executable target.}\end{aligned}}$$

Every intermediate representation may change while the source denotation is preserved through the corresponding semantic representation maps.

7.10 Compiler Pass Contracts

The preceding results suggest a direct interface for an executable Transformer Algebra compiler.

A source-level compiler pass may expose a contract of the form

$$T : \mathrm{Term}_\Theta(A, D) \longrightarrow \mathrm{Term}_\Theta(A, D)$$

together with the semantic obligation

$$T(P) \simeq P.$$

A backend lowering exposes

$$C_B$$

together with the commuting obligation

$$[\![C_B(P)]\!]_{B,\lambda_B(\rho)} \circ e_A^B = e_D^B \circ [\![P]\!]_\rho.$$

A backend optimization exposes

$$F \longrightarrow_B G$$

together with the obligation

$$[\![F]\!]_{B,\rho_B} = [\![G]\!]_{B,\rho_B}.$$

These contracts localize the trusted correctness obligations of the future compiler.

7.11 Scope of Parameter-Preserving Transformations

The transformation theory developed above assumes a fixed source parameter signature Θ. This covers transformations such as structural normalization and the QKV fusion construction of Section 5, whose fused operation refers to the same underlying parameter symbols as the unfused program.

A transformation that changes the source parameter signature requires an additional parameter-environment translation

$$\phi : \mathrm{Env}(\Theta) \to \mathrm{Env}(\Theta')$$

and a generalized correctness condition of the form

$$[\![T(P)]\!]_{\phi(\rho)} = [\![P]\!]_\rho.$$

Such transformations fit naturally into the same framework, but are not required for the exact results proved in the present paper.

7.12 Verified Transformation as Architecture Preservation

The principal invariant of source transformation is therefore

$$\boxed{P \simeq T_k(\cdots T_1(P)\cdots).}$$

The principal invariant of backend compilation is

$$\boxed{[\![C_B(P)]\!]_{B,\lambda_B(\rho)} \circ e_A^B = e_D^B \circ [\![P]\!]_\rho.}$$

Combining them yields the end-to-end invariant

$$\boxed{[\![F_r]\!]_{B,\lambda_B(\rho)} \circ e_A^B = e_D^B \circ [\![P]\!]_\rho.}$$

Transformer architecture identity is therefore preserved through verified source transformation, backend lowering, and semantics-preserving target optimization even when every intermediate representation differs syntactically from the original program.

8 Backend Instantiations and Executable Examples

The preceding sections define backend compilation abstractly. We now illustrate how the same Transformer Intermediate Representation may be lowered into distinct executable targets and then separate exact semantic preservation from finite-precision numerical execution.

The backend sketches in this section are explanatory instantiations of the interfaces developed in Sections 6 and 7. They are not claims that the present paper contains completed verified implementations of PyTorch, JAX, or ONNX compilers.

8.1 A Representative Source Program

Consider the single-head TIR program

$$P_{\text{att}} = \text{Attend} \circ \langle \text{Norm} \circ \text{Score} \circ \langle Q, K \rangle, V \rangle : X_{n,d} \to H_{n,d_h}.$$

For

$$\rho(\theta_Q) = W_Q, \qquad \rho(\theta_K) = W_K, \qquad \rho(\theta_V) = W_V,$$

its concrete tensor denotation is

$$[\![P_{\text{att}}]\!]_\rho(X) = \text{softmax}_{\text{row}}\left(\frac{(XW_Q)(XW_K)^{\mathsf{T}}}{\sqrt{d_h}}\right)(XW_V).$$

This program serves as a common source specification for the backend examples below.

8.2 Simplified Tensor Backend Instantiations

For the explanatory tensor backends in this subsection, the logical source and target tensor domains may be identified, so one may take

$$e_A^B = \text{id}$$

for the displayed tensor types.

This simplification is only for the examples. The representation-aware theorem of Section 6 remains the general specification and permits different layouts, devices, precisions, memory spaces, packed representations, and graph values.

8.3 PyTorch-Style Lowering

PyTorch provides an imperative tensor programming model and optimized tensor operators [18].

Let

$$B_{\text{PT}}$$

denote a simplified PyTorch-style backend.

The three projection primitives may be lowered schematically as

$$C_{\mathrm{PT}}(Q)(X) = XW_Q, \qquad C_{\mathrm{PT}}(K)(X) = XW_K, \qquad C_{\mathrm{PT}}(V)(X) = XW_V.$$

The remaining head computation may be represented by

$$S = \frac{Q_X K_X^{\mathsf{T}}}{\sqrt{d_h}}, \qquad A = \mathrm{softmax}_{\mathrm{row}}(S), \qquad Y = AV_X.$$

For the general backend model, the local obligation for every primitive

$$g : A \to D$$

remains

$$[\![C_{\mathrm{PT}}(g)]\!]_{\mathrm{PT},\lambda_{\mathrm{PT}}(\rho)} \circ e_A^{\mathrm{PT}} = e_D^{\mathrm{PT}} \circ [\![g]\!]_\rho.$$

Once the primitive obligations and structural backend contracts hold, Compositional Compiler Correctness yields

$$[\![C_{\mathrm{PT}}(P_{\mathrm{att}})]\!]_{\mathrm{PT},\lambda_{\mathrm{PT}}(\rho)} \circ e_{X_{n,d}}^{\mathrm{PT}} = e_{H_{n,d_h}}^{\mathrm{PT}} \circ [\![P_{\mathrm{att}}]\!]_\rho.$$

8.4 JAX-Style Lowering

JAX represents numerical programs through composable transformations over array computations [19].

