KOS-TL (KNOWLEDGE OPERATION SYSTEM TYPE LOGIC): A CONSTRUCTIVE FOUNDATION FOR EXECUTABLE KNOWLEDGE SYSTEMS

A PREPRINT

Chen Peng[*]
School of Information Science
Beijing University of Language and Culture
Beijing 100081
chenpeng@blcu.edu.cn

January 3, 2026

ABSTRACT

As knowledge representation shifts from static databases to evolving operational systems, traditional logics face limitations in handling event-driven state transitions. This paper proposes Knowledge Operation System Type Logic (KOS-TL), or "Zhi-Xing Logic" By integrating intuitionistic dependent type theory with small-step operational semantics, KOS-TL establishes a unified framework for static knowledge constraints, dynamic state evolution, and physical environment refinement.

Keywords Knowledge Operation System ·Dependent Types ·Operational Semantics ·Constructivism ·Formal Verification

1 Introduction

The core challenge of modern knowledge systems has shifted from conceptual modeling to the execution, traceability, and verification of operations within event-driven environments. Traditional Description Logic (DL) frameworks, rooted in static model-theoretic semantics, struggle to express "how knowledge is updated" and "how operations are executed."

Knowledge representation and reasoning is an important application domain of logic. However, with the proliferation of large-scale data integration and complex decision-making systems, the research objects in knowledge representation and reasoning are gradually shifting from static knowledge bases to continuously evolving knowledge operation systems. In knowledge operations oriented toward continuous development, concept modeling or ontology consistency verification is no longer the core focus; the core challenge has transformed into how to perform executable, traceable, and verifiable operations on knowledge in environments driven by events, state evolution, and strong engineering constraints.

In the face of new demands in the field of knowledge operations, existing mainstream logical frameworks (especially formal systems represented by Description Logic (DL) and its Semantic Web implementations such as OWL) exhibit fundamental mismatches with the aforementioned requirements in terms of theoretical assumptions and semantic structures. This is specifically manifested in the following aspects.

(1) Tension between static semantics and dynamic operations
Description Logic is based on static model-theoretic semantics, with its core reasoning problems revolving around concept satisfiability, concept inclusion, and instance checking. This paradigm assumes knowledge describes "possible states of the world" rather than "system runtime states." In contrast, the core objects in knowledge operations are events, operations, and state transitions, where the fundamental questions are no

[*] Chen Peng, Doctor of Computer Science, born in May 1979, from Nanfeng County, Jiangxi Province.

longer whether a certain assertion is true in some model, but whether a particular operation can be legally executed and how the system state evolves after its execution. The model-theoretic semantics of Description Logic centers on static interpretive structures, with the primary goal of characterizing "what the world might logically be like." Concepts are interpreted as subsets of the individual domain, roles as binary relations, and reasoning problems primarily focus on satisfiability, concept inclusion, and instance checking. This semantic structure is naturally suited for taxonomies, ontology engineering, and terminological reasoning. However, in the application scenarios faced by knowledge operations, knowledge is not a static collection but a state system that evolves continuously over time. The core issues are no longer "whether a certain assertion is true in some model," but "whether a certain event has occurred, whether a certain state has been updated, and which new facts these changes will trigger." Description Logic does not treat events and state transitions as first-class logical objects; its support for dynamic processes can only be achieved indirectly through external mechanisms or reification, which is costly in engineering terms and semantically opaque.

(2) Fundamental conflict between open world assumption and operational semantics
The open world assumption (Open World Assumption) adhered to by Description Logic fundamentally conflicts with the closed or semi-closed world semantics commonly used in knowledge operations. In engineering practice, missing information is often treated as an abnormal state or basis for operation failure, rather than logical "unknown." Description Logic adheres to the open world assumption (Open World Assumption, OWA), where unknown does not equate to false. This assumption is reasonable in Semantic Web and open knowledge environments but often becomes an obstacle in knowledge operations. In scenarios such as enterprise governance, risk control, and compliance auditing, "missing records" themselves constitute negative information or abnormal states. The semantics of knowledge operating systems are closer to the closed world assumption: facts that do not appear are regarded as non-occurring events, and unsatisfied constraints as system errors. This semantic orientation, centered on closed worlds and executable constraints, makes the existential reasoning of Description Logic models difficult to directly serve actual system operations.

(3) Differences between conceptual semantics and nominal type semantics
The conceptual semantics in Description Logic is extensional, with membership dynamically determined through reasoning; in practical knowledge operations, however, types more often play nominal and constraint roles. Whether an object belongs to a certain type is not derived through logical entailment but is a prerequisite that must be satisfied during data ingestion and operation phases, directly determining whether an operation is legal and whether the process can continue. Type errors manifest as system non-executable states rather than mere reasoning failures. This type semantics is more akin to type systems in programming languages and operating systems than traditional ontology logic.

(4) Fundamental shift in reasoning objectives
Finally, the reasoning objectives of Description Logic primarily involve proving logical entailment, whereas "reasoning" in knowledge operations is more akin to rule-driven fact materialization—i.e., under the given current data state, which new facts should be immediately generated, stored, and involved in subsequent computations[2]. The results of reasoning are not merely used to answer queries but directly alter the system's observable state and impose constraints on subsequent operations. This reasoning mode lacks direct formal characterization in classical logical frameworks.

In the field of knowledge operations, the issue is not whether logical systems are sufficiently powerful, but whether they can natively support events, time, state changes, and executable rules. Traditional Description Logic remains irreplaceable in static knowledge representation, but its logical assumptions and semantic structures are not suitable for direct use as the kernel of knowledge operating systems. The aforementioned differences reveal a key theoretical gap: traditional logical systems have yet to provide a unified formal foundation for systems where "knowledge is treated as an operable object." This directly gives rise to the demand for a new logical system: one that takes events and state transitions as core objects, employs type-driven and operational semantics for reasoning, incorporates built-in type constraints to characterize operational legality, supports rule-driven state evolution, and achieves an engineering-feasible balance between reliability and termination. To address this theoretical gap, we propose and study a new formal logical system—"Zhi-Xing Logic" (Knowledge Operation System Type Logic, abbreviated as KOS-TL). KOS-TL aims to build on intuitionistic type theory, introducing eventification and operational semantics, enabling the logical system to natively characterize the processes of knowledge comprehension, operation, and state updating, thereby providing a verifiable, executable, and extensible logical kernel for knowledge operating systems.

[2] This kind of "reasoning" is closer to rule-driven approaches such as Datalog, triggers, and operational semantics, rather than classical logical reasoning centered on proof theory or model theory.

2 The Architecture of KOS-TL

To integrate logical rigor with operational expressiveness, "Zhi-Xing Logic" (KOS-TL) is defined as a *layered formal system*. It consists of three distinct layers of formal definitions: *Core*, *Kernel*, and *Runtime*, each differentiated in terms of logical roles and semantic commitments (as shown in Table 1).

Table 1: Overview of KOS-TL Layered Structure

Layer	Formal Name	Core Responsibilities	Logical Objects	Decidability
L0 Core	Static Truth Layer (Logic)	Defines “what is valid.” Establishes type constructions and constraints based on intuitionistic dependent type theory (ITT).	Types T, proof terms p, propositions P	Strongly decidable
L1 Kernel	Dynamic Transition Layer (Dynamics)	Defines “how to change.” Introduces small-step operational semantics to handle event-driven state transitions.	States σ, events e, transitions $\rightarrow$	Locally decidable
L2 Runtime	Environment Evolution Layer (System)	Defines “how to run.” Handles external I/O, timeline mounting, and nondeterministic inputs.	Queues Q, external sources $\mathcal{E}_{\text{env}}$	Semi-decidable

The layering principles of "Zhi-Xing" Logic are as follows:

- Logical validity is determined solely by the core layer;
- Operational correctness is enforced by the kernel layer;
- System evolution is realized in the runtime layer.

This layered design ensures that the correctness of the core logic is maintained even in open, nondeterministic system evolutions.

2.1 L0: Core (Core Layer) — Static Logical Domain

2.1.1 Core Layer Requirements Analysis and Logical Construction

In terms of constructive expression requirements, traditional systems often focus solely on the static storage of data, whereas in high-security and high-trust scenarios, the system must transition to the storage of knowledge.” Based on Intuitionistic Type Theory **?**, each knowledge item is defined as a dependent pair $\Sigma(d : D).P(d)$, where data d is strongly coupled with the credential $P(d)$ that proves it satisfies the business ontology. This design eliminates rootless data” at the source, ensuring that all knowledge entering the kernel undergoes constructive verification. Deep alignment between physics and logic is another key requirement. Through the expressive power of dependent types (Dependent Types), the system can internalize physical laws and compliance constraints from industrial or financial domains, rather than relying on external ad-hoc logical judgments. This paradigm of “Make Illegal States Unrepresentable” ensures that any attempt to perform an operation violating axioms fails at the type-checking stage, thereby maintaining the system’s steady state. Furthermore, to address high-frequency compliance requirements, the system introduces computational reflexivity (Computational Reflexivity). By requiring the Core Layer to describe its own reduction rules, the system can automatically synthesize equivalence proofs $\text{Id}(t, t')$ during the execution of every small-step logical evolution (β, ι, δ reductions). This full-path automated auditing elevates traditional post-hoc investigation to runtime real-time formal verification. Finally, by defining elaboration operator (elab) templates, the Core Layer provides a semantic elevation benchmark for runtime (Runtime) signals, establishing a unique mapping path from physical bits (Bits) to logical truths (Truth).

2.1.2 Overall Architecture Description

The Core Layer is architected as a strongly verified microkernel based on dependent type theory, primarily driven by the following three functional modules: First, the type constructor and ontology manager act as the system’s legislators,responsible for transforming business domain ontologies into sorts (Sorts) and dependent type structures in the type system, clearly defining the boundaries of legal objects. Second, the reduction engine serves as the system’s reasoning machine,handling fine-grained evolution of logical terms by executing function applications and structural decompositions to compute the logical steady state after knowledge evolution. Third, the type checker acts as the system’s gatekeeper,” performing bidirectional type checking (Bidirectional Type Checking) to ensure that all operations entering the Kernel Layer satisfy pre-verified correctness (Correct-by-construction).

2.1.3 Key Design Decisions

The Core Layer's technology selections and architectural decisions reflect a balance between logical rigor and engineering feasibility.

(1) Replacement of First-Order Logic (FOL) with Dependent Type Theory (MLTT)
Traditional knowledge bases rely on FOL or description logics (DL), leading to a disconnect between logical assertions and concrete data. The Core Layer selects dependent type theory, utilizing Σ types to achieve atomic encapsulation of data and constraints. This decision resolves the persistent issue of evidence absence in knowledge operations, enforcing physical constraints at the architectural level.

(2) Introduction of Computational Reflexivity and Endogenous Auditing
To address the risks of traditional auditing being delayed and easily tampered with, the Core Layer models reduction rules as logical processing objects. Whenever a state changes, the system automatically synthesizes an identity proof (Identity Proof). This design transforms auditing behavior into an automated type-checking process; as long as the proof chain is complete, the system behavior achieves absolute compliance.

(3) Dual-Universe System and Proposition Shrinking (Prop-Shrinking)
To solve the computational overhead problem posed by formal proofs, the Core Layer designs multi-level universes ($\mathcal{U}_i$) for complex modeling and introduces a dedicated proof space (Prop). Based on the proof irrelevance (Proof Irrelevance) principle, the system contracts proof details through type erasure (Erasure) techniques after completing rigorous verification. This decision achieves a balance between logical depth and engineering efficiency, supporting efficient processing of large-scale real-time knowledge streams.

(4) Paradigm Shift from Denotational Semantics to Operational Semantics
Traditional logic emphasizes static truth values, whereas the KOS-TL Core Layer defines small-step reduction (Small-step reduction) rules for logical terms, defining knowledge operations as a reduction process. This decision ensures that the transformation from the Core Layer to the Kernel Layer is lossless and deterministic, achieving high isomorphism between logical inference steps and physical computation steps.

The comparison between the Core Layer's decisions and traditional knowledge base architectures is shown in Table 2.

Table 2: Comparison of Traditional Architecture and KOS-TL Core Layer Decisions

Design Dimension	**Traditional Architecture Decision**	**KOS-TL Core Layer Decision**
Knowledge Carrier	Database Records + External Validation	Logical Terms (Terms) with Dependent Proofs
Constraint Trigger	Runtime Interception / Business Code if-else	Type Checking
Evolution Driver	Database Transactions	Logical Reduction
Trust Root	System Administrator Permissions / Log Records	Immutable Mathematical Proof Chain

2.2 L1: Kernel (Kernel Layer) — Operational Semantics Domain

In the layered architecture of the KOS-TL system, the Kernel Layer undertakes the central function of transforming the static truths defined by the Core Layer into dynamic evolutionary drivers. If the Core Layer is likened to the constitution, then the Kernel Layer serves as the administrative hub" and power engine. Its core objective is to ensure, through formal reduction mechanisms, that the knowledge system maintains logical determinism and evolutionary continuity when handling high-frequency business streams and time streams.

2.2.1 Kernel Layer Requirements Modeling: Dynamic Evolution and State Determinism

For complex knowledge operating systems, the Kernel Layer's design primarily addresses the issues of collapse" and "generation" of knowledge states. First is the causal traceability requirement: the system mandates that every change in knowledge state must have explicit causal associations, meaning all changes must trace back to specific elaboration events (Event). Second is the state consistency requirement: especially in distributed or concurrent environments, atomicity of state transitions must be guaranteed, eradicating conflicting states at the logical level. Finally, real-time performance and progress guarantees (Progress Guarantee): the system must ensure that, upon receiving valid inputs, it can logically deterministically evolve to the next stable state, avoiding undefined behaviors or logical deadlocks caused by non-determinism.

2.2.2 Design Methodology: Small-Step Operational Semantics and State Machine Model

The Kernel Layer rejects traditional black-box batch processing and is instead built upon formalized small-step operational semantics (Small-step Operational Semantics). The core of its methodology lies in the state triple model. The Kernel abstracts the system as $\sigma = \langle \mathcal{K}, \mathcal{TS}, \mathcal{P} \rangle$, where $\mathcal{K}$ represents the current verified set of knowledge truths, $\mathcal{TS}$ is the coupling of logical clocks and physical anchors, and $\mathcal{P}$ is the queue of elaborated but unexecuted events. The Kernel Layer is essentially event-centric, based on the ontological status of events **?**, where events are defined as state transition operators (Events as Transitions): each type of event corresponds to an explicit transition operator that, by synchronously invoking the Core Layer's judgment results, drives the monotonic evolution from the old state σ to the new state σ'.

2.2.3 Overall Architecture Description

As the "logical router" between the logical foundation (Core) and the physical environment (Runtime), the Kernel Layer collaborates through three core modules:

- Event Queue Manager: Responsible for receiving event packets $\langle e, p \rangle$ elaborated from the Runtime Layer, and performing strict sequencing (Sequencing) and dependency conflict detection.
- Evolution Scheduler: Drives the system's core evolution loop. This module employs a Peek-Verify-Reduce-Confirm" process, i.e., invoking the Core Layer to verify preconditions before consuming events, executing reduction operations, and validating postconditions. Its operating objects are always logical terms (Terms) rather than underlying physical bits.
- State Mirror: Maintains the latest truth view" at the logical level, providing consistent context for state queries from the Runtime Layer and contextual verification from the Core Layer.

2.2.4 Key Design Decisions

(1) Decoupling of Strong Sequential Commit and Asynchronous Elaboration

To address the contradiction between the high-frequency generation of physical signals and the time-consuming nature of logical verification (deep proofs), the Kernel Layer adopts a decision of asynchronous elaboration, sequential commit." The Runtime Layer can execute signal elaboration in parallel, but the Kernel Layer insists on sequential commit (Sequential Commit). This decision ensures the uniqueness of the causal chain, presenting the knowledge evolution trajectory as a deterministic linear path, mitigating risks of complex logical branch backtracking.

(2) Closed-Loop Evolution: Precondition Verification and Postcondition Evidence Synthesis

To ensure that the system state after operations continues to conform to the ontology definition, the Kernel Layer establishes a closed-loop mechanism of pre-verification, post-synthesis." Before executing an operation, the Kernel forcibly verifies the $Pre(e)$ provided by the Core Layer; after the operation completes, it automatically triggers the Core Layer to synthesize a new proof for $Post(e)$. This decision guarantees that the system always transitions between states proven to be true," eliminating logical vacuum periods.

(3) Loose Coupling Mechanism Between Logical Time and Physical Time

To tackle the pain point of physical time precedence inconsistencies with causal logic in distributed networks, the Kernel Layer maintains an independent logical order. Physical timestamps serve only as evidence anchors, with synchronization occurring only during the materialization phase (Runtime). This decision leverages time evidence in logical proofs for reordering, fundamentally solving the phantom timing" problem and ensuring absolute fidelity of the **Causality Order**.

(4) Deterministic Reduction

The Kernel Layer strictly prohibits non-deterministic (Non-deterministic) choices. If an event points to multiple possible evolution branches during reduction, it will be deemed a type error in the Core Layer. This decision greatly enhances the system's predictability: under the same initial state and event sequence, the Kernel Layer will produce a mathematically unique, reproducible knowledge view.

2.3 L2: Runtime Layer — System Evolution Domain

In the architecture of KOS-TL (ZhiXing Logic), the Runtime Layer is defined as the key anchor point between the logical world and the non-deterministic physical world. If the Core Layer is regarded as the system's constitution and the Kernel Layer as the administrative hub," then the Runtime Layer serves as the system's senses, limbs, and physical carrier.” Its core mission is to solve the “last mile” problem of formal logic implementation in complex engineering environments, achieving seamless integration between logical semantics and underlying physical signals.

2.3.1 Requirements Modeling: Physical Fidelity and Environmental Elaboration

The design of the Runtime Layer aims to address the fundamental tension between the physical world and logical abstractions. First is the signal elevation requirement: raw bit streams (Raw Bits) generated by the physical world lack intrinsic semantics, and the Runtime Layer needs to elevate” them into event objects with logical connotations. Second is the resource mapping requirement: abstract state updates output from the logical layer must ultimately be precisely implemented in disk bits, memory entries, or hardware controller voltage states. Finally, real-time and concurrency requirements: when handling high-frequency sensor signals, the system must ensure low-latency ingestion without disrupting the causal order consistency maintained by the Kernel Layer.

2.3.2 Design Methodology: Elaboration and Materialization in Bidirectional Mapping

The methodological core of the Runtime Layer lies in establishing a bidirectional elaboration relationship between logical semantics and physical resources.

(1) Elaboration (Refinement/Elaboration) Methodology

The system introduces the elab operator to achieve signal elevation. Unlike traditional syntactic parsing (Parsing), elab is a proof construction process: it references the Core Layer’s ontology templates to find logical evidence p for the raw signal s, thereby converting it into a proven event $\langle e, p\rangle$ that conforms to the Kernel interface. This process ensures that all inputs entering the system have a formalized basis for legitimacy.

(2) Materialization Methodology

Through the $\mathcal{M}$ operator, the Runtime Layer is responsible for degrading the abstract knowledge state $\mathcal{K}$ in the Kernel Layer into concrete physical forms. For example, materializing a logical transfer successful assertion into an ACID transaction in a database or a transaction entry on a blockchain. The materialization mechanism ensures the faithful execution of logical conclusions in the physical world.

2.3.3 Overall Architecture Description

As the hub responsible for cross-border interaction,the Runtime Layer consists of the following key components:

- Elaboration Engine (Elaborator): Connects to physical I/O devices, responsible for monitoring external interrupts and sensor data streams, and performing bidirectional elaboration to anchor non-deterministic signals into deterministic logical events.
- Physical Storage Manager (Physical Storage Manager): Abstracts underlying media differences, manages databases and memory mappings, ensuring the atomicity and persistence of materialization operations.
- Scheduler Relay (Scheduler Relay): Acts as a buffer for the Kernel’s sequential evolution, responsible for concurrent ingestion of multi-threaded signals and ordered queuing.

2.3.4 Key Design Decisions

(1) Input Filtering Decision Based on Elaboration

To address the problem of traditional systems directly reading variables being susceptible to environmental interference (dirty data), the Runtime Layer mandates that all inputs must pass through the elaboperator. If a signal cannot construct a valid proof p in the Core Layer, it is deemed invalid input and immediately discarded. This decision establishes a logical firewall, ensuring the Kernel Layer is not polluted by unexpected signals.

(2) Atomic Commit Fence

To solve the knowledge-action inconsistency problem (i.e., logical update succeeds but physical write fails), the Runtime Layer introduces a fence mechanism similar to two-phase commit. Only after the physical layer returns a write acknowledgment (ACK) does the logical clock $\mathcal{TS}$ formally advance. This decision achieves precise synchronization between physical storage and logical truths.

(3) Resource Abstraction and Multi-Backend Plug-and-Play Support To adapt to diverse environments from embedded devices to cloud clusters, the Runtime Layer designs the materialization operator $\mathcal{M}$ as a pluggable backend. This decision realizes logical portability: the logical axioms of the Core Layer and Kernel Layer remain unchanged, and cross-platform semantic consistency can be achieved by simply replacing the Runtime Layer backend.

(4) Deterministic Trajectory Replay and Post-Disaster Self-Healing
Based on the unreliability of the physical environment, the Runtime Layer fully records the original trajectories of all elaboration events. Leveraging the deterministic reduction characteristics of the Kernel Layer, the system can, after a failure, replay the event stream to logically re-derive and repair the physical configuration. This decision provides the system with extremely strong logical robustness and self-healing capabilities.