Let

$$B_{\mathrm{JAX}}$$

denote a simplified JAX-style backend. The same source projection primitives may be lowered through array dot operations:

$$Q_X = \mathrm{dot}(X, W_Q), \qquad K_X = \mathrm{dot}(X, W_K), \qquad V_X = \mathrm{dot}(X, W_V),$$

followed by score computation, row-wise normalization, and value aggregation.

Operational structure may differ from the PyTorch-style lowering, but the semantic obligation has the same form:

$$[\![C_{\mathrm{JAX}}(P_{\mathrm{att}})]\!]_{\mathrm{JAX},\lambda_{\mathrm{JAX}}(\rho)} \circ e_{X_{n,d}}^{\mathrm{JAX}} = e_{H_{n,d_h}}^{\mathrm{JAX}} \circ [\![P_{\mathrm{att}}]\!]_\rho.$$

8.5 ONNX-Style Graph Lowering

ONNX provides a graph-oriented interchange representation for machine-learning computations [20].

Let

$$B_{\mathrm{ONNX}}$$

denote a simplified ONNX-style backend.

The source fan-out

$$\langle Q, K, V \rangle$$

does not require physical duplication of X. A graph lowering may represent one source value with three consumer nodes that compute

$$Q_X, \qquad K_X, \qquad V_X.$$

The graph then applies score computation, scaling, row-wise softmax, and value aggregation.

Correctness is again representation aware:

$$[\![C_{\mathrm{ONNX}}(P_{\mathrm{att}})]\!]_{\mathrm{ONNX},\lambda_{\mathrm{ONNX}}(\rho)} \circ e^{\mathrm{ONNX}}_{X_{n,d}} = e^{\mathrm{ONNX}}_{H_{n,d_h}} \circ [\![P_{\mathrm{att}}]\!]_\rho.$$

Thus the source cartesian structure expresses semantic sharing while the target graph chooses its own representation of that sharing.

8.6 One Source, Multiple Targets

For a source program

$$\Theta \vdash P : A \to D$$

and any correct backend B_i, Section 6 gives

$$\boxed{[\![C_{B_i}(P)]\!]_{B_i,\lambda_{B_i}(\rho)} \left(e^{B_i}_A(x)\right) = e^{B_i}_D \left([\![P]\!]_\rho(x)\right).}$$

This is the appropriate cross-backend statement. Target values need not live in one common representation space.

If each backend supplies a decoder

$$d^{B_i}_D$$

satisfying

$$d^{B_i}_D \circ e^{B_i}_D = \mathrm{id}_{[\![D]\!]},$$

then

$$d^{B_i}_D \left([\![C_{B_i}(P)]\!]_{B_i,\lambda_{B_i}(\rho)}(e^{B_i}_A(x))\right) = [\![P]\!]_\rho(x).$$

Decoded outputs therefore agree through the common TIR denotation.

8.7 QKV Fusion and Backend Lowering

Section 5 established

$$\langle Q_{\theta_Q}, K_{\theta_K}, V_{\theta_V} \rangle \simeq \mathrm{QKVFuse}_{\theta_Q,\theta_K,\theta_V}.$$

A compiler may therefore replace the expanded source projection fan-out by the certified fused source operation before backend lowering.

A backend may then lower the three source parameter values to a concatenated matrix

$$W_{\mathrm{QKV}} = [W_Q\ W_K\ W_V]$$

or another equivalent physical parameter representation.

Three logically distinct claims are involved:

1. the source rewrite is correct by QKV Fusion Correctness;
2. parameter lowering correctly represents the source parameters; and
3. the backend implementation satisfies its local commuting obligation.

Keeping these layers separate prevents an implementation-specific packing decision from being mistaken for the source-level semantic theorem.

8.8 Multi-Head Compilation

For

$$\mathrm{Head}_i : X_{n,d} \to H_{n,d_h},$$

the source multi-head program is

$$P_{\mathrm{MHA}} = \mathrm{Merge} \circ \langle \mathrm{Head}_1, \ldots, \mathrm{Head}_m \rangle .$$

A backend may preserve the head decomposition, schedule heads independently, combine projection matrices, or replace a lowered region by a fused target kernel.