The hierarchical structure and stability guarantees of the entire KOS-TL system are shown in Table 3.

Table 3: System Hierarchical Structure and Stability Guarantees

Layer	Core Focus	Core Operators	Stability Guarantee
L0: Core	Static Structure of Knowledge	$\Pi, \Sigma, \mathsf{Prop}$	Type Checking (Type Checking)
L1: Kernel	Event-Based State Transitions	$\mathsf{STEP}, \mathsf{Ev}$	Proof Verification (Proof Verification)
L2: Runtime	Mapping Between Reality and Logic	$\mathsf{elaborate}, \mathcal{M}$	Transactional Consistency (Transactional Consistency)

3 KOS-TL Core Layer: Static Logical Foundation

The Core Layer is the "formal constitution" of the KOS-TL system, based on Intuitive Dependent Type Theory . Its core task is to define the static structure of knowledge, logical constraints, and their validity proofs, providing an unalterable logical foundation for upper-layer execution. It does not change with time and is only responsible for defining what constitutes a "legal construction".

3.1 Syntax

3.1.1 Domain ($\mathcal{D}_{Core}$)

The domain ($\mathcal{D}_{Core}$) world of the Core Layer consists of a dual-axis structure: one is the data axis, and the other is the logical axis.

- **Dual-Axis Universes**:
  - **Computational Axis ($\mathcal{U}_i$)**: Follows predicativity. $\mathcal{U}_0$ contains base sorts (Sorts), and $\mathcal{U}_{i+1}$ contains $\mathcal{U}_i$ as an element. Used for modeling data with physical effects.
  - **Logical Axis (Type_i)**: Follows predicativity, but its base $\mathsf{Prop} : \mathsf{Type}_1$ has impredicativity. Type_i is used for modeling logical predicate spaces and metalogical rules.
  - **Hierarchical Relations**:
    - $\mathsf{Prop} : \mathsf{Type}_1, \quad \mathsf{Type}_i : \mathsf{Type}_{i+1}$
    - $\mathcal{U}_i : \mathcal{U}_{i+1}, \quad \mathcal{U}_i : \mathsf{Type}_{i+1}$ (Computational universes can be objects of logical discussion)
    - $\mathsf{Prop} \hookrightarrow \mathcal{U}_1$ (Propositions can be embedded into the data axis)
- **Base Sorts** : Val (atomic values), Time (time point scalars), ID (unique identifiers).
- **Knowledge Objects** : Instances of all dependent record types.

3.1.2 Syntax

- **Type Constructions (Types)**:

$$\begin{aligned} A, B ::= {} & \mathsf{Prop} \mid \mathsf{Type}_i \mid \mathcal{U}_i && \text{(Universes)} \\ & \mid \mathsf{Val} \mid \mathsf{Time} \mid \mathsf{ID} && \text{(Base Sorts)} \\ & \mid \Pi(x : A).B \mid \Sigma(x : A).B \mid A + B \mid \mathsf{Id}_A(a, b) && \text{(Constructors)} \end{aligned}$$

- **Term Constructions (Terms)**:

$$\begin{aligned} t, u ::= {} & x \mid \lambda x.t \mid t\, u && (\Pi \text{ Intro/Elim}) \\ & \mid \langle t, u \rangle \mid \mathsf{split}(t, x.y.u) && (\Sigma \text{ Intro/Elim}) \\ & \mid \mathsf{inl}(t) \mid \mathsf{inr}(t) \mid \mathsf{case}(t, x.u, y.v) && (+ \text{ Intro/Elim}) \\ & \mid \mathsf{refl} && (\text{Id Intro}) \end{aligned}$$

- **Judgments**:
  - $\Gamma \vdash A : \mathcal{S}$: Indicates that A is a valid type, where $\mathcal{S} \in \{\mathsf{Type}_i, \mathcal{U}_i\}$.
  - $\Gamma \vdash t : A$: Indicates that t is a valid instance of type A (for the data axis) or a valid proof (for the logical axis).
  - $\Gamma \vdash A \equiv B$ and $\Gamma \vdash t \equiv u$: Indicates that types or terms are computationally equivalent (conversion rules).

To maintain the simplicity of the Core Layer, logical operations in Prop are implemented through type construction operators in $\mathcal{U}$. The specific semantic mapping relations are shown in Table 4.

Regarding the special nature of Prop. The KOS-TL Core follows the **Proof Irrelevance** principle: for any $P : \mathsf{Prop}$, if $p, q : P$, then semantically $[\![p]\!] = [\![q]\!]$.

Table 4: Isomorphism Between Logical Propositions and Types

Logical Proposition (Prop)	**Type Construction ($\mathcal{T}$)**	**Term Construction (Terms)**
Universal Quantifier $\forall x : A.\, P(x)$	Dependent Product $\Pi(x : A).P$	$\lambda x.\, p$
Existential Quantifier $\exists x : A.\, P(x)$	Dependent Sum $\Sigma(x : A).P$	$\langle a, p \rangle$
Logical Implication $P \to Q$	Function Space $P \to Q$	$\lambda p.\, q$
Logical Conjunction $P \wedge Q$	Product Type $P \times Q$	$\langle p, q \rangle$
Logical Disjunction $P \vee Q$	Sum Type $P + Q$	$\mathsf{inl}(p)/\mathsf{inr}(q)$

According to the construction rules for terms and types, the type set $\mathcal{T}$ of the KOS-TL Core Layer is inductively defined by the following rules.

(1) Base Rules

$$\mathsf{Prop} : \mathsf{Type}_1 \quad \text{(Logical axis starting point)}$$
$$\mathsf{Type}_i : \mathsf{Type}_{i+1} \quad \text{(Logical universe accumulation)}$$
$$\mathcal{U}_i : \mathcal{U}_{i+1} \quad \text{(Data universe accumulation)}$$
$$\mathsf{Prop} \hookrightarrow \mathcal{U}_1 \quad \text{(Lifting rule: Propositions can be treated as data processing, which is an implicit coercion)}$$

$\mathsf{Prop} \hookrightarrow \mathcal{U}_1$ is a one-way embedding, allowing proofs to be embedded as objects into data records (such as Σ types), but ordinary data in $\mathcal{U}_i$ cannot be directly used as propositions for logical derivation. Atomic types: $\mathsf{Val} \in \mathcal{T}, \mathsf{Time} \in \mathcal{T}, \mathsf{ID} \in \mathcal{T}$.

(2) Dependent Product Construction

- Logical/Computational Hybrid Rule:

$$\frac{\Gamma \vdash A : \mathsf{Type}_i/\mathcal{U}_i \quad \Gamma, x : A \vdash B : \mathsf{Prop}}{\Gamma \vdash \Pi(x : A).B : \mathsf{Prop}}\text{(Impredicative)}$$

- Pure Universe Rule:

$$\frac{\Gamma \vdash A : \mathsf{Type}_i \quad \Gamma, x : A \vdash B : \mathsf{Type}_j}{\Gamma \vdash \Pi(x : A).B : \mathsf{Type}_{\max(i,j)}}\text{(Predicative)}$$

Prop has a special property called impredicativity (Impredicativity). Regardless of how high the level of A is, as long as B belongs to Prop, then $\Pi(x : A).B$ usually still belongs to Prop. Meaning: This allows us to perform logical judgments on "infinite objects." For example, "for all types in $\mathcal{U}_1$, they all satisfy the safety property P," this judgment itself is still just a simple Prop (true or false), without exploding into a super-complex type.

(3) Dependent Sum Construction

To prevent logical paradoxes (similar to Girard's Paradox), Σ types in KOS-TL must be predicative. If A is a type, and under the assumption of variable $x : A$, B is a type, then:

$$\frac{\Gamma \vdash A : \mathcal{U}_i \quad \Gamma, x : A \vdash B : \mathcal{U}_j}{\Gamma \vdash \Sigma(x : A).B : \mathcal{U}_{\max(i,j)}}$$

Note: If $A, B \in \mathcal{U}$, the result is in $\mathcal{U}$; if proof extraction is involved, the highest level is constrained by the logical axis Universe. The dependent sum construction models knowledge objects. The Σ type is the core of KOS-TL, forcing data x to be associated with a proof term $p : B(x)$.

(4) Sum Type Construction
If A and B are valid types respectively, then their disjunctive sum (disjoint union) is also a type:

$$\frac{\Gamma \vdash A : \mathcal{U} \quad \Gamma \vdash B : \mathcal{U}}{\Gamma \vdash A + B : \mathcal{U}}$$

The sum type models the "disjunction ($\vee$)" relation in logic. In manufacturing scenarios, it is used to model "mutually exclusive states" or "alternative paths." For example, a task's state is either $\mathsf{Success}$ or $\mathsf{Failure}$.

(5) Identity Type Construction
If A is a type, and u, v are two terms of type A, then:

$$\frac{\Gamma \vdash A : \mathcal{U} \quad \Gamma \vdash u : A \quad \Gamma \vdash v : A}{\Gamma \vdash \mathsf{Id}_A(u, v) : \mathsf{Prop}}$$

The identity type construction models the equivalence of knowledge, serving as the logical foundation for judging whether two facts are consistent during "causal tracing" and "state rollback".

In the KOS-TL Core, Prop is a special domain dedicated to handling logical assertions. Unlike ordinary $\mathcal{U}_i$, it exhibits impredicativity under the Π construction.

Definition 1. *Impredicative Π Construction Rule*

For any level type $A : \mathcal{U}_i$, if under the assumption $x : A$, B is a proposition, then its universal quantifier (or function space) still maps back to the smallest proposition world:

$$\frac{\Gamma \vdash A : \mathcal{U}_i \quad \Gamma, x : A \vdash B : \mathit{Prop}}{\Gamma \vdash \Pi(x : A).B : \mathit{Prop}}$$

Logical Closure Point*: This means that the complexity of propositions does not increase in level with the expansion of their quantifier scope. This property allows us to make consistency assertions over full data (even objects at the $\mathcal{U}_k$ level) without triggering Universe explosion.*

Definition 2. *Universe Lifting and Inclusion Rules*

To support the closure of dual-axis semantics, the system introduces the following implicit conversions:

- *Observation from Computation to Logic:*

$$\frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash A : \mathit{Type}_{i+1}}$$

This means any computational type can be treated as an object of discussion for logical propositions (e.g., discussing the algebraic properties of SensorData at the Type level).

- *Computational Embedding of Propositions:*

$$\frac{\Gamma \vdash P : \mathit{Prop}}{\Gamma \vdash P : \mathcal{U}_1}$$

This allows logical proof terms $p : P$ to be packaged into Σ records as inputs to real-time computational systems (i.e., "data packets with proofs").

3.1.3 Judgmental Rules

To ensure the above constructions are logically well-formed, the KOS-TL Core follows the following derivation rules.

(1) Dependent Product (Π-Types)

- Introduction Rules
For Π types, the construction (introduction) rule is:

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : \Pi(x : A).B}$$

In the current context Γ, if we assume a variable x of type A and can construct a term t of type B, then we can construct a λ abstraction (i.e., function) whose type is $\Pi(x : A).B$.

- Elimination Rules
  For Π types, there is a general rule f (of type $\Pi(x : A).B$) and a concrete object a (of type A). Applying f to a (denoted $f\,a$) yields a result of type $B[a/x]$. In the result type, all x are replaced by the concrete value a.

$$\frac{\Gamma \vdash f : \Pi(x : A).B \quad \Gamma \vdash a : A}{\Gamma \vdash f\,a : B[a/x]}$$

(2) Dependent Sum (Σ-Types)

- Introduction Rules
  For Σ types, the construction (introduction) rule is:

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B[a/x]}{\Gamma \vdash \langle a, b\rangle : \Sigma(x : A).B}$$

  The introduction rule embodies "construction as proof": only when you can provide evidence b satisfying $B(a)$ can the knowledge object be created.
- Elimination Rules
  For Σ types, we define a general dependent elimination operator split. It allows constructing a target term dependent on the overall pair by pattern matching on the pair structure.

$$\frac{\Gamma \vdash p : \Sigma(x : A).B \quad \Gamma, x : A, y : B \vdash t : C[\langle x, y\rangle/z]}{\Gamma \vdash \mathsf{split}(p, x.y.t) : C[p/z]}$$

  Traditional projection operators can be defined as special cases of split:

$$\mathsf{proj}_1(p) \text{ (left projection)} \equiv \mathsf{split}(p, x.y.x)$$
$$\mathsf{proj}_2(p) \text{ (right projection)} \equiv \mathsf{split}(p, x.y.y)$$

(3) Sum Types ($A + B$)

- Introduction Rules
  The introduction rules define how to create an object of type $A + B$. It has two branches, corresponding to "left choice" and "right choice".

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{inl}(a) : A + B} \qquad \frac{\Gamma \vdash b : B}{\Gamma \vdash \mathsf{inr}(b) : A + B}$$

  If there is evidence a of type A, it can be wrapped into type $A + B$ via the label inl (In-Left). Similarly, if there is evidence b of type B, it can be wrapped into $A + B$ via inr (In-Right).
- Elimination Rules
  The elimination rule defines how to safely use an object of type $A + B$. Since it is unknown whether the interior is A or B, two schemes must be prepared.

$$\frac{\Gamma \vdash s : A + B \quad \Gamma, x : A \vdash t : C \quad \Gamma, y : B \vdash u : C}{\Gamma \vdash \mathsf{case}(s, x.t, y.u) : C}$$

  $s : A + B$ is the input. $\Gamma, x : A \vdash t : C$ is scheme one. If s is ultimately proven to be of type A, extract the data inside and give it to x, then compute a result according to logic t, with result type C. $\Gamma, y : B \vdash u : C$ is scheme two. If s is of type B, give the data to y, and compute a result of type C according to logic u. $\mathsf{case}(s, x.t, y.u) : C$ indicates that regardless of which path s takes, a deterministic result of type C can ultimately be obtained.

(4) Conversion Rule

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash A \equiv B \quad \Gamma \vdash B : \mathcal{S}}{\Gamma \vdash t : B}$$

This rule ensures that if term t is valid in $\mathcal{U}_1$, and $\mathcal{U}_1 \hookrightarrow \mathsf{Type}_2$ holds, then t automatically has validity for observation at higher levels.

3.1.4 Reduction Rules

We use $\to$ to denote one-step reduction (One-step reduction) and $\twoheadrightarrow$ to denote multi-step reduction (computational closure).

(1) Function Reduction (β-reduction)
For Π type constructions (λ abstraction):

$$\frac{}{\Gamma \vdash (\lambda x : A.t)\, u \to t[u/x]}$$

(2) Dependent Record Reduction (ι-reduction)
For structured elimination of Σ types. This is the core correction point, directly deconstructing pairs via the split operator:

$$\frac{}{\Gamma \vdash \mathsf{split}(\langle u, v\rangle, x.y.t) \to t[u/x, v/y]}$$

Under this definition, traditional projection reductions can be naturally derived as special cases:

- Left Projection:
$$\mathsf{proj}_1(\langle u, v\rangle) \equiv \mathsf{split}(\langle u, v\rangle, x.y.x) \to u$$
- Right Projection:
$$\mathsf{proj}_2(\langle u, v\rangle) \equiv \mathsf{split}(\langle u, v\rangle, x.y.y) \to v$$

(3) Sum Type Reduction (ι-reduction)
For branch judgment of $+$ types. Matching labels via the case operator:

$$\frac{}{\Gamma \vdash \mathsf{case}(\mathsf{inl}(u), x.t, y.v) \to t[u/x]} \qquad \frac{}{\Gamma \vdash \mathsf{case}(\mathsf{inr}(w), x.t, y.v) \to v[w/y]}$$

(4) Identity Term Reduction (ι-reduction)
When the judgment term has been reduced to refl, the identity judgment automatically resolves.

To support modular definitions and local variables in engineering practice, the system defines the following auxiliary conversion rules. Unlike core reductions (β, ι), these conversions are typically triggered on demand during the type checker's **equivalence judgment (Conversion Check)** phase and are not counted in core logical steps.

- Global Unfolding (δ-conversion):
If term c is defined as $c := t : A$ in context Γ, then unfolding is allowed during equivalence judgment:

$$\frac{(c := t : A) \in \Gamma}{\Gamma \vdash c \equiv_\delta t}$$

Meaning: Allows the system to recognize that aliases (Alias) and their original definitions are logically the same object.

- Local Binding Unfolding (ζ-conversion):
For local definitions of the `let` structure, its semantics is equivalent to immediate substitution:

$$\Gamma \vdash (\mathsf{let}\ x = u\ \mathsf{in}\ t) \equiv_\zeta t[u/x]$$

Meaning: Supports local reuse of terms without increasing the overhead of function calls (β).

- Extensional Equivalence (η-conversion):
To ensure function consistency, the system supports functional extensionality judgment:

$$\Gamma \vdash \lambda x : A.(f\, x) \equiv_\eta f \quad (x \notin \mathrm{FV}(f))$$

Meaning: Ensures behavioral consistency between function abstractions and direct references, supporting functional programming paradigms.

δ and ζ ensure that the system has definitional transparency (Definitional Transparency), meaning that referencing names does not change the semantic essence of logical terms.

Definition 3. *Definitional Equality*

The judgmental equivalence relation $\equiv$ in KOS-TL is the smallest equivalence relation generated by all the above reductions (β, ι) and conversions (δ, ζ, η) (satisfying symmetry, transitivity, and congruence).

3.2 Logical Properties of the Core Layer

Definition 4. *Normal Form*

*A term t is said to be in **normal form** (denoted $t \in \textsf{NF}$), if and only if it contains no **Redex** (reducible expressions). That is, for the reduction relation $\to$ defined in KOS-TL Core, there does not exist a term t' such that:*

$$t \to t'$$

Definition 5. *Strong Normalization ($\textsf{SN}$)*

*A term t is **strongly normalizing** if and only if there is no infinite reduction sequence starting from t. That is, all possible reduction paths $t \to t_1 \to t_2 \ldots$ terminate in a finite number of steps at some normal form.*

Definition 6. *Reducibility Candidate Set ($\textsf{Red}_A$)*

For any type A and term $t : A$, the reducibility candidate set $\textsf{Red}_A \subseteq \textsf{Val}_A$ is defined inductively on the structure of A as follows:

(1) Universe Type Cases

- $A \equiv \mathcal{U}_i$

$$t \in \textsf{Red}_{\mathcal{U}_i} \iff t \in \mathcal{RC} \land \textsf{level}(t) < i$$

where $\mathcal{RC}$ is the set of all terms satisfying the CR properties (SN, stability, neutral term construction).

- $A \equiv \textsf{Type}_i$

$$t \in \textsf{Red}_{\textsf{Type}_i} \iff t \in \mathcal{RC} \land \textsf{level}(t) < i$$

- ***Cross-Axis Constraints**:*
Since $\textsf{Prop} : \textsf{Type}_1$, then $\textsf{Prop} \in \textsf{Red}_{\textsf{Type}_1}$.
Since $\textsf{Val} : \mathcal{U}_0$, then $\textsf{Val} \in \textsf{Red}_{\mathcal{U}_1}$.

(2) Base Type Cases

- $A \equiv \textsf{Val}$

$$t \in \textsf{Red}_A \iff t \in \textsf{SN} \land \exists c \in \textsf{Const}_{\mathbb{N}}.\, t \to^* c$$

- $A \equiv \textsf{Time}$
$t \in \textsf{Red}_A \iff t \in \textsf{SN} \land t$ *represents a valid timestamp or duration*
(i.e., $t \to^* \mathsf{timestamp}(n)$ or $t \to^* \mathsf{duration}(n)$ for $n \in \mathbb{N}$)

(3) Constructed Type Cases

- $A \equiv \Pi(x : B).C$

$$t \in \textsf{Red}_A \iff \forall u \in \textsf{Red}_B.\, (t\ u) \in \textsf{Red}_{C[u/x]}$$

- $A \equiv \Sigma(x : B).C$

$$t \in \textsf{Red}_{\Sigma(x:B).C} \iff t \twoheadrightarrow \langle u, v\rangle \land u \in \textsf{Red}_B \land v \in \textsf{Red}_{C[u/x]}$$

$$t \in \textsf{Red}_A \iff t \in \textsf{SN} \land (t \to^* \mathsf{inl}(u) \implies u \in \textsf{Red}_B) \land (t \to^* \mathsf{inr}(v) \implies v \in \textsf{Red}_D)$$

- $A \equiv \textsf{Id}_B(a, b)$

$$t \in \textsf{Red}_A \iff t \in \textsf{SN} \land (t \to^* \mathsf{refl}(w) \implies w \in \textsf{Red}_B \land a \equiv_B w \land b \equiv_B w)$$

where:

- *$\textsf{Const}_{\mathbb{N}}$ is the set of natural number constants (numeric constants).*
- *$\to^*$ denotes the multi-step closure of $\beta\eta$-reduction.*
- *$\equiv_B$ denotes reducibility equivalence under type B: $p \equiv_B q \iff \exists v \in \textsf{Red}_B.\, (p \to^* v \land q \to^* v)$.*

- *For Time, the precise semantics of "valid timestamp or duration" must be predefined (e.g.,* $\mathsf{timestamp}(n)$ *represents Unix timestamp n,* $\mathsf{duration}(n)$ *represents n milliseconds duration) to ensure formalization.*

It should be noted that the interpretation of Prop does not depend on its universe level.