If a target optimization replaces

$$C_B(P_{\mathrm{MHA}})$$

by a backend program F_B, its target-level obligation is

$$[\![F_B]\!]_{B,\lambda_B(\rho)} = [\![C_B(P_{\mathrm{MHA}})]\!]_{B,\lambda_B(\rho)}$$

for every compatible ρ.

The end-to-end theorem of Section 7 then preserves the original TIR denotation.

8.9 Compilation of Transformer Blocks and Depth

Let

$$B_i : X_{n,d} \to X_{n,d}$$

be well-formed transformer blocks and

$$P_L = B_L \circ \cdots \circ B_1.$$

Compositional compilation gives

$$C_B(P_L) = C_B(B_L) \circ_B \cdots \circ_B C_B(B_1).$$

For every compatible ρ,

$$[\![C_B(P_L)]\!]_{B,\lambda_B(\rho)} \circ e^B_{X_{n,d}} = e^B_{X_{n,d}} \circ [\![P_L]\!]_\rho.$$

Thus the exact correctness argument scales structurally from primitive operations to heads, blocks, and finite-depth architectures.

8.10 Finite-Precision Semantics

The exact theorems above describe ideal mathematical backend semantics.

Physical execution generally uses floating-point or other finite-precision arithmetic governed by implementation-dependent rounding and approximation effects [17, 16].

Exact semantic preservation and numerical accuracy are therefore distinct verification problems.

For the numerical analysis below, consider TIR programs whose nontrivial numeric semantic domains are finite-dimensional real spaces equipped with chosen norms. Product domains use the max norm:

$$\|(x, y)\|_{\max} = \max\{\|x\|, \|y\|\}.$$

The unit domain carries the trivial zero distance.

8.11 Observed Numerical Interpretation

Suppose a backend implementation, together with its representation and observation maps, induces for each primitive

$$g : A \to D$$

an observed finite-precision map

$$\widetilde{[\![g]\!]}_{B,\rho} : [\![A]\!] \to [\![D]\!].$$

This map is the source-domain observation of the physical primitive execution. It may include encoding, parameter lowering, finite-precision execution, and decoding or observation.

Definition 8.1 (Numerical Backend Interpretation)**.** The observed numerical interpretation is extended compositionally from primitives to raw TIR syntax by

$$\widetilde{[\![\mathrm{id}_A]\!]}_{B,\rho} = \mathrm{id},$$

$$\widetilde{[\![!_A]\!]}_{B,\rho} = [\![!_A]\!],$$

$$\widetilde{[\![\pi_i]\!]}_{B,\rho} = [\![\pi_i]\!],$$

$$\widetilde{[\![Q \circ P]\!]}_{B,\rho} = \widetilde{[\![Q]\!]}_{B,\rho} \circ \widetilde{[\![P]\!]}_{B,\rho},$$

and

$$\widetilde{[\![\langle P, Q\rangle]\!]}_{B,\rho}(x) = \left(\widetilde{[\![P]\!]}_{B,\rho}(x), \widetilde{[\![Q]\!]}_{B,\rho}(x)\right).$$

The compositionality assumption models a backend observation boundary at the semantic level. A concrete compiler must justify that its chosen numerical interpretation satisfies this abstraction.

8.12 Local Numerical Obligations

For every numeric primitive

$$g : A \to D,$$

choose an admissible domain

$$\mathcal{D}_g \subseteq [\![A]\!].$$

Assume a local backend error bound

$$\left\| \widetilde{[\![g]\!]}_{B,\rho}(x) - [\![g]\!]_\rho(x) \right\| \le \varepsilon_g$$

for every

$$x \in \mathcal{D}_g.$$

Also assume that the ideal primitive is Lipschitz on the relevant admissible domain:

$$\|[\![g]\!]_\rho(x) - [\![g]\!]_\rho(y)\| \le L_g \|x - y\|.$$

The constants may depend on the backend, precision, parameter bounds, and admissible input region. No global Lipschitz claim for arbitrary transformer programs is assumed.

8.13 Compositional Error and Lipschitz Bounds

Define recursively an ideal Lipschitz bound $L(P)$ and an accumulated backend error bound $E_B(P)$.

For a primitive g,

$$L(g) = L_g, \qquad E_B(g) = \varepsilon_g.$$

For identity,

$$L(\mathrm{id}) = 1, \qquad E_B(\mathrm{id}) = 0.$$

For terminal maps,

$$L(!) = 0, \qquad E_B(!) = 0.$$

For projections under the max product norm,

$$L(\pi_i) = 1, \qquad E_B(\pi_i) = 0.$$

For pairing,

$$L(\langle P, Q\rangle) = \max\{L(P), L(Q)\},$$

and

$$E_B(\langle P, Q\rangle) = \max\{E_B(P), E_B(Q)\}.$$

For sequential composition,

$$L(Q \circ P) = L(Q)L(P),$$

and

$$\boxed{E_B(Q \circ P) = E_B(Q) + L(Q)E_B(P).}$$

8.14 Compositional Numerical Error Bound

Theorem 8.2 (Compositional Numerical Error Bound)**.** *Let*

$$\Theta \vdash P : A \to D$$

be a well-formed TIR program in the numerical fragment described above.