Definition 7. *Neutral Term In the KOS-TL Core Layer, a term t is a* ***neutral term*** *if and only if it satisfies one of the following two conditions:*

(1) t is a variable x.

(2) The head of t (Head) is a variable, and the term is in the process of being eliminated (Elimination) but cannot be further reduced.

The reducibility set Red_A of type A must satisfy three key saturation properties (Saturation Properties), commonly known as $CR1, CR2, CR3$.

Definition 8. *Saturation Properties of Reducibility Candidate Set Red_A*

(1) ***CR 1 (Inclusivity)***

If $t \in \mathsf{Red}_A$, then $t \in \mathsf{SN}$ (i.e., t must first be strongly normalizing).

(2) ***CR 2 (Stability)***

If $t \in \mathsf{Red}_A$ and $t \to t'$, then $t' \in \mathsf{Red}_A$.

(3) ***CR 3 (Neutral Term Construction)***

If t is a neutral term, and all one-step reduction terms t' of t are in Red_A, then $t \in \mathsf{Red}_A$. All variables are neutral terms.

Lemma 1. *Substitution Lemma*

If $\Gamma, x : B, \Delta \vdash t : A$ and $\Gamma \vdash u : B$, then $\Gamma, \Delta[u/x] \vdash t[u/x] : A[u/x]$. Here, Δ is a general context to handle variables defined after x that depend on x. In simple cases, Δ is empty.

Proof. Prove by induction on the structure of the type derivation tree. Discuss the following core cases based on the construction rule of t:

(1) **Variable Case (Variable)**
Assume t is a variable y.

- **Subcase 1:** $y = x$
  According to the derivation rule, $A = B$ at this point.
  We need to prove $\Gamma, \Delta[u/x] \vdash x[u/x] : B[u/x]$.
  Since $x[u/x] = u$, and the premise is $\Gamma \vdash u : B$. According to the context weakening rule (Weakening), $\Delta[u/x]$ can be added after Γ, so the conclusion holds.
- **Subcase 2:** $y \neq x$
  At this point, y must be defined in Γ or Δ.
  If $y \in \Gamma$, then $y[u/x] = y$ and $A[u/x] = A$ (since types in Γ do not depend on x), the conclusion is obvious.
  If $y \in \Delta$, then $y[u/x] = y$, its type is $A[u/x]$, which is exactly the corresponding declaration in $\Delta[u/x]$.

(2) **Π-Type Introduction (λ-Abstraction)**
Assume $t = \lambda y : C.M$, and $A = \Pi(y : C).D$.
The last derivation step is

$$\frac{\Gamma, x : B, \Delta, y : C \vdash M : D}{\Gamma, x : B, \Delta \vdash \lambda y : C.M : \Pi(y : C).D}$$

- Apply the induction hypothesis: Use the induction hypothesis on M (current context is $\Delta, y : C$):

$$\Gamma, \Delta[u/x], y : C[u/x] \vdash M[u/x] : D[u/x]$$

- Construct the conclusion: Apply the Π-introduction rule:

$$\Gamma, \Delta[u/x] \vdash \lambda y : C[u/x].M[u/x] : \Pi(y : C[u/x]).D[u/x]$$

  This is equivalent to $(\lambda y : C.M)[u/x] : (\Pi(y : C).D)[u/x]$.

(3) **Π-Type Elimination (Application)**
Assume $t = (f\, v)$, where $\Gamma, x : B, \Delta \vdash f : \Pi(y : C).D$ and $\Gamma, x : B, \Delta \vdash v : C$.

- Induction Hypothesis 1: $\Gamma, \Delta[u/x] \vdash f[u/x] : (\Pi(y : C).D)[u/x]$.
- Induction Hypothesis 2: $\Gamma, \Delta[u/x] \vdash v[u/x] : C[u/x]$.
- Combination: Apply the elimination rule:

$$(f[u/x]\, v[u/x]) : D[u/x][v[u/x]/y]$$

According to the commutativity of substitution, the above type is equivalent to $D[v/y][u/x]$, i.e., $A[u/x]$.

(4) **Σ-Type Construction (Pairing)** Assume $t = \langle t_1, t_2 \rangle$, and $A = \Sigma(y : C).D$.

- By the induction hypothesis, $t_1[u/x] : C[u/x]$.
- By the induction hypothesis, $t_2[u/x] : D[t_1/y][u/x]$.
- According to the rule, construct $\langle t_1[u/x], t_2[u/x] \rangle : (\Sigma(y : C).D)[u/x]$.

(5) **Σ-Type Elimination (Structured Elimination)** Assume $t = \mathsf{split}(p, x.y.u)$, and the last derivation step is:

$$\frac{\Gamma, z : B, \Delta \vdash p : \Sigma(x : A).B \quad \Gamma, z : B, \Delta, x : A, y : B \vdash u : C(\langle x, y \rangle)}{\Gamma, z : B, \Delta \vdash \mathsf{split}(p, x.y.u) : C(p)}$$

(Here $z : B$ is the variable we are substituting)

- **Induction Hypothesis 1**: Apply the induction hypothesis to p, yielding $\Gamma, \Delta[v/z] \vdash p[v/z] : (\Sigma(x : A).B)[v/z]$.
- **Induction Hypothesis 2**: Apply the induction hypothesis to the elimination body u (now with added context x, y):
$$\Gamma, \Delta[v/z], x : A[v/z], y : B[v/z] \vdash u[v/z] : C(\langle x, y \rangle)[v/z]$$
- **Construct the Conclusion**: Reapply the Σ-elimination rule (split rule):

$$\Gamma, \Delta[v/z] \vdash \mathsf{split}(p[v/z], x.y.u[v/z]) : C(p)[v/z]$$

Since $\mathsf{split}(p, x.y.u)[v/z] = \mathsf{split}(p[v/z], x.y.u[v/z])$, the conclusion holds.

(6) **$+$-Type Introduction (Injection)**
Assume $t = \mathsf{inl}_D(s)$, and $A = C + D$ (inr case symmetric).

- **Premise**: Known that $\Gamma, x : B, \Delta \vdash s : C$ and $\Gamma, x : B, \Delta \vdash D : \mathcal{U}$.
- **Induction Hypothesis**: For s, $s[u/x] : C[u/x]$; for type D, $D[u/x] : \mathcal{U}$.
- **Construct the Conclusion**: Apply the $+$-introduction rule:

$$\Gamma, \Delta[u/x] \vdash \mathsf{inl}_{D[u/x]}(s[u/x]) : C[u/x] + D[u/x]$$

i.e., $(\mathsf{inl}_D(s))[u/x] : (C + D)[u/x]$.

(7) **$+$-Type Elimination (Branch Judgment)**
Assume $t = \mathsf{case}(s, y.t_1, z.t_2)$, and the last derivation step is:

$$\frac{\Gamma', s : C + D \quad \Gamma', y : C \vdash t_1 : A \quad \Gamma', z : D \vdash t_2 : A}{\Gamma' \vdash \mathsf{case}(s, y.t_1, z.t_2) : A}$$

(where Γ' is abbreviated as $\Gamma, x : B, \Delta$)

- **Induction Hypothesis 1**: For the judgment term s, $\Gamma, \Delta[u/x] \vdash s[u/x] : C[u/x] + D[u/x]$.
- **Induction Hypothesis 2**: For the left branch t_1 (now with added context $y : C$), $\Gamma, \Delta[u/x], y : C[u/x] \vdash t_1[u/x] : A[u/x]$.
- **Induction Hypothesis 3**: For the right branch t_2 (now with added context $z : D$), $\Gamma, \Delta[u/x], z : D[u/x] \vdash t_2[u/x] : A[u/x]$.
- **Combination**: Apply the $+$-elimination rule (case rule):

$$\Gamma, \Delta[u/x] \vdash \mathsf{case}(s[u/x], y.t_1[u/x], z.t_2[u/x]) : A[u/x]$$

The conclusion holds.

(8) **Identity Type (Identity)**
If $t = \mathsf{refl}_a$, then $A = \mathsf{Id}_C(a, a)$.

- By the induction hypothesis, $a[u/x] : C[u/x]$.
- Directly apply the construction rule to obtain $\mathsf{refl}_{a[u/x]} : \mathsf{Id}_{C[u/x]}(a[u/x], a[u/x])$, i.e., $A[u/x]$.

□

Lemma 2. *Fundamental Lemma of Reducibility*

Let $\Gamma = \{x_1 : A_1, \dots, x_n : A_n\}$ be a well-formed context. If $\Gamma \vdash t : C$, and there exists a reducible substitution $\gamma = [u_1/x_1, \dots, u_n/x_n]$ such that for all i, $u_i \in \textsf{Red}_{A_i}$.

Then the substituted term $t[\gamma]$ must satisfy:

$$t[\gamma] \in \textsf{Red}_C$$

Proof. Prove by induction on the structure of the type derivation tree.

(1) **Variable Rules (Variables)**

If the derivation is $\Gamma \vdash x_i : A_i$. According to the substitution definition, $x_i[\gamma] = u_i$. By the premise $u_i \in \mathsf{Red}_{A_i}$, the proposition obviously holds.

(2) **Π-Type Introduction (λ-Abstraction)** If $\Gamma \vdash \lambda x : A.M : \Pi(x : A).B$.
Need to prove: For any $u \in \mathsf{Red}_A$, $((\lambda x : A.M)[\gamma]\, u) \in \mathsf{Red}_{B[u/x]}$.

(a) This term β-reduces to $M[\gamma, u/x]$.
(b) Since $u \in \mathsf{Red}_A$ and $\gamma \in \mathsf{Red}_\Gamma$, then $(\gamma, u/x)$ is a reducible substitution under the context $(\Gamma, x : A)$.
(c) By the induction hypothesis, $M[\gamma, u/x] \in \mathsf{Red}_B$.
(d) Since the reducibility set is closed under reverse reduction, $(\lambda x : A.M)[\gamma] \in \mathsf{Red}_{\Pi(x:A).B}$.

(3) **Π-Type Elimination (Application)**

If $\Gamma \vdash (f\, v) : C$, where $C = B[v/x]$ and $\Gamma \vdash f : \Pi(x : A).B, \Gamma \vdash v : A$.

(a) By the induction hypothesis, $f[\gamma] \in \mathsf{Red}_{\Pi(x:A).B}$.
(b) By the induction hypothesis, $v[\gamma] \in \mathsf{Red}_A$.
(c) According to the definition of Red_Π: If a term belongs to the reducibility set of a Π type, then its application to any term in the reducibility set of the parameter type must belong to the reducibility set of the result type.
(d) Therefore, $(f[\gamma]\, v[\gamma]) \in \mathsf{Red}_{B[v[\gamma]/x]}$.
(e) Since $(f[\gamma]\, v[\gamma]) = (f\, v)[\gamma]$ and $B[v[\gamma]/x] = C[\gamma]$, the conclusion holds.

(4) **Σ-Type Introduction (Pairing)**

If $\Gamma \vdash \langle a, b \rangle : \Sigma(x : A).B$.

(a) By the induction hypothesis, $a[\gamma] \in \mathsf{Red}_A$.
(b) By the induction hypothesis, $b[\gamma] \in \mathsf{Red}_{B[a[\gamma]/x]}$.
(c) According to the definition of Red_Σ, if both components are reducible, then $\langle a, b \rangle[\gamma] \in \mathsf{Red}_{\Sigma(x:A).B}$.

(5) **Σ-Type Elimination (split Operator)**

If $\Gamma \vdash \mathsf{split}(p, x.y.t) : C$, where $\Gamma \vdash p : \Sigma(x : A).B$.

(a) **Induction Hypothesis**: By the induction hypothesis, $p[\gamma] \in \mathsf{Red}_{\Sigma(x:A).B}$. This means $p[\gamma]$ is strongly normalizing and ultimately reduces to some pair $\langle u, v \rangle$, where $u \in \mathsf{Red}_A, v \in \mathsf{Red}_{B[u/x]}$.
(b) **Reduction Analysis**: According to ι-reduction, $\mathsf{split}(p[\gamma], x.y.t[\gamma]) \twoheadrightarrow t[\gamma, u/x, v/y]$.
(c) **Apply Induction Hypothesis**: Since $(\gamma, u/x, v/y)$ is a valid reducible substitution under the context $(\Gamma, x : A, y : B)$, applying the induction hypothesis to t yields:

$$t[\gamma, u/x, v/y] \in \mathsf{Red}_C$$

(d) **Closure**: Using the closure of the reducibility set under reverse ι-reduction (a corollary of CR 3 property), the original term $\mathsf{split}(p, x.y.t)[\gamma]$ also belongs to Red_C.

(6) **$+$-Type Introduction (Injection)**

If $\Gamma \vdash \mathsf{inl}(a) : A + B$ (inr similarly).

(a) By the induction hypothesis, $a[\gamma] \in \mathsf{Red}_A$.
(b) According to the definition of Red_{A+B} (typically defined by neutral terms and injection properties): Since $a[\gamma]$ is reducible, its injection term $\mathsf{inl}(a[\gamma])$ is also reducible in the $+$-type reducibility system (using reverse reduction closure).
(c) Thus, $\mathsf{inl}(a)[\gamma] \in \mathsf{Red}_{A+B}$.

(7) **Sum Type Elimination (Case Analysis)**

If $\Gamma \vdash \mathsf{case}(t, x.M, y.N) : C$.

(a) By the induction hypothesis, $t[\gamma] \in \mathsf{Red}_{A+B}$.
(b) $t[\gamma]$ reduces to $\mathsf{inl}(u)$ or $\mathsf{inr}(v)$. Assume $\mathsf{inl}(u)$, then $u \in \mathsf{Red}_A$.
(c) At this point, the case term reduces to $M[\gamma, u/x]$.
(d) By the induction hypothesis, $M[\gamma, u/x] \in \mathsf{Red}_C$. Similarly for the inr case.

(8) **Identity Type Introduction (refl)**

If $\Gamma \vdash \mathsf{refl}_a : \mathsf{Id}_A(a, a)$.

(a) By the induction hypothesis, $a[\gamma] \in \mathsf{Red}_A$.
(b) Obviously $a[\gamma] \cong a[\gamma]$ and $\mathsf{refl} \in \mathsf{SN}$.
(c) Thus, $\mathsf{refl}_{a[\gamma]} \in \mathsf{Red}_{\mathsf{Id}_A(a[\gamma], a[\gamma])}$.

(9) **δ-Reduction and Local Definition (let)**

If $\Gamma \vdash \mathsf{let}\ x = u\ \mathsf{in}\ t : C$.

(a) By the induction hypothesis, $u[\gamma] \in \mathsf{Red}_A$.
(b) Construct the extended substitution $\gamma' = [\gamma, u[\gamma]/x]$. Since $u[\gamma]$ is reducible, γ' is a reducible substitution under the well-formed context.
(c) By the induction hypothesis, $t[\gamma'] \in \mathsf{Red}_C$.
(d) Since $(\mathsf{let}\ x = u\ \mathsf{in}\ t)[\gamma] \to t[\gamma, u[\gamma]/x]$, according to the closure of the reducibility set under reverse ζ-reduction, the conclusion holds.

□

Theorem 1. *Strong Normalization for KOS-TL Core*

Let Γ be a well-formed context. If term t satisfies $\Gamma \vdash t : A$, then t is strongly normalizing (i.e., t $\in \mathsf{SN}$). This means any reduction sequence starting from t, $t \to t_1 \to t_2 \ldots$, is finite.

Proof. The proof references the Tait-Girard method **?**. The core logic is to use the fundamental lemma to convert "type validity" to "reducibility", and then leverage the property that reducible terms are strongly normalizing.

Step 1: Introduce Identity Substitution

For the context $\Gamma = \{x_1 : A_1, \ldots, x_n : A_n\}$, we construct a special substitution γ_{id}:

$$\gamma_{id} = [x_1/x_1, \ldots, x_n/x_n]$$

To apply the fundamental lemma, we need to prove that γ_{id} is a reducible substitution. This means that for every variable x_i, it must be proven to belong to the reducibility set, i.e., $x_i \in \mathsf{Red}_{A_i}$.

Step 2: Reducibility of Variables

According to the properties of the reducibility candidate set (Girard's $\mathcal{RC}$), all Red_A sets satisfy the following two key properties:

1. CR 1 (SN Inclusivity): If $t \in \mathsf{Red}_A$, then $t \in \mathsf{SN}$.

2. CR 3 (Neutral Term Property): If t is a neutral term (i.e., a variable or application of a variable that cannot be further reduced) and all its one-step reduction terms are in Red_A, then $t \in \mathsf{Red}_A$.

Since the variable x_i is a basic neutral term and has no reduction forms, by CR 3, it follows that $x_i \in \mathsf{Red}_{A_i}$. Thus, γ_{id} satisfies the premise of the fundamental lemma.

Step 3: Application of Fundamental Lemma

Since the premise $\Gamma \vdash t : A$ holds, and γ_{id} is a reducible substitution, by the fundamental lemma 2:

$$t[\gamma_{id}] \in \mathsf{Red}_A$$

Since $t[\gamma_{id}]$ is syntactically equivalent to the term t itself, we obtain:

$$t \in \mathsf{Red}_A$$

Step 4: Conclusion Derivation

According to the property CR 1 of the reducibility set (all terms belonging to the reducibility set are strongly normalizing):

$$t \in \mathsf{Red}_A \implies t \in \mathsf{SN}$$

This completes the proof. □

Theorem 2. *Subject Reduction*

In KOS-TL Core, reduction operations do not change the type of a term. If $\Gamma \vdash t : A$ *and* $t \to t'$*, then* $\Gamma \vdash t' : A$.

Proof. Based on the substitution lemma 1, the proof proceeds by structural induction as follows.

(1) **Main Reduction Case: β-reduction**

Consider the most basic reduction step $(\lambda x : B.t)u \to t[u/x]$:

- From the premise $\Gamma \vdash (\lambda x : B.t)u : A$, there must exist a type B such that $\Gamma \vdash \lambda x : B.t : \Pi(x : B).A'$ and $\Gamma \vdash u : B$.
- According to the inversion of the Π-introduction rule, $\Gamma, x : B \vdash t : A'$.
- Applying the substitution lemma directly yields $\Gamma \vdash t[u/x] : A'[u/x]$.

(2) **Σ-Type Reduction: ι-reduction**

Consider the case $\mathsf{split}(\langle u, v\rangle, x.y.t) \to t[u/x, v/y]$:

- **Premise**: From $\Gamma \vdash \mathsf{split}(\langle u, v\rangle, x.y.t) : C$, it follows that:
  (a) $\Gamma \vdash \langle u, v\rangle : \Sigma(x : A).B$
  (b) $\Gamma, x : A, y : B \vdash t : C'$ (where C is actually $C'[\langle u, v\rangle/z]$)
- **Derivation**: From (a), by inversion of the introduction rule, $\Gamma \vdash u : A$ and $\Gamma \vdash v : B[u/x]$.
- **Application**: Continuously apply the substitution lemma twice to t (first substitute x, then y), directly obtaining:

$$\Gamma \vdash t[u/x, v/y] : C'[\langle u, v\rangle/z]$$

The type is preserved consistently, and the conclusion holds.

(3) **Sum Type Reduction: ι-reduction**

Consider the case $\mathsf{case}(\mathsf{inl}(u), x.M, y.N) \to M[u/x]$:

- From the premise $\Gamma \vdash \mathsf{case}(\mathsf{inl}(u), x.M, y.N) : A$, it follows that:
  (a) $\Gamma \vdash \mathsf{inl}(u) : B + C$
  (b) $\Gamma, x : B \vdash M : A$ and $\Gamma, y : C \vdash N : A$.
- From (a), by inversion of the $+$-introduction rule, $\Gamma \vdash u : B$.
- Applying the substitution lemma (lemma 1), from $\Gamma, x : B \vdash M : A$ and $\Gamma \vdash u : B$, we obtain $\Gamma \vdash M[u/x] : A[u/x]$.
- If A does not depend on the judgment term, then $A[u/x] = A$, and the conclusion holds. (For dependent type cases, substitution similarly preserves type consistency).

(4) **Local Definition Reduction: ζ-reduction**

Consider the case $\mathsf{let}\ x = u\ \mathsf{in}\ t \to t[u/x]$:

- From the premise $\Gamma \vdash \mathsf{let}\ x = u\ \mathsf{in}\ t : A$, there must exist a type B such that $\Gamma \vdash u : B$ and $\Gamma, x : B \vdash t : A$.
- This is a standard application scenario of the substitution lemma. According to lemma 1, we directly derive $\Gamma \vdash t[u/x] : A[u/x]$.

(5) **Definition Unfolding Reduction: δ-reduction**

Consider the case $c \to t$, where $(c := t : A) \in \Gamma$:

- From the premise $\Gamma \vdash c : A$, c is a constant declared in the context.
- According to the definition of δ-reduction, the type of the identifier c is completely consistent with the type of its definition body t in Γ.
- Thus, by the well-formedness of Γ, we directly obtain $\Gamma \vdash t : A$.

(6) **Congruence Cases**

If the reduction occurs in a subterm, e.g., $t = f\,u \to f'\,u$ (where $f \to f'$):

- By the induction hypothesis, f' preserves the type $\Pi(x : B).A$ of f.
- Reapply the Π-elimination rule; the overall term's type remains $A[u/x]$.
- Similarly, all other constructions (pairing, projection, injection, etc.) preserve types under congruence reductions.

Since all basic computational reductions (β, ι, etc.) satisfy type preservation, and the reduction relation $\to$ is closed under context construction, by structural induction on the reduction relation, the theorem holds for all reduction steps. □

We interpret the type A as a set of terms $[\![A]\!]$. These sets must satisfy the reducibility candidate (CR) properties mentioned earlier.

Definition 9. *Type Semantics*

- $[\![\mathit{Val}]\!] = \{t \mid t \in \mathit{SN} \wedge t$ *ultimately reduces to a numeric constant* $\}$
- $[\![\mathit{Time}]\!] = \{t \mid t \in \mathit{SN} \wedge t$ *ultimately reduces to a valid timestamp* $\}$
- $[\![\Pi(x : A).B]\!] = \{f \mid \forall u \in [\![A]\!], (f\,u) \in [\![B]\!][u/x]\}$
- $[\![\Sigma(x : A).B]\!] = \{p \mid p \twoheadrightarrow \langle u, v\rangle \wedge u \in [\![A]\!] \wedge v \in [\![B]\!][u/x]\}$
- $[\![A + B]\!] = \{t \mid t \in \mathit{SN} \wedge (t \twoheadrightarrow \mathit{inl}(u) \Rightarrow u \in [\![A]\!]) \wedge (t \twoheadrightarrow \mathit{inr}(v) \Rightarrow v \in [\![B]\!])\}$
- $[\![\mathit{Id}_A(a, b)]\!] = \{\mathit{refl} \mid a, b \in [\![A]\!] \wedge a \simeq_{Red} b\}$. *Where* $\simeq_{Red}$ *indicates that they reduce to the same normal form.*

The interpretation function $[\![t]\!]_\rho$ is responsible for converting syntactic terms with variables into their corresponding semantic values. In strong normalization proofs, this "interpretation" is typically the substitution (Substitution) operation.

Definition 10. *Semantic Interpretation of Terms*

Let ρ be a mapping from variables to semantic values (assignment).

- $[\![x]\!]_\rho = \rho(x)$ *(directly read the assignment from the environment)*
- $[\![\lambda x : A.M]\!]_\rho =$ *a function* $v \mapsto [\![M]\!]_{\rho[x \mapsto v]}$
- $[\![f\ a]\!]_\rho = [\![f]\!]_\rho([\![a]\!]_\rho)$ *(function application)*
- $[\![\langle a, b\rangle]\!]_\rho = ([\![a]\!]_\rho, [\![b]\!]_\rho)$ *(semantic pairing)*
- $[\![\mathit{refl}]\!]_\rho = \mathit{refl}$ *(constant interpreted as itself)*

To ensure $[\![t]\!]_\rho \in [\![A]\!]$, the assignment ρ must be "valid."

Definition 11. *Logical Closure of Prop Semantics In the semantic model $\mathcal{M}$, the interpretation $[\![\mathsf{Prop}]\!]$ of Prop is defined as the collection of all term sets S satisfying the following properties:*

(1) ***SN Property**: $S \subseteq \mathsf{SN}$.*

(2) ***CR Property**: S is closed under reduction and reverse reduction, and contains all neutral terms.*

(3) ***Impredicative Semantic Operator**: For any set X and function $F : X \to [\![\mathsf{Prop}]\!]$, the intersection operation is defined as:*

$$[\![\Pi(x : A).B]\!]_\rho = \bigcap_{u \in [\![A]\!]_\rho} \{f \mid (f\,u) \in [\![B]\!]_{\rho[x \mapsto u]}\}$$

***Closure Correction**: Since $[\![\mathsf{Prop}]\!]$ includes all Π-type interpretations it constructs itself (via intersection operations on the reducibility candidate $\mathcal{RC}$), this guarantees that even if A is an infinitely large Universe, the mapped result remains within the predefined set of $[\![\mathsf{Prop}]\!]$.*

Definition 12. *Valid Assignment*

An assignment ρ is said to satisfy the context Γ (denoted $\rho \models \Gamma$), if and only if for every binding $(x : A)$ in Γ,

$$\rho(x) \in [\![A]\!]_\rho$$

Theorem 3. *Semantic Soundness*

$$\textit{If } \Gamma \vdash t : A \textit{ and } \rho \models \Gamma, \textit{ then } [\![t]\!]_\rho \in [\![A]\!]_\rho.$$

Proof. Proceed by induction on the structure of the derivation tree $\Gamma \vdash t : A$. Due to the impredicativity of Prop, the construction of its reducibility candidates ($\mathcal{RC}$) is based on Girard's stratified candidate set method, rather than simple Tarski-style semantics.

A. Variable Rule (Variable)

Assume the derivation is

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$$

- Proof: From the premise $\rho \models \Gamma$, the assignment for each bound variable in the environment must belong to the interpretation of that type. Thus, $\rho(x) \in [\![A]\!]_\rho$.
- Since $[\![x]\!]_\rho = \rho(x)$, the conclusion $[\![x]\!]_\rho \in [\![A]\!]_\rho$ holds.

B. Π-Type Introduction (λ-Abstraction)

Assume the last derivation step is

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : \Pi(x : A).B}$$

- Goal: Prove $[\![\lambda x : A.M]\!]_\rho \in [\![\Pi(x : A).B]\!]_\rho$.
- According to the semantics of Π, need to prove: For any $u \in [\![A]\!]_\rho$, $([\![\lambda x : A.M]\!]_\rho \cdot u) \in [\![B]\!]_{\rho[x \mapsto u]}$.
- By the term interpretation definition, $([\![\lambda x : A.M]\!]_\rho \cdot u) \to_\beta [\![M]\!]_{\rho[x \mapsto u]}$.
- Since $u \in [\![A]\!]_\rho$ and $\rho \models \Gamma$, the new assignment $\rho' = \rho[x \mapsto u]$ satisfies $\rho' \models (\Gamma, x : A)$.
- Apply the induction hypothesis: $[\![M]\!]_{\rho'} \in [\![B]\!]_{\rho'}$.
- Since the reducibility set is closed under reverse reduction (CR property), the original application term also belongs to that set.

C. Π-Type Elimination (Application)

Assume the last derivation step is the application rule:

$$\frac{\Gamma \vdash f : \Pi(x : A).B \quad \Gamma \vdash u : A}{\Gamma \vdash f\,u : B[u/x]}$$

- Proof: By the induction hypothesis, $[\![f]\!]_\rho \in [\![\Pi(x:A).B]\!]_\rho$ and $[\![u]\!]_\rho \in [\![A]\!]_\rho$.
- According to the semantics of the Π type, the function $[\![f]\!]_\rho$ applied to any element in $[\![A]\!]_\rho$ must belong to the interpretation of B.
- Therefore, $[\![f]\!]_\rho([\![u]\!]_\rho) \in [\![B]\!]_{\rho[x \mapsto [\![u]\!]_\rho]}$.
- By the substitution lemma, this set is $[\![B[u/x]]\!]_\rho$.

D. Σ-Type Introduction (Pairing)

Assume the derivation is:

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B[a/x]}{\Gamma \vdash \langle a, b\rangle : \Sigma(x:A).B}$$

(1) By the induction hypothesis, $[\![a]\!]_\rho \in [\![A]\!]_\rho$.

(2) By the induction hypothesis, $[\![b]\!]_\rho \in [\![B[a/x]]\!]_\rho$.

(3) According to the semantics of Σ: $[\![\Sigma(x:A).B]\!]_\rho = \{(u,v) \mid u \in [\![A]\!]_\rho, v \in [\![B]\!]_{\rho[x\mapsto u]}\}$.

(4) Combining with the substitution lemma $[\![B[a/x]]\!]_\rho = [\![B]\!]_{\rho[x \mapsto [\![a]\!]_\rho]}$, the two components of $[\![\langle a,b\rangle]\!]_\rho$ fully conform to the definition.

E. Dependent Sum Elimination (Σ-Elimination / split)

Assume the last derivation step applies the Σ elimination rule:

$$\frac{\Gamma \vdash p : \Sigma(x:A).B \quad \Gamma, x:A, y:B \vdash t : C(\langle x,y\rangle)}{\Gamma \vdash \mathsf{split}(p, x.y.t) : C(p)}$$

We need to prove: If $\rho \models \Gamma$, then $[\![\mathsf{split}(p, x.y.t)]\!]_\rho \in [\![C(p)]\!]_\rho$.

(1) **Semantic Premise Derivation**
By the induction hypothesis (Inductive Hypothesis):

- For term p, $[\![p]\!]_\rho \in [\![\Sigma(x:A).B]\!]_\rho$.
- According to the semantics of the Σ type, there exist $u \in [\![A]\!]_\rho$ and $v \in [\![B]\!]_{\rho[x\mapsto u]}$, such that $[\![p]\!]_\rho$ is semantically equivalent to the pair (u, v).

(2) **Construct Valid Assignment**
Define a new assignment $\rho' = \rho[x \mapsto u, y \mapsto v]$.

- Since $u \in [\![A]\!]_\rho$, $\rho[x \mapsto u] \models (\Gamma, x : A)$.
- Since $v \in [\![B]\!]_{\rho[x\mapsto u]}$, and $\rho[x \mapsto u]$ satisfies the preceding context, $\rho' \models (\Gamma, x : A, y : B)$.

(3) **Apply Induction Hypothesis**
Apply the induction hypothesis to the elimination body t:

$$[\![t]\!]_{\rho'} \in [\![C(\langle x,y\rangle)]\!]_{\rho'}$$

According to the term semantic interpretation definition:

$$[\![\mathsf{split}(p, x.y.t)]\!]_\rho = [\![t]\!]_{\rho[x\mapsto \pi_1([\![p]\!]_\rho), y \mapsto \pi_2([\![p]\!]_\rho)]}$$

Substituting the components u and v of $[\![p]\!]_\rho$ yields:

$$[\![\mathsf{split}(p, x.y.t)]\!]_\rho = [\![t]\!]_{\rho'}$$

(4) **Type Consistency (Conversion)**
To complete the proof, ensure the result belongs to the consistent set.
According to the substitution property of dependent types:

$$[\![C(\langle x, y\rangle)]\!]_{\rho'} = [\![C]\!]_{\rho[p' \mapsto (u,v)]}$$

Since $[\![p]\!]_\rho \simeq (u, v)$ (in the sense of reducibility equivalence), and the Red set is closed under computational equivalence:

$$[\![C]\!]_{\rho[p' \mapsto [\![p]\!]_\rho]} = [\![C(p)]\!]_\rho$$

Therefore, $[\![t]\!]_{\rho'} \in [\![C(p)]\!]_\rho$, and the conclusion holds.

F. Semantic Preservation of ι-reduction (Reduction Invariance)

To support the above steps, we need to prove that ι-reduction steps do not change semantic properties. For the Σ type:

$$\mathsf{split}(\langle u, v\rangle, x.y.t) \to_\iota t[u/x, v/y]$$

(1) **Semantic Consistency**: By definition, the interpretation of the left side $[\![\mathsf{split}(\langle u, v\rangle, x.y.t)]\!]_\rho$ expands to $[\![t]\!]_{\rho[x \mapsto [\![u]\!]_\rho, y \mapsto [\![v]\!]_\rho]}$.

(2) **Substitution Lemma**: According to the substitution lemma (Lemma 1), the interpretation of the right side $[\![t[u/x, v/y]]\!]_\rho$ is semantically identical to the above expansion.

(3) **Conclusion**: Since the semantic interpretations of both are the same element in the set-theoretic sense, and Red_C satisfies CR 2 (stability), the reduced term remains in the corresponding reducibility set.

G. Sum Type Introduction (Injection)

Assume the last derivation step is the left injection rule (right injection inr similarly):

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash B : \mathcal{U}}{\Gamma \vdash \mathsf{inl}_B(a) : A + B}$$

- **Proof**: By the induction hypothesis, $[\![a]\!]_\rho \in [\![A]\!]_\rho$.
- According to the semantics of sum types: $[\![A + B]\!]_\rho = \{\mathsf{inl}(u) \mid u \in [\![A]\!]_\rho\} \cup \{\mathsf{inr}(v) \mid v \in [\![B]\!]_\rho\} \cup \mathsf{Neutral}$.
- Since $[\![\mathsf{inl}_B(a)]\!]_\rho = \mathsf{inl}([\![a]\!]_\rho)$, and $[\![a]\!]_\rho \in [\![A]\!]_\rho$.
- By set construction, $\mathsf{inl}([\![a]\!]_\rho)$ obviously belongs to the left branch definition part of $[\![A + B]\!]_\rho$.

H. Sum Type Elimination (+-Elimination / case)

Assume the last derivation step applies the + elimination rule:

$$\frac{\Gamma \vdash s : A + B \quad \Gamma, x : A \vdash t : C \quad \Gamma, y : B \vdash u : C}{\Gamma \vdash \mathsf{case}(s, x.t, y.u) : C}$$

We need to prove: If $\rho \models \Gamma$, then $[\![\mathsf{case}(s, x.t, y.u)]\!]_\rho \in [\![C]\!]_\rho$.

(1) **Branch Premise Analysis**
By the induction hypothesis (IH):

- For the judgment term s, $[\![s]\!]_\rho \in [\![A + B]\!]_\rho$.
- According to the semantics of $A + B$, $[\![s]\!]_\rho$ must be strongly normalizing (SN) and ultimately reduce to the form $\mathsf{inl}(a)$ or $\mathsf{inr}(b)$.

(2) **Branch Discussion (Case Analysis)**
We need to discuss two semantic paths:

- **Path One: Left Injection (inl)**

  (1) Assume $[\![s]\!]_\rho \twoheadrightarrow \mathsf{inl}(a)$; by definition, $a \in [\![A]\!]_\rho$.
  (2) Construct the assignment $\rho_x = \rho[x \mapsto a]$. Since $a \in [\![A]\!]_\rho$, $\rho_x \models (\Gamma, x : A)$.
  (3) Apply the induction hypothesis to the left branch term t: $[\![t]\!]_{\rho_x} \in [\![C]\!]_{\rho_x}$.
  (4) Since C in this rule does not depend on the specific value of s (simple elimination case), $[\![C]\!]_{\rho_x} = [\![C]\!]_\rho$.

- **Path Two: Right Injection (inr)**

  (1) Assume $[\![s]\!]_\rho \twoheadrightarrow \mathsf{inr}(b)$; by definition, $b \in [\![B]\!]_\rho$.
  (2) Construct the assignment $\rho_y = \rho[y \mapsto b]$. Then $\rho_y \models (\Gamma, y : B)$.
  (3) Apply the induction hypothesis to the right branch term u: $[\![u]\!]_{\rho_y} \in [\![C]\!]_{\rho_y}$.
  (4) Similarly, $[\![u]\!]_{\rho_y} \in [\![C]\!]_\rho$.

(3) **Unification of Semantic Interpretation**
According to the semantic definition of the case operator:

$$[\![\mathsf{case}(s, x.t, y.u)]\!]_\rho = \begin{cases} [\![t]\!]_{\rho[x \mapsto a]} & \text{if } [\![s]\!]_\rho \twoheadrightarrow \mathsf{inl}(a) \\ [\![u]\!]_{\rho[y \mapsto b]} & \text{if } [\![s]\!]_\rho \twoheadrightarrow \mathsf{inr}(b) \end{cases}$$

Regardless of which branch $[\![s]\!]_\rho$ collapses to, the result belongs to $[\![C]\!]_\rho$.

(4) **Reverse Reduction Closure (CR 3 Application)**
Since $\mathsf{case}(s, x.t, y.u)$ reaches $[\![t]\!]_{\rho_x}$ or $[\![u]\!]_{\rho_y}$ via reduction (ι-reduction), according to the CR 3 property of the reducibility set (and closure under reverse reduction), the original case term itself must also belong to the reducibility set $[\![C]\!]_\rho$. The conclusion holds.

□

Soundness means "everything that can be proven is true." In Theorem 3, "what can be proven" is the type judgment $\Gamma \vdash t : A$, and "true" is the term's membership in the semantic model $[\![t]\!] \in [\![A]\!]$. It ensures that the syntactic constructions of KOS-TL Core do not deviate from their logical semantics.

Theorem 4. *Consistency Theorem*

There does not exist a term t such that $\emptyset \vdash t : \bot$. The system is thus called consistent.

Proof. The core idea of this proof is that syntactic derivations cannot escape semantic boundaries. We unfold the argument in three stages.

Stage 1: Using Strong Normalization (Syntactic Normalization)

According to the strong normalization theorem of KOS-TL Core, if there exists a term t satisfying $\emptyset \vdash t : \bot$, then t must reduce to a normal form (Normal Form) t_{nf}, with the type preserved:

$$\emptyset \vdash t_{nf} : \bot$$

In the empty context, the normal form can only be a constructor term (Constructor). However, according to the definition of the $\bot$ type, it has no introduction rules (Introduction Rules), meaning no constructor can produce $\bot$. This implies that, at the syntactic level, t_{nf} does not exist.

Stage 2: Semantic Soundness Mapping

To make the argument mathematically irrefutable, we use the interpretation model $\mathcal{M}$.

Step 1: Establish the Mapping

The interpretation function $[\![\cdot]\!]$ maps the syntactic world (Types/Terms) to the semantic world (Sets/Elements).

- For any type A, its interpretation $[\![A]\!]$ is a set.
- For any term $t : A$, its interpretation $[\![t]\!]$ must be an element in the set $[\![A]\!]$.

Step 2: Special Nature of the Empty Type

When defining the semantics of $\bot$, we map it to the mathematical absolute empty set:

$$[\![\bot]\!] = \emptyset$$

This is reasonable, as the logical "false" corresponds in model theory to a state with no witnesses (Witness).

Step 3: Apply Semantic Soundness

According to the semantic soundness theorem 3:

$$\text{If } \Gamma \vdash t : A, \text{ then for all valid assignments } \rho, [\![t]\!]_\rho \in [\![A]\!]$$

In the empty context $\emptyset$, no assignment ρ is needed, directly yielding:

$$[\![t]\!] \in [\![\bot]\!]$$

Stage 3: Reduction to Absurdity and Contradiction

(1) From the above derivation: $[\![t]\!] \in \emptyset$.

(2) According to the set theory axioms of extensionality and the empty set: $\forall x, x \notin \emptyset$.

(3) Judgment: $[\![t]\!] \in \emptyset$ and $\forall x, x \notin \emptyset$ constitute a direct logical contradiction.

(4) Backtracking: Since the semantic interpretation and set theory axioms are presupposed to be correct, the source of the contradiction can only be the assumption "existence of term t."

Conclusion: The assumption does not hold, $\neg\exists t, \emptyset \vdash t : \bot$. Consistency is proven. □

In systems like KOS-TL Core based on the Curry-Howard isomorphism, logical consistency is equivalent to proving that the empty type (Empty Type) is uninhabited (Uninhabited).

Corollary: Logical Consistency (Consistency) Since the KOS-TL core layer satisfies strong normalization (SN) and has type preservation (Subject Reduction), and there is no constructor for the empty type $\bot$ in the system, there does not exist a term t such that $\vdash t : \bot$. This proves that the core layer, as the "formal constitution," is logically contradiction-free.

Definition 13. *Confluence*

For any Core layer terms M, N, P, if there exist reduction paths such that $M \twoheadrightarrow N$ and $M \twoheadrightarrow P$, then there necessarily exists a term Q such that $N \twoheadrightarrow Q$ and $P \twoheadrightarrow Q$.

Let $\rightarrow$ be the one-step reduction relation (including $\beta, \delta, \zeta, \eta$ reductions), and $\twoheadrightarrow$ its transitive closure (multi-step reduction). Confluence logically means that all forks eventually converge.

Theorem 5. *KOS-TL Core Confluence Theorem*

All well-formed terms in the KOS-TL Core layer satisfy confluence and have a unique normal form (Unique Normal Form).

Proof. We adopt the Tait-Martin-Löf parallel reduction method (Parallel Reduction) combined with the strong normalization property for the proof.

Step A: Define Parallel Reduction ($\Rightarrow$)

To handle the "simultaneous reduction of forks" that single-step reduction cannot cover, we define a parallel reduction relation $\Rightarrow$, which is an extension of the single-step reduction $\rightarrow$, allowing simultaneous reduction of multiple subparts of a term. Specifically, $\Rightarrow$ is the minimal relation satisfying the following rules:

(1) If $M \rightarrow M'$ (single-step β-reduction), then $M \Rightarrow M'$.

(2) For any reduction context $C[\cdot]$, if $M \Rightarrow M'$, then $C[M] \Rightarrow C[M']$.

(3) Parallel application of reduction:

$$\frac{M \Rightarrow M' \quad N \Rightarrow N'}{(\lambda x.M)N \Rightarrow M'[x := N']}$$

where $[x := N']$ denotes capture-avoiding substitution.

(4) For constructors (e.g., $\Pi x : A.M$), allow simultaneous reduction of domain and body:

$$\frac{A \Rightarrow A' \quad M \Rightarrow M'}{\Pi x : A.M \Rightarrow \Pi x : A'.M'}$$

(Σ and Id types are similar).

Step B: Prove the Diamond Property

We prove that the parallel reduction $\Rightarrow$ satisfies the diamond property: If $M \Rightarrow N$ and $M \Rightarrow P$, then there exists Q such that $N \Rightarrow Q$ and $P \Rightarrow Q$. The proof proceeds by induction on the structure of M:

- **Base Case**: If M is a variable or atom, then $N = P = M$, take $Q = M$.
- **Inductive Case**: Assume it holds for all proper subterms. **Case:** $M = (\lambda x.M_1)M_2$: By the definition of $\Rightarrow$, N and P arise from different sub-reduction forks. The induction hypothesis applies to M_1 and M_2, yielding Q_1, Q_2 such that the subterms converge, then $Q = Q_1[x := Q_2]$. **Case: Constructor (e.g.,** $\Pi x : A.M$**)**: Constructors in KOS-TL Core are orthogonal (no overlapping reduction rules); induction applies to A and M, converging to $\Pi x : A''.M''$. The Id type is similar, with no internal conflicts.