Assume:

1. *every primitive occurring in P satisfies its stated local numerical error bound;*
2. *every ideal primitive satisfies its stated Lipschitz bound;*
3. *all ideal and observed intermediate values generated while evaluating P remain inside the admissible domains on which the corresponding local error and Lipschitz bounds hold; and*
4. *the observed numerical backend interpretation is compositional as defined above.*

Then for every admissible input x and compatible parameter environment ρ,

$$\boxed{\left\| \widetilde{[\![P]\!]}_{B,\rho}(x) - [\![P]\!]_\rho(x) \right\| \le E_B(P).}$$

Proof. Proceed by structural induction on P.

The primitive case is the local numerical obligation.

Identity, terminal, and projection cases have zero observed error by definition.

For pairing

$$P = \langle R, Q \rangle,$$

the max product norm gives

$$\begin{aligned}
&\left\| \widetilde{[\![P]\!]}_{B,\rho}(x) - [\![P]\!]_\rho(x) \right\|_{\max} \\
&= \max\left\{ \left\| \widetilde{[\![R]\!]}_{B,\rho}(x) - [\![R]\!]_\rho(x) \right\|, \left\| \widetilde{[\![Q]\!]}_{B,\rho}(x) - [\![Q]\!]_\rho(x) \right\| \right\} \\
&\leq \max\{E_B(R), E_B(Q)\}.
\end{aligned}$$

For sequential composition

$$P = Q \circ R,$$

write

$$\widetilde{R} = \widetilde{[\![R]\!]}_{B,\rho}, \qquad R^* = [\![R]\!]_\rho,$$

and similarly for Q.

Then

$$\begin{aligned}
&\left\| \widetilde{Q}(\widetilde{R}(x)) - Q^*(R^*(x)) \right\| \\
&\leq \left\| \widetilde{Q}(\widetilde{R}(x)) - Q^*(\widetilde{R}(x)) \right\| + \left\| Q^*(\widetilde{R}(x)) - Q^*(R^*(x)) \right\| \\
&\leq E_B(Q) + L(Q) \left\| \widetilde{R}(x) - R^*(x) \right\| \\
&\leq E_B(Q) + L(Q) E_B(R).
\end{aligned}$$

The first inequality is the triangle inequality. The second uses the induction bound for Q at the observed intermediate input and the ideal Lipschitz bound for Q; the admissibility hypothesis ensures that both applications are valid. The final inequality uses the induction hypothesis for R.

These cases establish the recursive bound. □

8.15 Sequential Error Propagation

For a sequential program

$$P = g_k \circ g_{k-1} \circ \cdots \circ g_1$$

with

$$E_B(g_i) = \varepsilon_i \qquad \text{and} \qquad L(g_i) = L_i,$$

the recurrence gives

$$E_B(P) \leq \varepsilon_k + \sum_{i=1}^{k-1} \varepsilon_i \prod_{j=i+1}^{k} L_j.$$

Thus an early approximation may be amplified by the Lipschitz bounds of later ideal computations, while a late approximation contributes more directly to the final error.

8.16 Approximate Compiler Correctness

Definition 8.3 (Program-Specific Approximate Correctness)**.** A finite-precision backend realization is approximately correct for a program P on an admissible input class when

$$\left\| \widetilde{[\![P]\!]}_{B,\rho}(x) - [\![P]\!]_{\rho}(x) \right\| \leq \varepsilon$$

for every permitted ρ and input x in that class.

Under the hypotheses of the Compositional Numerical Error Bound, one may take

$$\varepsilon = E_B(P).$$

This does not replace the exact compiler theorem. The exact theorem specifies the ideal semantic contract of the compiler; the numerical theorem bounds the deviation of a particular finite-precision realization from that ideal.

8.17 Executable Validation

A concrete implementation may supplement proofs with executable validation.

For a source program P and backends $B_1, \ldots, B_k$, a validation harness may:

1. construct or load a source parameter environment ρ;
2. lower it to each $\lambda_{B_i}(\rho)$;
3. compile P into each backend;
4. encode common source inputs with $e_A^{B_i}$;
5. execute the target programs;
6. decode or observe outputs when appropriate; and
7. compare them with TIR reference semantics and any proved numerical error bounds.

Testing does not replace semantic proof. It can, however, expose failures of a concrete implementation to satisfy the formal backend assumptions.

8.18 Implementation Boundary

The examples make the intended implementation architecture explicit:

$$\boxed{\begin{aligned}&\text{Transformer language} \longrightarrow \text{typed TIR} \longrightarrow \text{certified transformations}\\ &\qquad\qquad\qquad\qquad\longrightarrow \text{backend lowering} \longrightarrow \text{executable target.}\end{aligned}}$$

TIR therefore serves as the semantic boundary between architecture description and executable realization, while backend-specific representation and finite-precision behavior remain explicit verification obligations.