By orthogonality and type preservation, reductions have no critical pairs, ensuring the diamond property.

Step C: Deriving Multi-Step Reduction from Parallel Reduction

The multi-step reduction $\twoheadrightarrow$ is the reflexive-transitive closure of $\Rightarrow$: $M \twoheadrightarrow N$ if and only if there exists a chain $M = M_0 \Rightarrow M_1 \Rightarrow \cdots \Rightarrow M_k = N$. By Newman's lemma, if $\Rightarrow$ satisfies the diamond property and the system is strongly normalizing (SN, no infinite chains), then $\twoheadrightarrow$ satisfies confluence: For $M \twoheadrightarrow N_1$ and $M \twoheadrightarrow N_2$, there exists Q such that $N_1 \twoheadrightarrow Q$ and $N_2 \twoheadrightarrow Q$. SN ensures weak normalization, and the diamond property implies local confluence.

Step D: Unique Normal Form Proof

Assume M has two normal forms N_1 and N_2 (irreducible). By confluence, there exists Q such that $N_1 \twoheadrightarrow Q$ and $N_2 \twoheadrightarrow Q$. But N_1, N_2 are normal forms, hence $N_1 = Q = N_2$. By SN, every term has a normal form, hence it is unique. □

The decidability of KOS-TL Core is primarily supported by the following two properties:

(1) **Decidability of Type Checking**: Given Γ, t, A, determine whether $\Gamma \vdash t : A$ holds.

(2) **Decidability of Equivalence Judgment**: Given Γ, t, u, determine whether $t \equiv u$ (whether the two terms are logically equivalent in the current context) holds.

As the mathematical foundation of the entire system, the Core layer must theoretically guarantee that all basic operations (type checking, equivalence judgment) terminate in a finite number of steps under any circumstances.

Theorem 6. *Core Layer Decidability Theorem*

The type checking problem and term equivalence judgment problem in the KOS-TL Core language are decidable.

Proof. This proof is built on the foundations of strong normalization (Strong Normalization, SN) and confluence (Confluence).

1. Finiteness of Reduction Sequences

According to the strong normalization theorem proven by the Tait-Girard method, any well-formed term t in KOS-TL Core has no infinite reduction sequences. This means that starting from any term, through reduction steps such as β, δ, ζ, etc., it inevitably reaches a unique normal form $\mathsf{nf}(t)$ in a finite number of steps.

2. Algorithmization of Equivalence Judgment

The process of determining $t \equiv u$ can be transformed into:

(1) Reduce t to normal form t^*.

(2) Reduce u to normal form u^*.

(3) Compare whether t^* and u^* are syntactically identical.

Since steps 1 and 2 are guaranteed to complete in finite time, and step 3 is simple symbol matching, $t \equiv u$ is decidable.

3. Recursive Termination of Type Checking Algorithm

The type checker processes term t recursively according to its syntactic structure:

- For application terms $(f\ a)$, check if the type of f is a Π-type and determine if the type of a matches.
- During type matching, invoke the aforementioned equivalence judgment algorithm.

Since the term construction is finite and equivalence judgment is decidable, the entire recursive process must terminate. □

3.3 Application Example: Quality Anomaly Knowledge Modeling

In manufacturing scenarios, a "qualified batch" is not just a data record; it must include evidence of passing quality inspection.

- **Type Definition**:

$$\mathsf{QualifiedBatch} \equiv \Sigma(b : \mathsf{BatchID}).\Sigma(res : \mathsf{Result}).\mathsf{Id}_{\mathsf{Result}}(res, \mathsf{Pass})$$

- **Logical Interpretation**: This type requires that any instance must contain a batch ID, a quality inspection result, and an identity term proving that the result equals Pass.
- **Construction Attempt**: If the batch's quality inspection result is $\mathsf{Failure}$, then since the type is empty (no constructors), the system will reject instantiation of the object at the core layer, thereby preventing unqualified products from entering subsequent processes at the logical layer.

Example 1. *Constructing a Temperature Reading Within Safe Range*

(1) Declare atomic predicates (as axioms or basic judgments)

In the initialization context of the Core layer, we need to introduce a dimensional judgment predicate (is_unit):

$$\Gamma \vdash \textsf{\textit{is\_unit\_Celsius}} : \Pi(v : \textsf{\textit{Val}}).\textsf{\textit{Prop}}$$

Additionally, introduce the comparison predicate construction $\textsf{\textit{is\_safe}}$*:*

$$\Gamma \vdash L, H : \textsf{\textit{Val}} \quad \Gamma \vdash \textsf{\textit{is\_safe}} \equiv \lambda v.\textsf{\textit{And}}(L \leq v, v \leq H) : \textsf{\textit{Val}} \to \textsf{\textit{Prop}}$$

(2) Construct the $Temp$ Type

Using the Σ-type construction rule:

$$\frac{\Gamma \vdash \textsf{\textit{Val}} : \mathcal{U} \quad \Gamma, v : \textsf{\textit{Val}} \vdash \textsf{\textit{is\_unit\_Celsius}}(v) : \textsf{\textit{Prop}}}{\Gamma \vdash \Sigma(v : \textsf{\textit{Val}}).\textsf{\textit{is\_unit\_Celsius}}(v) : \mathcal{U}}$$

At this point, $Temp \equiv \Sigma(v : \textsf{\textit{Val}}).\textsf{\textit{is\_unit\_Celsius}}(v)$ formally becomes a valid type.

(3) Construct $QualifiedTemp$ on This Basis

Now, we overlay the logical safety predicate $\textsf{\textit{is\_safe}}$*:*

$$QualifiedTemp \equiv \Sigma(t : Temp).\textsf{\textit{is\_safe}}(\textsf{\textit{proj}}_1(t))$$

(4) Construct Object Instance

Assume $v = 25$: $p_{unit} : \textsf{\textit{is\_unit\_Celsius}}(25)$ $p_{range} : \textsf{\textit{is\_safe}}(25)$ $obj = \langle\langle 25, p_{unit}\rangle, p_{range}\rangle$

4 KOS-TL Kernel Layer: State Evolution and Operational Semantics

The Kernel Layer is the dynamical core of KOS-TL. Based on the static type system of the Core layer, it introduces a time dimension and state transition mechanism, realizing the atomic evolution of the knowledge base from state Σ to Σ' through controlled event-driven processes.

4.1 Syntax

The KOS-TL Kernel Layer introduces the concept of "dynamics," extending the domain from static objects to state transition trajectories. It serves as the bridge connecting logic and execution.

The KOS-TL Kernel Layer can be represented as a triple:

$$\langle \Sigma, \mathsf{Ev}, \Delta \rangle$$

Definition 14. *State (Σ)*

The Kernel Layer state (Σ) is defined as a configuration triple:

$$\Sigma \equiv \langle \mathcal{K}, \mathcal{TS}, \mathcal{P} \rangle$$

(1) Knowledge Base ($\mathcal{K}$ - Knowledge Base)

$$\mathcal{K} = \{(id_i, t_i, A_i) \mid \Gamma_{Core} \vdash t_i : A_i\}$$

It stores all currently verified truths (Facts) in the system.

(2) Logical Clock ($\mathcal{TS}$ - Logical Clock) Based on the Core layer base sort Time. It is not merely a counter but a measure of the total order of states.

$$\mathcal{TS} : \mathsf{Time} \quad \textit{satisfying the monotonicity rule: } \Sigma \to \Sigma' \implies \mathcal{T}' > \mathcal{T}$$

(3) Pending Queue ($\mathcal{P}$ - Pending Events) An ordered sequence composed of restricted events (Events).

Definition 15. *Event Ev An event Ev is a well-formed quintuple under the global context Γ:*

$$\mathsf{Ev} \equiv \langle \mathsf{Args}, \mathsf{Pre}, \mathsf{Op}, \mathsf{Post}, \mathsf{Prf} \rangle$$

The type constraints and logical semantics of each component are as follows:

(1) Args (Argument Set):

$$\mathsf{Args} : A$$

(where $A \in \mathcal{U}_{Core}$). Instantiated data ingested from the external world (Runtime layer), such as sensor_value or transaction_amount.

(2) Pre (Precondition Predicate):

$$\mathsf{Pre} : \mathsf{Args} \to \Sigma \to \mathsf{Prop}$$

A dependent proposition defining the logical prerequisites that the event must satisfy in the current state Σ. It can reference the current knowledge base $\mathcal{K}$ or logical time $\mathcal{TS}$.

(3) Op (Operation Operator):

$$\mathsf{Op} : \mathsf{Args} \to \Sigma \to \Sigma$$

The core evolution function. It describes how to generate the new state Σ'.

- *$\mathcal{K} \to \mathcal{K}'$: Addition and removal of knowledge items.*
- *$\mathcal{TS} \to \mathcal{TS} + \Delta t$: Stepping of the logical clock.*
- *$\mathcal{P} \to \mathcal{P}'$: Update of the pending intent queue (consuming itself or deriving new intents).*

(4) Post (Postcondition/Invariant):

$$\mathsf{Post} : \Sigma' \to \mathsf{Prop}$$

Defines safety criteria (Safety Properties) that must be satisfied after the transformation, such as "energy conservation," "non-negative account," or "clock monotonicity."

(5) Prf (Proof Term):

$$\mathsf{Prf} : \mathsf{Pre}(\mathsf{Args}, \Sigma)$$

This is the "passport" of the event. When signals are refined into events in the Runtime layer, a constructive proof that the precondition holds must be constructed. If there is no valid Prf, the kernel will reject execution of the event.

Definition 16. *Transition Record (Δ)*

To support causal tracing, Δ needs to record clock jumps:

$$\Delta \subseteq \Sigma \times \textsf{\textit{Ev}} \times \Sigma$$

A typical transition record item:

$$\delta = \langle\langle \mathcal{K}, \mathcal{TS}, \mathcal{P}\rangle \xrightarrow{e} \langle \mathcal{K}', \mathcal{TS}', \mathcal{P}'\rangle\rangle$$

$$\forall \langle \Sigma \xrightarrow{e} \Sigma'\rangle \in \Delta, \quad \Sigma'.\mathcal{T} > \Sigma.\mathcal{T}$$

. This theoretically locks the "arrow of time," ensuring the irreversibility of knowledge evolution. Every new knowledge item ku_{new} injected into $\mathcal{K}'$ implicitly carries the current $\mathcal{T}'$. During a transition, e is dequeued from $\mathcal{P}$, and after executing **Op***, its result is merged into $\mathcal{K}'$.*

Definition 17. *Evolutionary Determinism*

Given a state Σ and event e, if there exists a Σ' satisfying the operational semantics, then its normal form (Normal Form) is unique in the sense of intensional equivalence.

4.2 Operational Semantics

The evolution of the Kernel Layer follows "Small-step Operational Semantics." Let Σ be the system's configuration (Configuration); its state transition rules are defined as:

$$\frac{e = \mathsf{head}(\Sigma.\mathcal{P}) \quad \Gamma, \Sigma.\mathcal{K}, \Sigma.\mathcal{TS} \vdash p : \mathsf{Pre}(\mathsf{Args}_e, \Sigma) \quad \Sigma' = \mathsf{Op}(\mathsf{Args}_e, \Sigma) \quad \Sigma' \vdash p' : \mathsf{Post}(\mathsf{Args}_e, \Sigma')}{\langle \Sigma, e\rangle \longrightarrow_{KOS} \Sigma'} \tag{1}$$

where:

- Intent Trigger Condition ($e = \mathsf{head}(\Sigma.\mathcal{P})$)
  This specifies the source of the event e. Transitions are not random but driven by the head event of the pending queue $\mathcal{P}$. This ensures the ordered nature of evolution, meaning the kernel schedules according to the logical order of the intent queue.
- Environment-Aware Proof Judgment ($\Gamma, \Sigma.\mathcal{K}, \Sigma.\mathcal{T} \vdash p$)
  Explicitly lists the context on which the proof term p depends. The proof of the precondition not only depends on the global context Γ but must also be consistent with facts in the current knowledge base $\mathcal{K}$ and the logical time $\mathcal{TS}$. This implements the logic that events can only occur on the correct time and facts.
- Parameterized Operator Application ($\Sigma' = \mathsf{Op}(\mathsf{Args}_e, \Sigma)$)
  Introduces Args_e. Emphasizes that the transition is an overall transformation of the current triple configuration based on the specific parameters carried by the event (refined from the Runtime layer).
- Postcondition Self-Consistency ($\Sigma' \vdash p' : \mathsf{Post}(\mathsf{Args}_e, \Sigma')$)
  Explicitly states that Post is judged under the new state Σ'. This defines the hard threshold for logical commit (Commit). If the evolved state cannot satisfy its safety invariant (e.g., the clock did not step forward, or the knowledge base has inconsistencies), the judgment fails, and the transition rule is invalid.

This semantics stipulates that a valid knowledge transition must simultaneously satisfy "provable premise" and "compliant result". If any condition cannot be proven in the Core layer, the state remains unchanged (i.e., execution rollback). The Kernel layer does not handle retry strategies for physical failures; it only defines "logically valid evolution trajectories".Any physical attempt that fails Post validation manifests as an "unoccurred transition" in the Kernel layer, thereby enforcing atomicity of transactions at the logical layer.

Example 2. *Reduction Example Demonstration*

Assume the following scenario. The sensor data fusion assumes there are two independent sensors in the system: ku_1 (temperature) and ku_2 (humidity). We need a merge function combine to encapsulate them into an "environment state" object.

(1) Basic Type and Predicate Definitions The target type $Env \equiv \Sigma(t : Temp).(Humi)$, where the environment state is a dependent pair containing temperature and humidity. Among them:

$$Temp \equiv \Sigma(v : \textsf{\textit{Val}}).\textsf{\textit{is\_T}}(v)$$
$$Humi \equiv \Sigma(v : \textsf{\textit{Val}}).\textsf{\textit{is\_H}}(v)$$

(2) Specific Knowledge Items (Instances) [[$ku_1 = \langle 25, p_T\rangle : Temp\; ku_2 = \langle 60, p_H\rangle : Humi$]]

(3) Merge Function (Π-type) Define a function that receives temperature and humidity and returns an environment object:

$$\textsf{combine} \equiv \lambda t : Temp.\lambda h : Humi.\langle t, h\rangle$$

Its type is $\Pi(t : Temp).\Pi(h : Humi).Env$.

Now we demonstrate the reduction process of applying **combine** *to ku_1 and ku_2. This typically occurs when the Kernel receives two signals and attempts to update the global state.*

Step 1: Construct the Initial Application Term

In the Kernel's control flow, a term to be reduced is generated:

$$(\textsf{combine}\ ku_1)\ ku_2$$

Step 2: First β-reduction (Substitute Temperature)

According to the β-reduction rule $(\lambda x.M)N \to M[N/x]$:

$$(\lambda t.\lambda h.\langle t, h\rangle)\ ku_1 \to \lambda h.\langle ku_1, h\rangle$$

The function "consumes" the temperature data, becoming a specialized function "waiting for humidity input."

Step 3: Second β-reduction (Substitute Humidity)

$$(\lambda h.\langle ku_1, h\rangle)\ ku_2 \to \langle ku_1, ku_2\rangle$$

The humidity data is filled in, generating a complete pair.

Step 4: Unfolding and Structural Reduction (ι-reduction)

If the system needs to further extract the original values (e.g., for executing analyze), then ι-reduction occurs:

$$\textsf{proj}_1(\langle ku_1, ku_2\rangle) \to ku_1 = \langle 25, p_T\rangle$$

$$\textsf{proj}_1(\textsf{proj}_1(\langle ku_1, ku_2\rangle)) \to 25$$

The above reduction processes are accompanied by type judgment. According to type preservation (Subject Reduction), every term in the entire reduction process must be well-formed: Initial term: $(\textsf{combine}\ ku_1)\ ku_2$ has type Env. Intermediate term: $\lambda h.\langle ku_1, h\rangle$ has type $\Pi(h : Humi).Env$. Final term: $\langle ku_1, ku_2\rangle$ has type Env. In this example, the reduction operation completes the transformation from "logical intent" (how to merge data) to "logical fact" (the merged data object). From the Core layer's perspective, $(\textsf{combine}\ ku_1)\ ku_2$ and $\langle ku_1, ku_2\rangle$ are judgmentally equal (Judgmentally Equal). They are different expressions of the same truth. From the Kernel layer's perspective, reduction is a computational evaluation. It consumes CPU cycles, merging two scattered memory pointers into a new Σ structure.

KOS-TL builds a "firewall" through static type semantics before logical execution, with type mismatch interception (Type Mismatch Interception) occurring before reduction. According to the Core layer's judgment rules, if a term (Term) cannot pass type checking, it will never be pushed into the Kernel's reduction engine.

Assume we have the merge function combine, which expects a humidity object $Humi$:

$$\textsf{combine} : \Pi(t : Temp).\Pi(h : Humi).Env$$

Now, the Runtime layer erroneously captures a pressure signal $p : Press$ and attempts to perform the merge:

$$(\textsf{combine}\ ku_1)\ p$$

The Kernel invokes the Π-elimination rule (Application Rule):

$$\frac{\Gamma \vdash f : \Pi(h : Humi).Env \quad \Gamma \vdash p : A}{\Gamma \vdash f\,p : Env[p/h] \quad (\text{requiring } A \equiv Humi)}$$

The kernel attempts to judge $Press \equiv Humi$. $Humi \equiv \Sigma(v : \textsf{Val}).\textsf{is\_H}(v)$ $Press \equiv \Sigma(v : \textsf{Val}).\textsf{is\_P}(v)$ Since the predicates $\textsf{is\_H} \neq \textsf{is\_P}$, type unification (Unification) fails. Thus, the term $(\textsf{combine}\ ku_1)\ p$ is judged ill-typed (Ill-typed). The reduction engine rejects the β-reduction execution, the system state σ remains unchanged, and a type error exception is triggered.

Example 3. *Causal Backtracking Analysis*

Based on Example 2, when the system discovers that although the merged result is "type correct", it is "logically anomalous" (e.g., the value of Env exceeds the safe range), it needs to use the Id type (identity type) for causal backtracking.

Assume we have obtained the merged object $obj = \langle ku_1, ku_2 \rangle$, but the analyze predicate deems it invalid. The backtracking process follows the following logical reduction:

(1) *Deconstruction Through the Core layer's projection operator* $\textsf{proj}_i$*, decompose the composite object back to the original evidence:*

$$t = \textsf{proj}_1(obj) \quad h = \textsf{proj}_2(obj)$$

(2) *Identity Verification The kernel constructs an equivalence statement, requiring proof that the current data is consistent with the input source:*

$$\textsf{Id}_{Temp}(t, ku_1) \wedge \textsf{Id}_{Humi}(h, ku_2)$$

If refl (reflexivity proof) cannot be constructed here, it indicates a computational error or memory corruption during merging.

(3) *Root Cause Localization The backtracking analysis function analyze will reverse-search along the reduction steps. In KOS-TL, this manifests as checking the proof term: inspect the right projection* $\textsf{proj}_2(ku_1)$ *of* ku_1*, i.e., the temperature safety proof* p_T*. If* p_T *validation fails, judge: the root cause lies in the input data of sensor 1. If* p_T *validation passes, judge: the root cause lies in the logical computation of the merge function combine.*

4.3 General Operators

4.3.1 State Projection Operators

The projection operators define the logic for extracting components from complex dependent pairs (Dependent Pairs).

- **Knowledge Extraction Operator (get_K)**
  - **Core Type**: $\textsf{get\_K} : \Pi(\sigma : \Sigma).\textsf{Set}(\textsf{Facts})$
  - **Operator Definition**: $\textsf{get\_K} \equiv \lambda\sigma.\textsf{proj}_1(\sigma)$
  - **Description**: This operator uses the first projection to extract the knowledge base $\mathcal{K}$. In the Core layer, it ensures that the returned set items are all well-formed type instances.
- **Clock Reading Operator (now)**
  - **Core Type**: $\textsf{now} : \Pi(\sigma : \Sigma).\textsf{Time}$
  - **Operator Definition**: $\textsf{now} \equiv \lambda\sigma.\textsf{proj}_1(\textsf{proj}_2(\sigma))$
  - **Description**: Extracts the middle item $\mathcal{T}$ of the triple. This operator is the foundation for all temporal logic judgments (e.g., "whether the contract has expired").

4.3.2 Intention Scheduling Operators

The scheduling operators manage the intent queue $\mathcal{P}$ through recursive list operations.

- **Intent Push Operator (schedule)**
  - **Core Type**: $\textsf{schedule} : \Pi(\sigma : \Sigma).\Pi(e : \textsf{Ev}).\Sigma$
  - **Operator Definition**: $\textsf{schedule} \equiv \lambda\sigma.\lambda e.\langle \textsf{get\_K}(\sigma), \textsf{now}(\sigma), \textsf{append}(\textsf{proj}_2(\textsf{proj}_2(\sigma)), e)\rangle$
  - **Description**: This operator constructs a new Σ instance. Its core is to append a quintuple conforming to the Ev type restrictions to the end of the queue.

4.3.3 Evolution Control Operators

This is the core driving the system's forward evolution, involving the fusion of computation and consistency judgment.