9 Discussion

Transformer Algebra separates transformer architecture, denotational meaning, program transformation, backend compilation, and finite-precision execution.

The central perspective of the paper is that a transformer architecture can be treated as a typed program whose semantics is independent of any one execution framework.

The Transformer Intermediate Representation provides the formal boundary at which this separation becomes explicit. TIR records architecture structure and parameter references; denotational semantics assigns mathematical meaning; certified transformations alter representation while preserving that meaning; and backend compilation realizes the program in an executable target language.

This organization turns compiler correctness from an informal implementation expectation into a mathematical property.

9.1 From Transformer Equations to Compiler Semantics

A conventional mathematical specification of attention determines what an attention computation means.

For example,

$$\mathrm{softmax}_{\mathrm{row}}\left(\frac{(XW_Q)(XW_K)^{\mathsf{T}}}{\sqrt{d_h}}\right)(XW_V)$$

specifies the mathematical result of one attention head.

Such an equation does not by itself specify:

- how the architecture is represented as a compiler program;
- how shared inputs and branching are represented;
- when two architecture programs are semantically equivalent;
- which graph transformations are correct;
- how parameters are lowered into target representations;
- how source types correspond to backend values; or
- why an executable target preserves the source computation.

Transformer Algebra supplies these additional layers.

For a source program

$$\Theta \vdash P : A \to D,$$

the principal exact compiler invariant is

$$\boxed{[\![C_B(P)]\!]_{B,\lambda_B(\rho)} \circ e_A^B = e_D^B \circ [\![P]\!]_\rho.}$$

Thus exact correctness is not generally literal equality between source and target functions, because source and backend values may inhabit different semantic representation spaces.

The invariant instead states that the compiler diagram commutes.

9.2 Verified Compilation as a Compositional Property

The compiler theorem of Section 6 reduces global correctness to local obligations.

A backend implementation must provide:

- a type lowering;
- source-to-backend representation maps;
- parameter lowering;
- locally correct primitive implementations; and
- structurally correct interpretations of identity, composition, terminal maps, projections, and pairing.

Once these obligations are established, correctness of arbitrary finite TIR programs follows by structural induction.

This is important from a compiler-design perspective. The theorem does not require a separate proof for every complete transformer architecture. It provides a small trusted semantic interface from which correctness scales to attention heads, multi-head attention, transformer blocks, and finite-depth networks.

9.3 Architecture Identity

Transformer Algebra distinguishes three notions that should not be conflated.

Raw syntax records the concrete program representation manipulated by a compiler.

Structural equivalence

$$P \equiv_{\mathrm{str}} Q$$

identifies terms by the cartesian and categorical equations built into TIR.

Semantic equivalence

$$P \simeq Q$$

is generally coarser: it may identify programs that are not structurally equivalent whenever

$$\forall \rho \in \mathrm{Env}(\Theta), \qquad [\![P]\!]_\rho = [\![Q]\!]_\rho.$$

Structural soundness gives

$$P \equiv_{\mathrm{str}} Q \Longrightarrow P \simeq Q,$$

but the converse need not hold.

The semantic quotient

$$\mathcal{T}_{\mathrm{IR}}(\Theta)/\simeq$$

therefore identifies architecture programs by mathematical behavior rather than by one privileged syntax tree.

This provides a natural notion of architecture identity for optimization: different compiler representations may describe the same semantic architecture.

9.4 Transformation and End-to-End Correctness

The transformation theory of Sections 5 and 7 permits architecture programs to change before backend compilation.

If

$$P_k = T_k(\cdots T_1(P)\cdots)$$

and every source transformation is semantics preserving, then

$$P_k \simeq P.$$

If the compiled target is subsequently transformed by semantics-preserving backend optimizations to a final target program

$$F,$$

the end-to-end theorem gives

$$\boxed{[\![F]\!]_{B,\lambda_B(\rho)} \circ e_A^B = e_D^B \circ [\![P]\!]_\rho.}$$

This is the principal architecture-preservation statement of the framework.

Normalization may alter syntax.

QKV fusion may replace several source operations with one certified derived operation.

Backend lowering may change parameter layout and representation.

Target optimization may replace expanded target graphs with fused kernels.

Nevertheless, the source computation remains invariant when the corresponding semantic obligations are satisfied.

9.5 Compiler Optimization and Proof Obligations

Transformer Algebra deliberately separates semantic correctness from performance.

A transformation may be semantically valid without being faster.

For example, QKV fusion is justified by the algebraic identity

$$X[W_Q\ W_K\ W_V] = [XW_Q\ XW_K\ XW_V].$$

This proves equivalence of the expanded and fused projection computation.

It does not prove reduced latency, reduced energy, improved utilization, or reduced memory traffic on every backend.

Those are properties of a backend-dependent cost model.

The compiler therefore has two logically distinct questions:

Does the transformation preserve meaning?

and

Does the transformation improve the chosen cost objective?