- **Clock Stepping Operator (tick)**
  - **Core Type**: $\textsf{tick} : \Pi(\sigma : \Sigma).\Sigma$

  - **Operator Definition**: $\mathsf{tick} \equiv \lambda\sigma.\langle \mathsf{get\_K}(\sigma), \mathsf{now}(\sigma) + 1, \mathsf{consume}(\sigma)\rangle$
  - **Description**: It not only increments the time count but is usually accompanied by the consumption of the current event, representing the completion of a logical step.
- **Knowledge Unification Operator (unify)**
  - **Core Type**: $\mathsf{unify} : \Pi(\sigma : \Sigma).\Pi(f : \mathsf{Fact}).\Sigma$
  - **Operator Definition**: $\mathsf{unify} \equiv \lambda\sigma.\lambda f.\mathsf{if}\ \mathsf{is\_consistent}(\mathsf{get\_K}(\sigma), f)$ then $\langle \mathsf{get\_K}(\sigma) \cup \{f\}, \mathsf{now}(\sigma), \dots\rangle$ else σ
  - **Description**: This is the most complex operator. Before merging a new fact, it uses the Core layer's judgment rules to verify the logical compatibility (Consistency) of f with the existing $\mathcal{K}$.

4.3.4 Causal & Trace Operators

Utilize the identity type (Identity Type) for deep auditing.

- **Identity Verification Operator (verify_id)**
  - **Core Type**: $\mathsf{verify\_id} : \Pi(\sigma_1 : \Sigma).\Pi(\sigma_2 : \Sigma).\mathsf{Type}$
  - **Operator Definition**: $\mathsf{verify\_id} \equiv \lambda\sigma_1.\lambda\sigma_2.\mathsf{Id}_\Sigma(\sigma_1, \sigma_2)$
  - **Description**: Returns a proposition type (Prop). To execute in the Kernel, a constructive proof (e.g., refl) must be provided to verify whether the two configurations are logically the same truth.

When the above Core layer operators are invoked by the Kernel layer, their execution follows the following reduction path:

(1) **Parameter Substitution (β-reduction)**: Substitute the current state instance of the Kernel layer (e.g., $\sigma_{current}$) into the λ-term of the operator.

(2) **Structural Unfolding (ι-reduction)**: The projection operator proj extracts specific components from the triple.

(3) **State Materialization**: The final term obtained from reduction (e.g., new Σ') is stored in the kernel storage, becoming the input for the next cycle.

Definition 18. *Termination of Operators*

Due to the Core layer's computational model based on strong normalization (Strong Normalization), all kernel general operators must terminate and produce results in a finite number of steps. This theoretically avoids "dead loops" in the kernel during state transitions.

4.4 Logical Properties of the Kernel Layer

In the KOS-TL kernel architecture, state preservation (Preservation), also often referred to as type preservation (Subject Reduction) in the state machine dimension, ensures that the logical "well-formedness" of the system does not collapse due to data inflow during environment evolution or transaction commit.

Theorem 7. *State Preservation*

Let Γ be the system global context. If the kernel state Σ is well-formed (denoted $\Gamma \vdash \Sigma\ \mathsf{ok}$), and there exists a transition step triggered by event e such that $\Sigma \xrightarrow{e} \Sigma'$, then the new state Σ' after the transition is still well-formed:

$$\Gamma \vdash \Sigma\ \mathsf{ok} \quad \wedge \quad \Sigma \xrightarrow{e} \Sigma' \implies \Gamma \vdash \Sigma'\ \mathsf{ok}$$

where well-formedness $\Sigma\ \mathsf{ok}$ is defined as: for all fact items $ku_i \in \Sigma$ contained in the state, there exists a type A_i such that $\Gamma \vdash ku_i : A_i$, and Σ satisfies consistency $\Sigma \nvdash \bot$.

Proof. We prove by structural induction on the transition operator according to the nature of event e.

1. Internal Computation Step: If e corresponds to an internal reduction in the kernel (e.g., expression simplification via β-reduction), then $\Sigma' = \Sigma$ and only the control item $ctrl$ changes.

- According to the **Subject Reduction** theorem of the Core layer: If $\Gamma, \Sigma \vdash ctrl : A$ and $ctrl \to ctrl'$, then $\Gamma, \Sigma \vdash ctrl' : A$.
- Since Σ itself does not change, its well-formedness $\Gamma \vdash \Sigma\ \mathsf{ok}$ is automatically preserved.

2. External Fact Injection: If e corresponds to injecting a new fact ku_{new} into the knowledge base, the transition is defined by $\mathsf{unify}(\Sigma, ku_{new})$.

- **Type Pre-check**: The premise of the transition is that ku_{new} must pass type checking, i.e., $\Gamma, \Sigma \vdash ku_{new} : A_{new}$.
- **Consistency Conflict Handling**:
  - *Branch A (Compatible)*: If $\Sigma \cup \{ku_{new}\} \nvdash \bot$, then $\Sigma' = \Sigma \cup \{ku_{new}\}$. By the weakening lemma (Weakening Lemma), the existing fact items remain well-typed in the larger context.
  - *Branch B (Conflict)*: If $\Sigma \cup \{ku_{new}\} \vdash \pi : \bot$, the kernel does not merge directly but constructs $\Sigma' = \Sigma \cup \{\mathsf{Invalidated}(ku_{new}, \pi)\}$.
- In both branches, Σ' does not contain directly derivable $\bot$, and all elements have corresponding constructive proofs. Thus, $\Gamma \vdash \Sigma'$ ok.

3. Environment Elimination Step: If e corresponds to the application of an elimination rule (e.g., extracting a projection item from a Σ-type fact).

- Assume $\langle a, p\rangle : \Sigma(x : A).B$ exists in Σ. The transition step produces $a : A$.
- According to the semantic soundness of the Σ-elimination rule, the projected item a has a predefined and valid type A.
- This operation is merely an unfolding of existing well-formed knowledge and does not introduce inconsistencies, so Σ' remains well-formed.

Conclusion: In summary, regardless of the transition event e, the new state Σ' maintains logical well-formedness and consistency. □

In the KOS-TL Kernel layer, determinism (Determinism) is the cornerstone ensuring distributed consensus (Consensus) and logical traceability. Under the dependent type system, determinism not only means computational result consistency but also confluence (Confluence) of reduction paths.

Theorem 8. *Determinism of Kernel Evolution*

Let Op be the kernel state transition function, Σ the current well-formed kernel state, and e the triggering event. If the transition rule is defined as $\Sigma' = \mathsf{Op}(\Sigma, e)$, then for the same input pair (Σ, e), the output new state Σ' is unique in the sense of logical equivalence:

$$\forall \Sigma, e, \Sigma'_1, \Sigma'_2 : (\Sigma \xrightarrow{e} \Sigma'_1 \wedge \Sigma \xrightarrow{e} \Sigma'_2) \implies \Sigma'_1 \equiv \Sigma'_2$$

where $\equiv$ denotes intensional equality (Intensional Equality), i.e., the normal forms (Normal Form) of the two are completely identical.

Proof. The proof is based on the pure function nature of KOS-TL Core and the confluence of the strong normalization calculus, unfolded in the following three dimensions:

1. Purity of Operators:

All transition operators in the kernel (such as unify, subst, eval) are defined as terms in the Core Layer.

- In Core layer theory, all constructors satisfy **computational consistency**. Given the same input ρ (assignment environment), the interpretation function $[\![\mathsf{Op}]\!]_\rho$ is a single-valued function in the mathematical sense.
- Since Op does not depend on any external implicit state or random source, its mapping relation $\Sigma \times e \to \Sigma'$ is deterministic under functional semantics.

2. Strong Normalization and Confluence:

Since KOS-TL has strong normalization (Strong Normalization) property, by the **Church-Rosser theorem**, the calculus system has confluence.

- Even if there are multiple optional reduction redexes during the reduction $\Sigma \xrightarrow{e} \Sigma'$, confluence guarantees that regardless of the reduction strategy (Reduction Strategy) taken, the ultimately obtained normal form $\mathsf{nf}(\Sigma')$ is unique.
- Therefore, although intermediate steps in physical memory may differ slightly, the state at the logical level (i.e., the set of facts that can participate in subsequent derivations) is unique.

3. Deterministic Conflict Resolution:

When handling consistency conflicts caused by e, the kernel's branch judgment logic:

- The unify operator performs exhaustive search according to the priority of typing rules (Typing Rule Priority).
- The construction of the contradiction proof term π follows standard search algorithms (such as the Unification Algorithm). Within the given search space, the first minimal proof term found is deterministic.
- The choice of Branch A or Branch B is completely determined by the logical truth value of "whether there exists a proof term π," without non-deterministic choice (Non-deterministic Choice).

Conclusion: In summary, due to the pure function definition of operators and the confluence of the underlying calculus system, the state transitions in the KOS-TL kernel have strict determinism. □

Progress is the core mathematical cornerstone ensuring the kernel's real-time responsiveness and robustness. It guarantees that the logical self-healing engine never gets stuck in a "computational dead end" at any moment.

Theorem 9. *Kernel Progress*

Let $\mathcal{C} = \langle \Sigma, \mathsf{Ev}, \Delta \rangle$ be a well-formed KOS-TL kernel configuration, where the state $\Sigma = \langle \mathcal{K}, \mathcal{TS}, \mathcal{P} \rangle$ and the transition record Δ satisfies temporal monotonicity. If $\mathcal{C}$ satisfies the global type assignment and the current active event Ev is well-formed under the context Σ, then one of the following must hold:

PP1. Logical Steady State (Logical Quiescence): *$\mathsf{Ev} = \mathsf{null}$ and the pending queue $\mathcal{P} = \emptyset$. At this point, all causal chains in the system have been materialized in Δ, and computation is temporarily terminated.*

PP2. Causal Progression (Causal Advancement): *There exists a new configuration $\mathcal{C}' = \langle \Sigma', \mathsf{Ev}', \Delta \cup \{\delta\} \rangle$ such that the system advances forward through one of the following reduction steps:*

- ***Execution Step (Execution):*** *If $\mathsf{Ev} = e \neq \mathsf{null}$, then execute Op to produce a new state Σ', and generate a transition record $\delta = \langle \Sigma \xrightarrow{e} \Sigma' \rangle$.*
- ***Activation Step (Activation):*** *If $\mathsf{Ev} = \mathsf{null}$ and $\mathcal{P} = e_0 :: \mathcal{P}_{rest}$, then activate the first event in the queue via the extraction operator.*

Proof. Classify and discuss based on the construction of the current active item Ev and the state of the queue $\mathcal{P}$:

1. Calculus of Active Events

When $\mathsf{Ev} = e \equiv \langle \mathsf{Args}, \mathsf{Pre}, \mathsf{Op}, \mathsf{Post}, \mathsf{Prf} \rangle$, due to the well-formed configuration, there exists a proof term Prf satisfying $\mathsf{Pre}(\mathsf{Args}, \Sigma)$. According to the strong normalization property of the Core Layer:

- Op as a total function (Total Function) must have a defined output value Σ' for the input pair (Args, Σ).
- According to the completeness of the elimination rules, the postcondition $\mathsf{Post}(\Sigma')$ is decidable after Σ' is constructed.

Therefore, executing the operator must produce a new transition item δ, thereby transitioning the system to PP2.

2. Queue Dynamics

If $\mathsf{Ev} = \mathsf{null}$, the system checks the pending queue $\mathcal{P}$:

- If $\mathcal{P} = e_0 :: \mathcal{P}_{rest}$, according to the kernel's operational semantics, there exists a well-defined "enqueue-dequeue" transformation that updates Ev to e_0. This step does not change $\mathcal{K}$ but alters the system's kinetic allocation.

- If $\mathcal{P} = \emptyset$, then the system satisfies the steady-state condition described in PP1.

3. Temporal Arrow Constraint

In all transitions $\Sigma \xrightarrow{e} \Sigma'$, the monotonicity rule $\Sigma'.\mathcal{TS} > \Sigma.\mathcal{TS}$ ensures the uniqueness of the transition item δ. Since the transition record set Δ is a monotonically increasing union, the system excludes the possibility of logical cycles (Cycles). According to the canonical forms lemma (Canonical Forms Lemma) of dependent type theory, under a well-formed Σ, no Op can produce type-mismatched pendings. In summary, a well-formed KOS-TL kernel configuration always has the ability to evolve to the next step until all events are cleared. □

Theorem 10. *Evolutionary Consistency*
Let the well-formed kernel configuration be $\mathcal{C} = \langle \Sigma, \mathsf{Ev}, \Delta \rangle$, where the state $\Sigma = \langle \mathcal{K}, \mathcal{TS}, \mathcal{P} \rangle$. If it satisfies:

(1) ***State Legitimacy****: For all $(id_i, t_i, A_i) \in \mathcal{K}$, there exists $\Gamma_{Core} \vdash t_i : A_i$ and $\mathcal{K} \not\vdash \bot$.*

(2) ***Causal Completeness****: The active event Ev carries a valid proof term $\mathsf{Prf} : \mathsf{Pre}(\mathsf{Args}, \Sigma)$.*

If the kernel executes a small-step evolution $\mathcal{C} \xrightarrow{step} \mathcal{C}'$, then the new configuration $\mathcal{C}' = \langle \Sigma', \mathsf{Ev}', \Delta \cup \{\delta\} \rangle$ still maintains state legitimacy and global logical consistency.

Proof. We unfold the proof by classifying and discussing the atomic nature of configuration transitions.

1. **Computational Reduction Step (Atomic Term Evolution)**

   If the reduction only involves $\mathsf{Ev} \to \mathsf{Ev}'$ (e.g., intensional simplification of proof terms or parameter substitution), while the knowledge base $\mathcal{K}$ remains unchanged:

   - **Consistency Inheritance**: Since $\mathcal{K}$ does not change and it is known that $\mathcal{K} \not\vdash \bot$, consistency is naturally preserved.
   - **Subject Reduction**: According to the metatheory of KOS-TL Core, reduction of dependent type terms preserves their types. Since Ev is well-formed under Σ, the reduced Ev' still satisfies the original type signature.

2. **State Transition Step (Core Evolution Operator)**

   When executing Op leads to $\mathcal{K} \to \mathcal{K}'$, the system executes the operator $\mathsf{commit}(\mathcal{K}, ku_{new})$. We analyze the consistency of the new knowledge item ku_{new} and its proof:

   Case A: Monotonic Expansion

   - **Application of Weakening Lemma**: If ku_{new} has no conflict with existing knowledge, by the weakening lemma (Weakening Lemma) of constructive logic, all proofs in the original $\mathcal{K}$ remain valid in $\mathcal{K}' = \mathcal{K} \cup \{ku_{new}\}$.
   - **Safety Closure**: Since Ev includes $\mathsf{Post} : \Sigma' \to \mathsf{Prop}$, the kernel forcibly checks the postcondition before materializing $\mathcal{K}'$. If the check passes, $\mathcal{K}' \not\vdash \bot$ is formally guaranteed.

   Case B: Conflict Mitigation

   - **Logical Isolation**: If ku_{new} introduces a logical contradiction (i.e., there exists $\pi : \mathcal{K}, ku_{new} \vdash \bot$), the kernel's protection mechanism prevents direct merging.
   - **Negation Introduction Construction**: The kernel instead constructs $\mathsf{absurd}(ku_{new}, \pi)$ and stores it in the knowledge base. In the semantic model, this equates to transforming the conflict into a falsifying conclusion for the input signal. Since the contradiction is wrapped in a negation constructor, it cannot serve as a premise for elimination rules, thereby protecting global consistency.

3. **Constraints of Temporal Arrow and Transition Records** To prove that the evolution trajectory is legitimate, the kernel constructs causal evidence δ using Δ:

   - **Proof of Clock Monotonicity**: Every evolution is accompanied by $\mathcal{TS}' > \mathcal{TS}$. This proves that Σ' is not a simple loop of Σ but a logical monotonic successor.

- **Materialization Induction**: The legitimacy of the system state Σ_n can be traced back to the initial empty state Σ_0 by induction:

$$\Sigma_n = \mathsf{Apply}(\delta_n, \mathsf{Apply}(\delta_{n-1}, \dots \Sigma_0))$$

  Each $\delta_i = \langle \Sigma_{i-1} \xrightarrow{e_i} \Sigma_i \rangle$ includes verification of Pre and satisfaction of Post, ensuring that every link in the evolution chain aligns with the logical base.

□

This proof explains how KOS-TL handles "dirty data" in the real world (such as erroneous bank transactions or sensor false alarms): Logical Firewall: Evolutionary Consistency ensures that any data attempting to disrupt system consistency is converted into "evidence about contradictions" during the unify phase, rather than allowing the system itself to become contradictory. Bidirectional Synchronization Safety: In the "global supply chain" example you mentioned earlier, even if the underlying database is illegally tampered with (producing conflicting data), Evolutionary Consistency will force the kernel to generate a Refute item, thereby maintaining the logical purity at the ontology view layer.

These two major properties jointly define the operational boundaries of the KOS-TL Kernel: Progress guarantees livelock freedom (Livelock Freedom) for the kernel. As long as the logic is well-formed, the kernel analysis program will always proceed and produce analysis results. Evolutionary Consistency guarantees runtime safety (Runtime Safety) for the kernel. It ensures that the kernel knowledge base, during dynamic runtime processes, never degenerates into a self-contradictory abandoned system.

Theorem 11. *Local Decidability Theorem*

Given the kernel state Σ, new fact ku_{new}, and search boundary $\Delta = \{depth, fuel\}$, the execution process of the kernel operator $\mathsf{unify}(\Sigma, ku_{new}, \Delta)$ is decidable.

Proof. The proof is completed by double induction on the search space and the number of reduction steps:

1. Finiteness of Search Space:

Due to the kernel limiting $depth$ (recursion depth), the proof search tree is forcibly pruned to finite height. At each level, the number of unification candidates is determined by the number of variables in the context Γ, which is also finite.

2. Forced Termination via Fuel:

Introduce the $fuel$ parameter (computational energy). Each execution of a β-reduction or δ-unfolding consumes one unit of $fuel$.

- The algorithm checks $fuel > 0$ before each operation.
- Since $fuel$ is a natural number and strictly decreases with the number of steps, by the well-ordering principle, the computation must either reach a normal form or exhaust $fuel$ in a finite number of steps.

3. Completeness of Result Set:

When the computation stops:

- If the normal form matches, return **True**.
- If a structural conflict (e.g., constructor mismatch) is found, return **False**.
- If stopped due to exhaustion of $depth$ or $fuel$, return **Unknown**.

Since the algorithm guarantees termination on all paths, the process is decidable. □

4.5 Application Example: Quality Anomaly Tracing and Derivation

In manufacturing scenarios, when a batch defect is detected, the kernel layer automatically triggers tracing logic.

- **Event Definition**: Let e_{trace} be the tracing event.
  - Pre: There exists a defect report item $r : \mathsf{DefectReport}$ in the state.

  - Op: Based on production logs, search for equipment anomaly records $s : \mathsf{EquipmentStatus}$ associated with the batch.
  - Post: Generate and materialize a causal chain knowledge object $cc : \mathsf{CausalChain}$.
- **Evolution Process**: Once the Runtime refines the defect report and stores it in Σ, the kernel layer discovers that $p : \mathsf{Pre}$ holds via the above equation and automatically executes Op. The system takes a small step from the state "known defect exists" to the higher-entropy state "known defect cause."

Example 4. *Cross-Border Compliance Transfer Event ($e_{transfer}$)*

Assume the current system state is σ, containing the balances of accounts A and B. We want to define an event for transferring amount v from A to B.

1. State Definition ($\sigma \in \Sigma$)

The state σ is a knowledge snapshot containing: $Balance(A, \sigma) = 1000$ $Balance(B, \sigma) = 500$

2. Specific Event Construction ($e_{transfer} : \mathsf{Ev}$)

According to your Ev definition, the event consists of three parts: Precondition (pre):

$$\mathsf{pre} \equiv (Balance(A, \sigma) \geq v) \wedge \mathsf{IsVerified}(A)$$

(Explanation: Account A's balance must be sufficient, and A must have passed real-name verification.)

Action Operator (act):

$$\mathsf{act}(\sigma) \equiv \sigma[Balance(A) \leftarrow Balance(A) - v, Balance(B) \leftarrow Balance(B) + v]$$

(Explanation: This is a function describing how the state changes: subtract money from A, add money to B.)

Post-Transformation Self-Proof ($\mathsf{post\_prf}$): This is a proof term guaranteeing that: For any state σ satisfying pre, the new state after executing act must satisfy the conservation law ($Sum_{after} = Sum_{before}$).

$$\mathsf{post\_prf} : \Pi(\sigma : \Sigma).\mathsf{pre} \to \mathsf{Correct}(\mathsf{act}(\sigma))$$

3. Execution Process: Small-Step Transition (Δ)

When the kernel attempts to execute this transfer, the following judgment process occurs: Type Checking: The kernel first verifies $\Gamma \vdash e_{transfer} : \mathsf{Event}$. If the developer's act logic is flawed (e.g., subtract money but not add it), then $\mathsf{post\_prf}$ cannot be constructed, and the event will be rejected at the compilation stage.

Trigger Transition: Input the current snapshot σ and event e.

$$\mathsf{STEP}(\sigma, e_{transfer}) \to \sigma'$$

Generate Transition Record (Δ): Produce a triple ($\sigma \xrightarrow{e_{transfer}} \sigma'$). This record is permanently stored in the evolution trajectory Δ.

5 KOS-TL Runtime Layer: Environment Interaction and Signal Refinement

The Runtime Layer is the boundary between the KOS-TL logical system and the physical world. It is responsible for handling non-deterministic external signals, managing computational resources, scheduling event queues, and persisting the logical states generated by the kernel layer into physical storage.

5.1 Syntax

The Runtime Layer state is described by a configuration Cfg, which embeds the logical kernel into the physical host:

$$Cfg \equiv \langle \Sigma, Q_{raw}, \mathsf{Env}, \mathcal{M} \rangle \tag{2}$$

where:

- Σ — Logical Kernel State
  This is the current form of the system's "brain" at the logical level. It contains the knowledge base $\mathcal{K}$, logical clock $\mathcal{T}$, and pending intent queue $\mathcal{P}$. It represents all logical facts that the system currently considers "true" and "proven." The Runtime Layer observes Σ to decide what actions to perform externally next or how to respond to external signals.

- Q_{raw} — Raw Physical Signal Queue

  This is the system's "perception input buffer." It stores raw binary or text data from the external physical world (such as sensors, network packets, user clicks) that has not yet been logicalized. The arrival of signals is random. The signals here do not yet have corresponding logical proofs (Proof); they are just "dirty data." They are the raw material for the elab (refinement operator). The system retrieves signals from Q_{raw} and attempts to promote them to events understandable by the kernel.

- Env — Physical Runtime Environment

  This is the physical host context in which the system resides. It includes physical resources that the logical layer cannot directly perceive but that the Runtime Layer must manage: (1) Physical Clock (T_{wall}): Similar to wall-clock time in the real world, used for timeout judgments. (2) I/O Handles: Database connection pools, network sockets, hardware register addresses. (3) Computational Resources: Memory state, thread pool load, etc. During execution of elab, Env provides the physical evidence needed to construct logical proofs (e.g., sensor self-test success status bits).

- $\mathcal{M}$ — Materialization Mapping and Storage

  $$\mathcal{M} : \mathcal{K} \to \mathsf{PhysicalStorage}$$

  This is the system's "memory" and "physical projection." $\mathcal{M}$ represents both physical storage (e.g., databases on disk) and the mapping rules from logical items to physical representations. Its roles include: (1) Projection: Mapping abstract "dependent pair knowledge" from the kernel layer to "rows, columns, indexes" in the database. (2) Persistence: Ensuring that logical evolution results in Σ are safely written to non-volatile storage. (3) External Consistency: Ensuring that the state seen in the physical world (e.g., balance displayed on the screen) remains synchronized with the logical kernel Σ.

Scheduler Main Loop Logic (reflected in driving kernel operators):

```
while Q_raw is not empty:
  s = pop(Q_raw)
  match elab(s, Env):
    case (e):
      // Invoke the kernel layer's defined transition rule
      if <Sigma, e> -->_KOS Sigma':
        Sigma = Sigma'
        Commit_to_Storage(M, Sigma)
    case None:
      Log_Refinement_Failure(s)
```

The Runtime Layer introduces the elaborator (Elaborator), whose syntactic function is to convert "dirty data" into "constructor terms" understandable by the Core layer:

$$\mathsf{elab} : \mathsf{RawSignal} \to \mathsf{Env} \to \mathsf{Option}\left(\sum_{e:\mathsf{Ev}} \mathsf{Pre}(e, \Sigma)\right)$$

The elaborator is the gateway for realizing "signal logicalization." Its core task is to supplement external data with proof terms:

$$\mathsf{elab}(s, \mathsf{Env}) = \begin{cases} \mathsf{Some}(\langle e, \pi \rangle) & \text{if } \pi : \mathsf{Pre}(e, \Sigma) \text{ can be constructed} \\ \mathsf{None} & \text{otherwise} \end{cases} \tag{3}$$

The refinement process includes:

- **Signal Parsing**: Parse external JSON/binary streams into base sort values ($\mathsf{Val}, \mathsf{ID}$).
- **Proof Construction**: Automatically attempt to construct logical proof terms p for preconditions based on the current Env.
- **Time Anchoring**: Map physical reception time to the Core layer's Time type.

5.2 Runtime Semantics

The evolution of the Runtime Layer is presented as an "asynchronous-driven small-step transition", with its core rule being the "refinement-commit" loop. The semantic rules of the Runtime Layer must include updates to the external environment and persistence of storage:

$$\frac{s = \mathsf{head}(Q_{raw}) \quad \mathsf{elab}(s, \mathsf{Env}) = \mathsf{Some}(\langle e, \pi \rangle) \quad \langle \Sigma, e, \pi \rangle \longrightarrow_{KOS} \Sigma' \quad \mathsf{Persist}(\Sigma', \mathcal{M}) = \mathsf{Success}}{\langle \Sigma, s :: Q, \mathsf{Env}, \mathcal{M} \rangle \xrightarrow{\mathsf{commit}} \langle \Sigma', Q, \mathsf{Env}', \mathcal{M}' \rangle} \tag{4}$$

Here, $\mathcal{M} \vdash \Sigma' \Downarrow \mathcal{M}'$ indicates that the new state Σ' is successfully “downcast” (Down-cast) and materialized into the physical medium $\mathcal{M}'$.

5.3 Logical Properties

In the Runtime layer, we model each execution action (Action) as a pair $e = (t, p)$, where t is the target proposition (task), and p is its corresponding proof term.

Definition 19. *Causal Dependency Order*

Let $\mathcal{E}$ be the set of all possible execution items in the system. Define the causal dependency relation $\prec_L \subseteq \mathcal{E} \times \mathcal{E}$: If in the Core layer, the construction item of proposition t_2 contains a reference to t_1 (i.e., t_1 is a premise of t_2), then call $e_1 \prec_L e_2$.

Definition 20. *Runtime Execution Sequence*

The execution sequence $S = [e_1, e_2, \ldots, e_n]$ is a total order set, representing the physical time order in which the Runtime layer actually processes data.

Theorem 12. *Causal Ordering Consistency*

For any Runtime execution sequence S, if the sequence is accepted by the kernel (Accepted), then for any two execution items e_i and e_j in S, it must hold that:

If $e_i \prec_L e_j$, then in the sequence S, e_i must precede e_j in completion of reduction.

If the physical network causes e_j to arrive before e_i, the Runtime must block the execution of e_j until the proof term for e_i is completed.

Proof. We prove by contradiction (Proof by Contradiction) combined with the Core layer’s type checking mechanism.

Step 1: Assume an Out-of-Order Execution Exists.

Assume the Runtime accepts a sequence S' that violates the causal order, where there exists e_j completing execution before e_i, and it is known that $e_i \prec_L e_j$.

Step 2: Core Layer Constraint Mapping.

According to the definition of $\prec_L$, in the Core layer, the proof term validation of e_j depends on the existence of e_i. Its type checking rule is as follows:

$$\frac{\Gamma \vdash p_i : T_i \quad \Gamma, x : T_i \vdash p_j : T_j}{\Gamma \vdash \langle p_i, p_j \rangle : \Sigma(x : T_i).T_j}$$

This means that to judge e_j as legitimate, the kernel must include the proof term for e_i in the context Γ.

Step 3: Runtime State Evolution.

The Runtime’s state is represented by the context sequence Γ_t. When executing e_j, its operator is $\mathsf{check}(\Gamma_{current}, e_j)$.

- If e_i has not been executed, then $e_i \notin \Gamma_{current}$.
- At this point, according to the Core layer’s **Scope Determinism**, the reference to e_i in e_j will produce an “undefined variable” error or “free variable” escape.
- The type checker will return Fail.

Step 4: Inevitability of the Blocking Mechanism.

Due to KOS-TL's Runtime enforcing a **Type-Safe Fence**, any failed validation operation cannot change the state of the Σ fact base. To continue execution, the Runtime scheduler must suspend e_j, place it in the pending pool (Pending Pool), and issue a $\mathsf{Requirement}(e_i)$ signal.

Step 5: Conclusion.

Only when e_i arrives and successfully reduces into Γ does the context for e_j satisfy the validation condition. Therefore, the final "accepted" sequence must satisfy the causal order. □

Theorem 13. *Refinement Fidelity*

Let $\mathcal{S}$ be the physical hardware state space (e.g., FPGA registers or sensor reading sets), and $\mathcal{D}_{Core}$ the logical domain. Define the refinement function $\mathcal{E} : \mathcal{S} \to \mathcal{D}_{Core}$. If the Runtime captures physical state $s \in \mathcal{S}$ to obtain $ku = \mathcal{E}(s)$, then it satisfies:

(1) ***Well-formedness****: There exists a type $A \in \mathcal{U}$ such that $\Gamma \vdash ku : A$ always holds.*

(2) ***Simulation Consistency****: For physical migration $s \xrightarrow{hw} s'$, there exists a simulation relation $R \subseteq \mathcal{S} \times \mathcal{D}_{Core}$ such that:*

$$(s, ku) \in R \implies \exists ku'.(s', ku') \in R \land (ku \xrightarrow{small}^{*} ku' \lor \mathit{Invalidated}(ku'))$$

Proof. We prove by constructing a simulation relation and combining it with the reduction of the hardware abstraction layer (HAL):

1. Construct Simulation Relation R:

Define the relation R as follows:

$$(s, ku) \in R \iff (\mathsf{val}(ku) = \mathsf{measure}(s)) \land (\mathsf{proof}(ku) \models \mathsf{Inv}_{HW}(s))$$

where $\mathsf{measure}(s)$ is the quantification of the physical signal, and $\mathsf{Inv}_{HW}(s)$ is the physical invariant enforced by hardware circuits (such as redundant check bits or watchdog states).

2. Well-formedness Mapping Proof:

According to the runtime refinement rules of TL-Lang, the construction of $\mathcal{E}(s)$ is:

$$\mathcal{E}(s) \triangleq \langle \mathsf{quantize}(s), \mathsf{synthesize\_witness}(s) \rangle$$

Since $\mathsf{synthesize\_witness}$ is a deterministic operator defined by hardware description language (HDL), it directly maps hardware register states Reg_{status} to introduction items (Introduction Rules) in the Core layer. According to the construction principles of Σ types, as long as the hardware signal is within the physical range, a well-formed item ku can always be constructed. If the signal exceeds the range, the refinement function, by completeness, maps to a predefined error type item, still maintaining well-formedness.

3. Simulation Consistency Proof:

Classify the physical state migration $s \xrightarrow{hw} s'$:

- **Case A: Compliant Migration.**

  If s' satisfies all hardware safety constraints, the refinement function $\mathcal{E}$ extracts new status bits and synthesizes new proof terms p'. Since the hardware layer guarantees s' comes from s via legitimate logic gates, in the Core layer, the corresponding mapping item ku' must evolve from ku through kernel reduction steps (such as β or ι reduction), maintaining the simulation relation.

- **Case B: Illegal/Anomalous Migration.**

  If the physical migration violates Inv_{HW} (e.g., sensor disconnection), the hardware status bit flips. At this point, the refinement mapping $\mathcal{E}(s')$ cannot construct an item of the original type A, instead constructing $\mathsf{Invalidated}(ku')$. This shift from "normal item" to "invalid item" manifests in the Kernel layer as a non-monotonic flip of conclusions, conforming to KOS-TL's semantics for handling conflicts, and the simulation relation is still maintained in the "error handling" dimension.

In summary, the refinement process ensures that any valid change in the physical world can find a corresponding truth representation in the logical world. □

Theorem 14. *Observational Adequacy*

Let ctrl $\in \mathcal{D}_{Core}$ *be the logical control item generated by the kernel, with type instruction set* Cmd*. Let* $\mathcal{G} : \mathsf{Cmd} \to \Pi^*$ *be the instruction generator, mapping logical items to hardware instruction sequences* π*. If* $\Gamma \vdash ctrl : \mathsf{Cmd}$ *and the logical layer asserts that ctrl satisfies property* ϕ*, then:*

$$\forall s \in \mathcal{S}, \quad (ctrl \vdash \phi) \implies (\mathsf{Exec}(\mathcal{G}(ctrl), s) \models \mathsf{Refine}^{-1}(\phi))$$

where $\mathsf{Refine}^{-1}(\phi)$ *is the predicate interpretation of the logical property* ϕ *in the physical state space* $\mathcal{S}$*.*

Proof. We prove using a combination of Hoare Logic and Refinement Calculus:

1. Construct Mapping Relation:

Define the mapping $\mathcal{M} : \mathsf{Prop} \to \mathcal{P}(\mathcal{S})$ between logical predicates ϕ and physical state predicates P. According to the inverse mapping of Refinement Fidelity, if the semantic goal of *ctrl* is to bring the system into state ϕ, then the corresponding underlying register state must satisfy $P = \mathsf{Refine}^{-1}(\phi)$.

2. Backward Derivation:

Induct on the construction of the Cmd type:

- **Atomic Operations (Atomic Instructions):**

  If *ctrl* is an atomic operation (e.g., $\mathsf{SetValve(open)}$), it is mapped to a specific machine code sequence π_a in the underlying reduction of TL-Lang. According to the Hoare triple definition of the hardware abstraction layer (HAL):

  $$\{s \in \mathcal{S}\}\, \pi_a\, \{s' \in \mathsf{Refine}^{-1}(\phi)\}$$

  Since $\mathcal{G}$ has passed static verification based on HAL axioms during construction, the correctness of instruction generation is guaranteed by HAL's completeness.

- **Composite Operations (Sequences and Branches):**

  If *ctrl* is composed of multiple sub-items, according to Hoare logic composition rules: If $\{P\}\, \pi_1\, \{Q\}$ and $\{Q\}\, \pi_2\, \{R\}$, then $\{P\}\, \pi_1; \pi_2\, \{R\}$. Since the Cmd type satisfies **strong normalization** in the Core layer, the generated instruction sequence π has finite length and deterministic paths, with no undefined side effects in the logical layer.

3. Atomicity and Interference Analysis:

During physical execution, if an interrupt occurs, the Runtime must maintain observational consistency. KOS-TL's Runtime adopts a transactional I/O mechanism. Each group of π generated by $\mathcal{G}(ctrl)$ is wrapped in a logical atomic block. According to the kernel's "progress" proof, the sequence either fully executes and achieves $s' \models \mathsf{Refine}^{-1}(\phi)$, or rolls back on failure and submits an $\mathsf{Invalidated}$ proof term to the kernel. Under this mechanism, there is no intermediate ambiguous state where "instructions executed but did not achieve the goal."

Conclusion: The semantic goal ϕ of the logical layer can be losslessly projected onto the physical state space. □

Property Discussion: Bridging the "Hardware-Software Gap" Observational Adequacy addresses the defense against "instruction drift" in practical high-security scenarios by eliminating semantic gaps: In traditional C/C++ development, compiler optimizations or driver errors may cause execution effects to deviate from design intentions (e.g., race conditions due to instruction reordering). In KOS-TL, since the instruction generator $\mathcal{G}$ is formally proven, this divergence between "intention and behavior" is logically eliminated. Verifiable Physical Effects: If a financial account is logically frozen, Observational Adequacy ensures that the corresponding record in the underlying database is also locked, with the operation guaranteed to be atomic.

We define the system's state space as $\mathcal{S}$, and decompose the system state into two views: Logical View ($\mathcal{S}_L$): The type context Γ and reduced fact base Σ in kernel memory. Physical View ($\mathcal{S}_P$): The persisted bitstream in storage media (disk or solid-state storage). Define the mapping function $\mathsf{Encode} : \mathcal{S}_L \to \mathcal{S}_P$, which converts logical proof terms into physical storage formats.

Theorem 15. *Durability Atomicity and Visibility*

Let the system execute a state transition $\delta : \mathcal{S}_L \to \mathcal{S}'_L$ *at time t. The Runtime layer guarantees the existence of an atomic operator* Commit*, satisfying:*

(1) ***Atomicity****: The logical acknowledgment of $\mathcal{S}'_L$ holds if and only if $\mathsf{Encode}(\mathcal{S}'_L)$ is fully persisted in $\mathcal{S}_P$.*

(2) ***Visibility****: For any subsequent read operation* $\mathsf{Recover}$*, if* Commit *has succeeded, then necessarily* $\mathsf{Recover}(\mathcal{S}_P) \equiv \mathcal{S}'_L$.

That is: There does not exist a state where the proof is logically established, but lost after physical restart.

Proof. We prove by constructing a "logic-physical sync lock" and idempotent reduction mechanism. First, we define the system's state space as $\mathcal{S}$, and decompose the system state into two views:

- Logical View ($\mathcal{S}_L$): The type context Γ and reduced fact base Σ in kernel memory.
- Physical View ($\mathcal{S}_P$): The persisted bitstream in storage media (disk or solid-state storage).

Define the mapping function $\mathsf{Encode} : \mathcal{S}_L \to \mathcal{S}_P$, which converts logical proof terms into physical storage formats.

Step 1: Construct Persistent Serialization of Proof Terms. Every logical change $\Delta\Sigma$ in KOS-TL is evidence with an Id type. Let $\Delta\Sigma = (p : T)$. The persistence process is modeled as a dependent pair: $\mathsf{Record} \equiv \Sigma(p : T).\mathsf{Persist}(p)$ where $\mathsf{Persist}(p)$ is a hardware-level primitive that only returns a witness upon physical write completion.

Step 2: Prove Atomicity. The Runtime maintains a write-ahead log (WAL) mechanism, whose entries are themselves items in the Core layer.

- If the system crashes during $\mathsf{Write}(\mathcal{S}_P)$, since $\mathsf{Persist}(p)$ has not yet generated a valid witness, according to the Core layer's **confluence**, the recovery operator $\mathsf{Recover}$ upon restart will discover that the transaction does not satisfy the completeness of the Σ type, automatically rolling back.
- Only when the physical layer returns p_{stored} does the logical layer update Γ to $\Gamma \cup \{p\}$.

Step 3: Prove Visibility (Consistent Recovery). Assume the system restarts. Due to the Core layer's **strong normalization** property:

- Every proof term p stored in $\mathcal{S}_P$ is self-contained and already reduced.
- The $\mathsf{Recover}$ operator reconstructs the logical view by re-executing type checks $\mathsf{check}(\Gamma, p, T)$.
- Since the Core layer is decidable and reduction paths are protected by confluence, the recovered logical state $\mathcal{S}^{rec}_L$ is necessarily logically equivalent to the last valid Commit state $\mathcal{S}'_L$ before the crash ($\mathcal{S}^{rec}_L \equiv \mathcal{S}'_L$).

Conclusion: Atomic physical writes guarantee the irrevocability of logical states, while the logical layer's strong normalization ensures that physical data, when reloaded at any moment, produces a unique and deterministic logical interpretation. □

Definition 21. *Semi-decidability*

A set $S \subseteq \mathbb{N}$ (or a proposition language L) is called ***semi-decidable*** *if there exists a Turing machine (or algorithm) M such that for any input x:*

- *If $x \in S$, then $M(x)$ halts and accepts;*
- *If $x \notin S$, then $M(x)$ either halts and rejects or runs forever (halting problem unknown).*

Theorem 16. *Semi-decidability of Proof Search*

Let Γ be a finite context and P a proposition. The problem of determining "whether there exists a proof term p such that $\Gamma \vdash p : P$" is semi-decidable.

Proof. In a system like KOS-TL that includes dependent types and higher-order logic, without $Fuel$ restrictions, the proof search problem has semi-decidability. We prove this theorem by constructing a universal enumerator (Enumerator).

Step 1: Enumerability of Proof Terms.

All well-formed proof terms in KOS-TL Core layer are generated by a finite set of syntactic rules (such as λ-abstraction, application, pair construction, etc.). We can enumerate all possible proof terms in dictionary order by term length (or structural complexity), denoted as the sequence $\{p_1, p_2, p_3, \dots\}$.

Step 2: Construct the Decision Algorithm $\mathcal{A}$.

For the given proposition P and context Γ, algorithm $\mathcal{A}$ performs the following steps:

1. Start a loop, sequentially retrieving one proof term p_i.
2. Invoke the Core layer's type checker to verify $\mathsf{check}(\Gamma, p_i, P)$. Due to the Core layer's **strong normalization property**, this step necessarily returns $True$ or $False$ in finite time.
3. If it returns $True$, algorithm $\mathcal{A}$ halts and outputs "P is provable."
4. If it returns $False$, continue the loop and check the next item p_{i+1}.

Step 3: Analyze Halting Behavior.

- **Case One: P is indeed provable.** Then there necessarily exists some proof term q satisfying the condition. Since our enumeration is complete, in finite steps, we will encounter $p_k = q$, at which point the algorithm halts.
- **Case Two: P is unprovable.** The algorithm will forever enumerate and check new items in the loop, never halting.

Conclusion:

Algorithm $\mathcal{A}$ can recognize all "true" propositions (provable propositions) but cannot guarantee halting for "false" propositions (unprovable propositions). By definition, the problem is semi-decidable. □

Definition 22. *Some Definitions for Decidability Proofs*

- *Proposition Space P: All well-formed KOS-TL Core propositions.*
- *Proof Algorithm $\mathcal{A}$: The automated process in the Runtime layer attempting to find a proposition $p : P$.*
- *Resource Vector $\vec{\Delta} = \langle f, d, \tau \rangle$:*
  - *$f \in \mathbb{N}$ (Fuel): Maximum number of β-reduction steps.*
  - *$d \in \mathbb{N}$ (Depth): Maximum recursion depth for search.*
  - *$\tau \in \mathbb{R}^+$ (Timeout): Physical wall-clock time limit.*

Theorem 17. *Bounded Decidability of KOS-TL Runtime*

Let P be a logical proposition to be decided by the Runtime. There exists a decision procedure $\mathcal{R}(P, \vec{\Delta})$ such that for any P and finite $\vec{\Delta}$, $\mathcal{R}$ necessarily halts in finite time, and its output space is: $\mathcal{O} = \{\mathsf{True}, \mathsf{False}, \mathsf{Unknown}\}$ where $\mathsf{Unknown}$ is the deterministic "resource exhaustion" state.