Keeping these questions separate prevents implementation heuristics from being mistaken for semantic theorems.

9.6 Cross-Backend Meaning

Different backends need not use the same internal value representation.

A PyTorch-style runtime, a functional array system, a graph interchange representation, and a future accelerator backend may lower the same TIR program in structurally different ways.

Correct compilation does not require their target functions to be literally equal as mathematical functions on one common representation domain.

Instead, backend B_i must satisfy

$$[\![C_{B_i}(P)]\!]_{B_i,\lambda_{B_i}(\rho)}\left(e_A^{B_i}(x)\right) = e_D^{B_i}\left([\![P]\!]_\rho(x)\right).$$

If suitable decoding maps exist, decoded outputs agree with the same source denotation.

Backend independence is therefore semantic rather than representational.

9.7 Exact and Numerical Correctness

The framework distinguishes exact semantic correctness from finite-precision accuracy.

Exact correctness is expressed by the commuting equation

$$[\![C_B(P)]\!]_{B,\lambda_B(\rho)} \circ e_A^B = e_D^B \circ [\![P]\!]_\rho.$$

Physical implementations, however, generally use floating-point or other finite-precision arithmetic.

Section 8 therefore introduces a numerical interpretation

$$\widetilde{[\![P]\!]}_{B,\rho}$$

and a compositional error bound

$$\left\| \widetilde{[\![P]\!]}_{B,\rho}(x) - [\![P]\!]_\rho(x) \right\| \le E_B(P).$$

For sequential composition,

$$E_B(Q \circ P) = E_B(Q) + L(Q)E_B(P),$$

under the admissible-domain and Lipschitz hypotheses stated in Section 8.

This separates two verification layers.

The exact compiler theorem establishes semantic preservation in the ideal backend model.

The numerical theorem quantifies the discrepancy introduced by a concrete finite-precision realization.

A compiler implementation may satisfy the first while still requiring backend-specific numerical analysis for the second.

9.8 Relation to Categorical Models of Attention

Category-theoretic structure provides an important foundation for Transformer Algebra, but it is not the sole contribution.

Objects provide types, morphisms provide composable architecture operations, products represent structured fan-out and shared inputs, and categorical laws support compositional reasoning.

Related categorical treatments of transformer attention demonstrate that attention itself admits meaningful categorical structure [2].

Transformer Algebra has a different primary emphasis.

Its central object is not only a categorical model of attention, but a formal intermediate representation and compiler semantics for transformer architectures.

The framework combines:

- a typed architecture language;
- explicit raw compiler syntax;
- a cartesian structural quotient;
- parameter-indexed denotational semantics;
- certified source rewriting;
- backend type and parameter lowering;
- a compositional compiler-correctness theorem;
- verified end-to-end transformation pipelines; and
- compositional finite-precision error analysis.

Category theory therefore supplies part of the semantic foundation of the compiler rather than serving as an alternative notation for standard transformer equations.

9.9 Relationship to Compiler and Tensor-Verification Systems

The semantic-preservation formulation follows the general verified-compilation principle that a compiler should preserve the observable meaning of source programs [15, 14].

Modern compiler infrastructure also emphasizes extensible intermediate representations and staged lowering. MLIR provides a multi-level framework for domain-specific compiler construction across heterogeneous targets [21]. More recent work has made formal semantics and verification more explicit inside that ecosystem through first-class verification dialects [24].

Verified tensor compilation and rewrite verification are especially close adjacent areas. Liu et al. develop a verified compiler for a functional tensor language [22], while TensorRight automatically verifies classes of tensor-graph rewrites [23]. These systems show that semantic preservation and proof obligations are already central concerns in tensor compilation.

LLM-specific compiler work has also begun to use explicit high-level and hardware-oriented IR layers. Hu et al. describe an MLIR-based LLM compilation method that lowers a backend-independent graph dialect toward a TPU-oriented dialect carrying quantization, layout, and scheduling information [25].

The phrase "transformer compiler" is also used in the opposite direction. Zhai et al. study transformers *as* compilers and prove expressive-power results for compiler tasks [3]. The present paper instead studies compilation *of* transformer architectures into executable target representations.

Transformer Algebra therefore does not claim to be the first neural-network IR, tensor compiler, verified tensor compiler, or multi-level lowering framework. Its contribution is the combination of a transformer-architecture-specific typed IR, parameter-indexed compositional semantics, certified source transformations, representation-aware backend compilation, end-to-end semantic preservation, and compositional finite-precision error analysis.

The resulting compiler boundary differs from a conventional general-purpose language compiler in its domain-specific structure. TIR types carry transformer shape and architecture information, parameter symbols are explicit components of the source signature, and architecture-specific rewrites such as QKV fusion appear as certified source transformations.

9.10 Scope of the Present Results

The present paper establishes an abstract formal framework rather than a completed production compiler.

Its principal results include:

1. a typed cartesian architecture language for transformer computation;
2. a Transformer Intermediate Representation separating source architecture from backend implementation;
3. parameter-indexed denotational semantics;
4. concrete tensor semantics for attention and transformer blocks;
5. structural normalization and certified rewriting;
6. a proof of QKV fusion correctness;
7. a compositional backend compiler-correctness theorem;
8. an end-to-end theorem covering source transformation, lowering, and backend optimization; and
9. a compositional finite-precision error bound under explicit local assumptions.

Several stronger conclusions require additional hypotheses.

The paper does not claim that every possible TIR rewrite system is terminating or confluent.

It does not claim that a semantically correct optimization improves performance on every backend.

It does not claim bitwise equality across finite-precision implementations.

It does not claim that the PyTorch-, JAX-, or ONNX-style examples constitute completed verified backend implementations.

Correctness of a concrete implementation requires proof that its actual primitive lowerings, parameter transformations, graph passes, numerical assumptions, and runtime behaviors satisfy the contracts introduced by the formal model.

9.11 Implementation Roadmap

The formal structure suggests a direct implementation architecture.

A future Transformer Algebra compiler may be organized as

$$\begin{aligned}&\text{source language} \longrightarrow \text{typed TIR} \longrightarrow \text{verification} \longrightarrow \text{certified transformation}\\ &\qquad\qquad\qquad\qquad \longrightarrow \text{backend lowering} \longrightarrow \text{target optimization} \longrightarrow \text{execution}.\end{aligned}$$

The frontend constructs raw well-typed TIR syntax.

A type checker verifies sequence dimensions, head dimensions, parameter requirements, and product structure.

A transformation engine applies structural normalization and certified architecture rewrites.

A lowering layer converts source types and parameter environments into backend representations.

Backend-specific optimization passes transform the lowered target under their own semantic contracts.

A validation layer compares executable results with source reference semantics and with numerical bounds where exact arithmetic is unavailable.

This architecture follows directly from the formal distinctions established in the paper rather than being imposed as an unrelated implementation plan.

9.12 Mechanized Verification

The definitions and proofs developed here are suitable candidates for future mechanization in a proof assistant.

A mechanized development could formalize:

- TIR types and raw syntax;
- well-formedness and typing;
- structural equivalence;
- parameter environments;
- denotational semantics;
- semantic equivalence;
- certified rewrite rules;
- QKV fusion correctness;
- backend compiler contracts; and
- the compositional compiler-correctness theorem.

Such mechanization would strengthen the trusted semantic core of an eventual compiler.

It is, however, a separate implementation and formalization task rather than a claim of the present paper.

9.13 Extensions

The framework admits natural extensions.

Masked attention already fits the same semantic discipline through typed mask parameters and masked normalization.

Sparse, local, grouped, structured, linearized, or architecture-specific attention variants may be represented by extending the primitive signature and supplying corresponding semantics.

Alternative normalization operations, feedforward structures, positional mechanisms, routing systems, mixture-of-experts components, and residual organizations may be introduced in the same way.

Additional source types may be required for richer architectures.

Hardware-oriented backends may enrich type lowering with layout, memory, device, scheduling, precision, and resource information.

Such extensions should preserve the central separation among:

source syntax, mathematical denotation, certified transformation,
backend representation, execution.

This separation is the principal extensibility criterion of Transformer Algebra.

9.14 From Formal Specification to Compiler Discovery

An important consequence of the framework is methodological.

Designing the semantics and designing the compiler are not independent tasks.

Attempting to state a correct compiler exposes requirements that may remain hidden in a purely architectural description.

In the present development, compiler-oriented reasoning motivates explicit treatment of shared fan-out, parameter environments, raw syntax, structural quotients, representation maps, parameter lowering, local primitive obligations, transformation certificates, and numerical error propagation.

Conversely, the formal semantics constrains the implementation by making its correctness obligations explicit.

The resulting interaction is therefore iterative:

$$\boxed{\begin{aligned}&\text{formal architecture} \longrightarrow \text{compiler requirements} \longrightarrow \text{semantic refinement}\\ &\qquad\qquad\qquad\qquad \longrightarrow \text{stronger compiler contracts.}\end{aligned}}$$

Transformer Algebra is intended to support this process while keeping the mathematical specification independent of any one implementation.

10 Conclusion

Transformer Algebra develops a formal semantics and compilation framework for transformer architectures.

Its central construction is the Transformer Intermediate Representation: a typed compositional architecture language separating source-level transformer structure from concrete execution backends.

TIR makes explicit the components required for correct compilation: architectural types, parameter signatures, shared dataflow, denotational semantics, structural normalization, certified transformation, parameter lowering, backend representation, and executable interpretation.

The principal exact correctness invariant is the commuting equation

$$\boxed{[\![C_B(P)]\!]_{B,\lambda_B(\rho)} \circ e_A^B = e_D^B \circ [\![P]\!]_\rho.}$$

For every well-formed source program

$$\Theta \vdash P : A \to D,$$

this equation states that representing a source input and executing the compiled target produces the same semantic result as executing the source denotation and then representing its output.

Compiler correctness is therefore expressed without requiring source and backend values to inhabit identical representation spaces.

The framework extends this invariant across verified compilation pipelines.

If source transformations preserve TIR semantics, backend lowering satisfies the compiler-correctness theorem, and subsequent target transformations preserve backend denotation, then a final executable target F satisfies

$$\boxed{[\![F]\!]_{B,\lambda_B(\rho)} \circ e_A^B = e_D^B \circ [\![P]\!]_\rho.}$$

Thus normalization, architecture transformation, fusion, parameter lowering, backend optimization, and executable realization may alter program representation while preserving the meaning of the original transformer architecture.

The paper also separates exact semantic correctness from finite-precision accuracy.

Under explicit local approximation and Lipschitz assumptions, the numerical semantics admits a compositional whole-program error bound

$$\left\| \widetilde{[\![P]\!]}_{B,\rho}(x) - [\![P]\!]_\rho(x) \right\| \leq E_B(P),$$

with sequential error propagation governed by

$$E_B(Q \circ P) = E_B(Q) + L(Q)E_B(P).$$

This provides a quantitative bridge between ideal compiler semantics and finite-precision execution without conflating exact denotational preservation with bitwise equality.

The resulting perspective is that a transformer architecture is not merely a collection of tensor equations. It can be treated as a typed program with a mathematical semantics, while its realization in tensor libraries, computation graphs, accelerator runtimes, or future hardware targets can be treated as a compilation problem governed by explicit correctness contracts.

Transformer Algebra therefore provides a foundation for:

- backend-independent transformer representation;
- semantics-preserving architecture transformation;
- certified optimization;
- compositional compiler verification;
- cross-backend semantic reasoning;
- quantitative finite-precision analysis; and
- the construction of an executable transformer compiler and language.

The framework is intentionally implementation-independent, but its formal structure is designed to guide implementation.

In this sense, the semantics and the compiler are mutually informative: formalization exposes compiler requirements, while compiler design exposes semantic structure that must be made explicit.

The central principle of Transformer Algebra can therefore be summarized as

architecture meaning is invariant under correct compilation.

This invariant provides the semantic boundary between transformer architecture and its executable realization.

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," in *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[2] C. O'Neill, "Self-Attention as a Parametric Endofunctor: A Categorical Framework for Transformer Architectures," *arXiv preprint arXiv:2501.02931*, 2025.

[3] X. Zhai, R. Zhou, L. Zhang, and S. S. Du, "Transformers are Efficient Compilers, Provably," in *Proceedings of the Conference on Language Modeling (COLM)*, 2025.

[4] S. Mac Lane, *Categories for the Working Mathematician*, 2nd ed. Springer, 1998.

[5] S. Awodey, *Category Theory*, 2nd ed. Oxford University Press, 2010.

[6] B. C. Pierce, *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[7] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002.

[8] G. Winskel, *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

[9] G. D. Plotkin, "A Structural Approach to Operational Semantics," DAIMI FN-19, Department of Computer Science, Aarhus University, 1981.

[10] F. Baader and T. Nipkow, *Term Rewriting and All That*. Cambridge University Press, 1998.

[11] Terese, *Term Rewriting Systems*. Cambridge University Press, 2003.

[12] A. W. Appel, *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[13] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[14] X. Leroy, "Formal Verification of a Realistic Compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[15] X. Leroy, "Formal Certification of a Compiler Back-End or: Programming a Compiler with a Proof Assistant," in *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 42–54, 2006.

[16] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. SIAM, 2002.

[17] IEEE, *IEEE Standard for Floating-Point Arithmetic*, IEEE Std 754-2019, 2019.

[18] A. Paszke et al., "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[19] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: Composable Transformations of Python+NumPy Programs," 2018.

[20] J. Bai, F. Lu, K. Zhang, et al., "ONNX: Open Neural Network Exchange," 2019.

[21] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14, 2021.

[22] A. Liu, G. L. Bernstein, A. Chlipala, and J. Ragan-Kelley, "A Verified Compiler for a Functional Tensor Language," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 320–342, 2024.

[23] J. Arora, S. Lu, D. Jain, T. Xu, et al., "TensorRight: Automated Verification of Tensor Graph Rewrites," *Proceedings of the ACM on Programming Languages*, vol. 9, no. POPL, pp. 832–863, 2025.

[24] M. Fehr, Y. Fan, H. Pompougnac, J. Regehr, and T. Grosser, "First-Class Verification Dialects for MLIR," *Proceedings of the ACM on Programming Languages*, vol. 9, no. PLDI, pp. 1466–1490, 2025.

[25] P. Hu, Z. Xin, Y. Chen, Y. Zhou, and L. Wang, "An MLIR-Based Compilation Method for Large Language Models," *arXiv preprint arXiv:2607.15865*, 2026.