Proof. The proof is completed by structural induction on the number of execution steps k and the monotonicity of the measure function.

Step 1: Definition of the Measure Function.

Define the measure function $\mu(\sigma)$ for the current execution snapshot σ:

$$\mu(\sigma) = \langle \text{fuel}, \text{depth}, \text{remaining\_time} \rangle$$

In every logical execution step (an atomic state transition $\sigma \to \sigma'$), this measure function strictly decreases in lexicographical order:

$$\mu(\sigma') <_{lex} \mu(\sigma)$$

Step 2: Completeness Classification of State Transitions.

For the Runtime's single-step actions, its logic has only three possibilities:

1. **Logical Termination**: Find a proof p or conflict π. At this point, the algorithm directly returns True or False.
2. **Continued Reduction**: Resources are not exhausted ($\mu > 0$). The algorithm enters σ_{k+1}, and since μ is well-founded, this path cannot extend infinitely.

3. **Boundary Hit**: Any component of $\mu(\sigma)$ reaches zero. The algorithm immediately stops and returns $\mathsf{Unknown}$.

Step 3: Termination Proof.

Since the range of $\mu(\sigma)$ is a finite set of natural numbers (or bounded real interval), by the **well-ordering principle**, any strictly decreasing sequence must reach a minimum value in finite steps. In KOS-TL Runtime, the minimum value corresponds to the output set $\mathcal{O}$.

Step 4: Decidability Verification.

Decidability means the algorithm halts on all inputs. Since:

- Every atomic reduction step is decidable as guaranteed by the Core layer;
- The total number of steps is forcibly limited by $\vec{\Delta}$.

Thus, the Runtime no longer has the "infinite search" feature of semi-decidability, and the program becomes a total function over input propositions and resource boundaries. □

5.4 Application Example: Causal Repair of Out-of-Order Logs

In manufacturing scenarios, if the equipment anomaly signal s_{ES} arrives later than the quality inspection signal s_{QI} due to delay:

- **Refinement Blocking**: When s_{QI} arrives, the refinement operator finds it cannot construct the proof p for "existing equipment anomaly," and the event is placed in the pending queue by Runtime.
- **Evidence Completion**: After s_{ES} arrives, the runtime layer updates Σ. At this point, the scheduler detects the environment change and re-triggers the refinement of s_{QI}.
- **Logical Materialization**: The originally broken causal chain is completed by the kernel layer with atomic transition after logical evidence is supplemented, and finally, the runtime layer inserts the tracing conclusion in the physical database.

Example 5. *Industrial Sensor-Triggered Safety Shutdown*

1. Configuration State

The current runtime state $\langle \sigma, Q, \mathsf{Env} \rangle$ is as follows: σ (logical snapshot): Equipment state is Running, temperature threshold is $80°C$. Q (event queue): $[\dots]$ (currently empty). Env (external environment): Connected to a Modbus protocol temperature sensor.

2. External Stream and inject

The sensor sends a raw bitstream to the system: s (Raw Signal): 0x4A 0x02 (representing temperature reading of $82°C$). Action: inject(s, Q) pushes this hexadecimal signal into the pending queue.

3. Refinement Process: elaborate(s)

The Runtime layer attempts to convert this "meaningless" number into a "semantic event" recognized by the Kernel layer: Refinement Logic: elaborate looks up configuration rules and discovers 0x4A is a temperature alarm. Mapping Result: Maps to L1 layer event e_{stop}. $e_{stop}.\mathsf{pre}$: Current state must be Running. $e_{stop}.\mathsf{act}$: Change state to Stopped. $e_{stop}.\mathsf{post\_prf}$: Proof that this operation complies with the "over-temperature forced protection axiom".

4. Scheduling and Judgment

According to the scheduling algorithm you provided, the system executes as follows: Pop: Retrieve s from Q. Elaborate: s successfully refines to e_{stop}. Kernel_Check: Runtime calls the Kernel layer judgment $\Gamma \vdash e_{stop} : \mathsf{Event}$. Verification Passed: The event carries the correct $\mathsf{post\_prf}$ (shutting down at $82°C$ complies with the safety definition). Step: Logical state update: $\sigma_{new} = \mathsf{STEP}(\sigma, e_{stop})$.

5. Persistence: commit and Materialize

Action: commit(σ_{new}). Materialized Storage $\mathcal{M}$: Write the updated state to the physical database (e.g., PostgreSQL) and trigger the physical hardware relay to disconnect the current.

6 KOS-TL System

Integrating the kernel layer, core layer, and runtime layer of KOS-TL forms KOS-TL (Knowledge Operating System - Type Logic), also known as "Knowledge-Action Logic." Knowledge-Action Logic is a complete logical system based on intuitionistic dependent type theory integrated with small-step operational semantics. Through a layered architecture, it unifies static constraints on knowledge, dynamic evolution, and environmental refinement.

6.1 Overall Architecture

The syntax of KOS-TL consists of three nested layers of expressions, covering the full spectrum from abstract types to physical configurations.

6.1.1 Core Layer: Type Definition and Logical Foundation (The Denotational Foundation)

The Core layer is the system's "brain," mapping domain ontologies to dependent type theory. Ontology Integration: Define domain axioms as base types (Base Types) and predicates. Verification Mechanism: Type checker based on BHK interpretation, ensuring every $t : A$ is a valid knowledge construction. Responsibilities: Provide static constraints. It specifies what the system "can understand" and "what is truth."

6.1.2 Kernel Layer: Dynamic Evolution and Intent Scheduling (The Operational Engine)

The Kernel layer is the system's "heart," responsible for controlled state migrations. State Model: Maintain the triple $\sigma = \langle \mathcal{K}, \mathcal{T}, \mathcal{P} \rangle$ (knowledge, time, intents). Evolution Mechanism: Execute small-step operational semantics (Small-step Semantics). It invokes the Core layer's judgment capabilities to verify each state jump. Responsibilities: Provide dynamic consistency. It specifies how the system "evolves from the current truth to the next truth."

6.1.3 Runtime Layer: Environment Refinement and Physical Execution (The Physical Interface)

The Runtime layer is the system's "senses and limbs," handling boundary interactions with the physical world. Refinement: Elevate fuzzy physical signals (Signals) to proof terms recognized by the Core layer via the elab operator. Materialization: Degrade logical conclusions to persistent storage or hardware instructions via the $\mathcal{M}$ mapping. Responsibilities: Provide fidelity. It specifies how logical instructions reliably act on physical entities.

6.1.4 Architecture Global Invariant

The Grand Map of KOS-TL reveals a core law:

$$\forall \text{ physical change } \delta \in \mathcal{M}, \quad \exists \text{ logical proof } p \in \mathsf{Core} \quad \text{s.t.} \quad \mathsf{TypeCheck}(p, \mathsf{Ontology}) = \mathsf{Pass}$$

6.2 Global Interaction Protocol

This protocol describes how a physical pulse traverses the four-layer architecture and ultimately solidifies into a globally accepted truth.

6.2.1 Phase I: Refinement and Injection

(1) Triggering Party Runtime Layer (External Environment)

(2) Action

- The physical sensor generates a raw signal $s \in Q_{raw}$.
- Runtime invokes the core operator $\mathsf{elab}(s, \mathsf{Env})$.
- Cross-layer Interaction: elab references predicate templates defined in the Ontology layer and constructs a dependent pair proof term $p : \mathsf{Pre}(e, \Sigma)$ in the Core layer.
- Result: Generates a valid intent item $\langle e, p \rangle$.

6.2.2 Phase II: Kernel Enqueue and Sequencing

(1) Triggering Party Kernel Layer

(2) Action

- The kernel receives the intent item from Runtime.
- Invokes the Kernel operator $\mathsf{schedule}(\Sigma, e)$ to mount the event to the intent queue $\mathcal{P}$.
- At this point, the system clock $\mathcal{T}$ remains unchanged, but the configuration of Σ has undergone logical pre-allocation.

6.2.3 Phase III: Logical Reduction and Judgment

(1) Triggering Party Kernel Layer (Core Engine)

(2) Action

- The kernel loops to invoke $\mathsf{peek}(\Sigma)$ to retrieve the head event from the queue.
- Core Validation: Perform judgment based on the Core layer's type checking rules:

$$\Gamma, \mathcal{K}, \mathcal{T} \vdash p : \mathsf{Pre}(e, \Sigma)$$

- Reduction Computation: Execute $\mathsf{Op}(e)$. At this point, the Core layer performs β and ι reductions to compute candidate new states Σ_{try}.
- Postcondition Closure: Verify $\Sigma' \vdash p' : \mathsf{Post}(e)$.

6.2.4 Phase IV: Atomic Materialization and Persistence

(1) Triggering Party Runtime Layer (Storage Subsystem)

(2) Action

- The kernel issues the validated Σ' to Runtime.
- Runtime invokes the materialization mapping $\mathcal{M} \vdash \Sigma' \Downarrow \mathcal{M}'$.
- Physical Confirmation: The underlying database returns ACK, and the logical clock executes tick, formally completing the state jump.
- Causal Anchoring: Record the transition item $\delta = \langle \Sigma \xrightarrow{e} \Sigma' \rangle$ in physical storage.

Table 5: Entity Attribute Table in System Evolution Process

Step	Entity	Data Form	Responsible Layer	Property
1	Signal	Raw Bitstream (Raw Bits)	Physical	Non-deterministic
2	Proof	Dependent Pair $\langle e, p \rangle$	Runtime/Core	Constructive
3	Intent	Pending Queue $\mathcal{P}$	Kernel	Ordered
4	Reduction	λ-term Evolution	Core/Kernel	Deterministic
5	Fact	Persistent Knowledge $\mathcal{K}$	Runtime	Immutable

Protocol Consistency Guarantee (Global Invariant) This protocol enforces a global invariant: "Any bit flip in physical storage must have a complete proof chain extending from Ontology to Core." This means the KOS-TL system has no "undefined behavior." Any operation not satisfying this protocol path (e.g., illegal injection, proof missing, clock reversal) will be automatically intercepted at its respective layer and rolled back to the previous well-formed state Σ_{last}.

6.3 Interaction Interface

6.3.1 Core and Kernel Interaction Interface: Type Judgment Interface (Logic-Kernel Interface)

- **Direction**: Kernel calls Core.
- **Interaction Content**: The Kernel submits the current intent e and its carried proof term p to the Core.
- **Interface Primitives**: $\mathsf{check}(\Gamma, p, \mathsf{Pre}(e))$ and $\mathsf{reduce}(\mathsf{Op}(e), \sigma)$.
- **Properties**: Intensional. It is purely logical and does not perceive physical time or hardware states.

6.3.2 Kernel and Runtime Interaction Interface: Evolution-Driven Interface (Kernel-Runtime Interface)

- **Direction**: Bidirectional.
- **Interaction Content**:
  - *Upward* (Runtime → Kernel): Push refined event pairs $\langle e, p \rangle$ into the queue.

  - *Downward* (Kernel → Runtime): Issue validated new states σ' for materialization.
- **Interface Primitives**: $\mathsf{schedule}(e, p)$ and $\mathsf{commit}(\sigma')$.
- **Properties**: Atomicity. Ensures synchronization between logical state jumps and physical storage updates.

6.3.3 Core and Runtime Lateral Dependency: Refinement Template Interface (Refinement Interface)

- **Direction**: Runtime references Core.
- **Interaction Content**: The Runtime's elab operator needs to reference ontology templates defined in the Core layer to construct valid proofs.
- **Properties**: Constructiveness. Ensures that data extracted from physical signals conforms to the sorts (Sorts) defined in the logical specification.

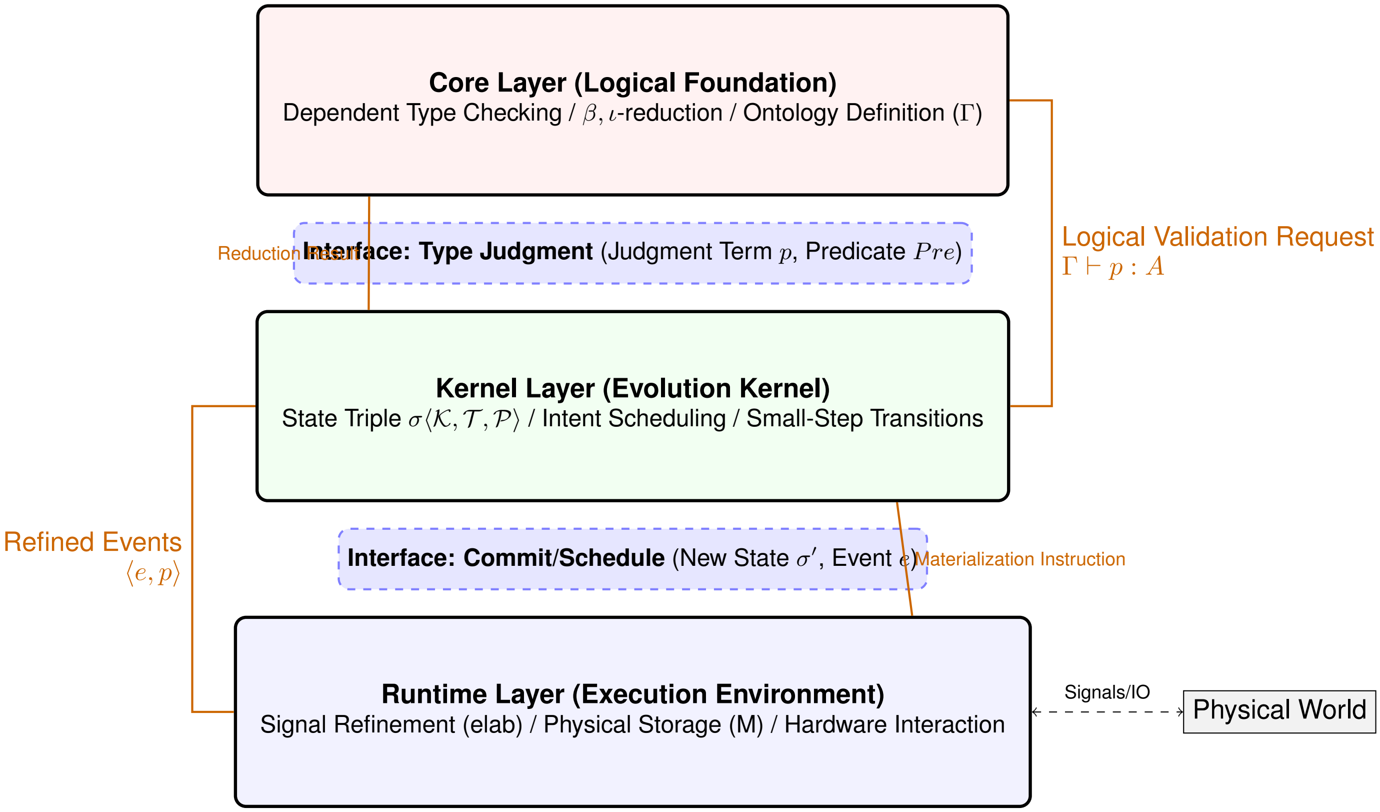
Core Layer (Logical Foundation)
Dependent Type Checking / β, ι-reduction / Ontology Definition (Γ)
Reduction Result
Interface: Type Judgment (Judgment Term p, Predicate Pre)
Logical Validation Request
$\Gamma \vdash p : A$
Kernel Layer (Evolution Kernel)
State Triple $\sigma\langle \mathcal{K}, \mathcal{T}, \mathcal{P} \rangle$ / Intent Scheduling / Small-Step Transitions
Refined Events
$\langle e, p \rangle$
Interface: Commit/Schedule (New State σ', Event e)
Materialization Instruction
Runtime Layer (Execution Environment)
Signal Refinement (elab) / Physical Storage (M) / Hardware Interaction
Signals/IO
Physical World

Figure 1: KOS-TL Layered Interaction Interface Diagram

KOS-TL couples "knowledge" (static knowledge and proofs) with "action" (dynamic state transitions) via Σ-types. From an overall perspective, it is a self-consistent, computable logical entity: The Core provides the semantic framework, the Kernel provides evolutionary power, and the Runtime provides environmental mapping. This architecture enables complex systems not only to store data but also to achieve causal tracing and compliance self-verification through logical reduction.

6.4 System Properties

Theorem 18. *Knowledge Monotonicity*

Let σ be the kernel knowledge base (i.e., the set of accepted facts), and ku a well-formed knowledge object such that $\Gamma \vdash ku : A$. If σ satisfies the introduction condition for ku (denoted $\sigma \vdash ku$), then for any subsequent state σ' satisfying evolutionary consistency, if there is no conflict proof π against ku (i.e., $\sigma' \nvdash \mathsf{refute}(ku)$), then:

$$\sigma \subseteq \sigma' \implies (\sigma' \vdash ku)$$

That is: Established truths remain unchanged under valid expansions of the knowledge base.

Proof. We prove using the Kripke Semantics framework and the Weakening Lemma of constructive logic:

1. Establish the Frame Expansion Model:

We define the kernel state evolution as a Kripke frame $\langle W, \leq, \Vdash \rangle$, where:

- W is the set of all possible knowledge base states.
- $\leq$ is a partial order on W, where $\sigma \leq \sigma'$ indicates σ' is a valid evolutionary successor of σ.
- $\Vdash$ is the forcing relation, where $\sigma \Vdash ku$ indicates that in state σ, the proof term for knowledge object ku is constructible.

2. Prove Persistence of the Core Layer:

The Core layer of KOS-TL is based on intuitionistic type theory. In intuitionistic logic, all operators (Π, Σ, etc.) satisfy persistence. We induct on the proof structure of ku:

- **Base Items**: If ku is an atomic fact (e.g., physical constant or verified ID), by the Kripke model definition, if $\sigma \Vdash ku$ and $\sigma \leq \sigma'$, since $\sigma \subseteq \sigma'$, then ku and its original proof evidence still exist in σ'.
- **Composite Items**: If $ku = \langle v, p \rangle$ is a dependent pair, by the induction hypothesis, the value v remains unchanged under expansion. For the proof term p, since σ' only adds new facts without introducing a counterproof against p (guaranteed by the theorem premise), by the type theory's **Weakening Lemma**, $\Gamma, \sigma \vdash p : P \implies \Gamma, \sigma' \vdash p : P$ still holds.

3. Exclusion of Causal Cancellation:

In KOS-TL, an item only moves from the "current active base" to the "historical archive" when the kernel explicitly constructs a contradiction item $\mathsf{contra}(ku)$. If $\sigma' \nVdash \mathsf{refute}(ku)$, then there is no opposing evidence in σ''s search space that can reduce-collapse with ku. Therefore, the logical evolution operator unify maintains the accessibility of ku.

Conclusion: $\sigma' \Vdash ku$ holds, and knowledge has monotonicity. □

Property Discussion: Significance for "Causal Tracing" Knowledge Monotonicity solves the "memory inconsistency" problem in complex systems: Evidence Persistence: It guarantees that if the banking system proves a transaction compliant at time T1, unless evidence forgery is discovered at T2 (counterproof), this compliance conclusion will never inexplicably disappear due to database cleanup or addition of other transactions. Decision Consistency: This enables unmanned systems built on KOS-TL (e.g., autonomous driving) to maintain long-term environmental cognition, avoiding "forgetting" previous safety boundaries when processing new sensor information.

Theorem 19. *Computational Reflexivity*

In the KOS-TL kernel, there exists a reflection operator $\mathsf{reflect}$*, such that for any well-formed term $t \in \mathcal{D}_{Core}$ and its evolution step* $\mathsf{step} : t \xrightarrow{small} t'$ *in the Kernel layer, the system can automatically synthesize an internal proof term π, satisfying:*

$$\Gamma \vdash \pi : \mathit{EvalPath}(t, t')$$

where $\mathit{EvalPath}$ *is a dependent type recording the axiomatized derivation sequence from t to t'. This means every state change in the kernel comes with a "meta-proof" of its own legitimacy.*

Proof. We prove using identity types in Martin-Löf type theory and meta-circular mapping:

1. Algebraic Mapping of Reduction Steps:

Since the Core layer of KOS-TL is based on pure, side-effect-free dependent type calculus, its computational semantics is referentially transparent. Every reduction step $t \xrightarrow{small} t'$ is not a random memory overwrite but the application of a specific reduction rule (e.g., β-reduction or ι-reduction).

2. Automatic Synthesis of Proof Terms:

For each type of basic reduction executed by the kernel, we define a mapping function $\mathcal{R}$:

- **Beta Reduction**: When executing $(\lambda x.M)N \to M[N/x]$, the kernel constructs $\pi = \mathsf{refl}_\beta(M, N)$ using the internal axiom $\mathsf{beta\_axiom}$.
- **Iota Reduction**: When executing $\mathsf{proj}_1\langle a, b\rangle \to a$, the kernel constructs $\pi = \mathsf{refl}_\iota(a, b)$ using $\mathsf{proj\_axiom}$.

Since all reduction rules have corresponding axioms defined in the Core layer, the kernel can synchronously record the axiom sequence used while performing computation.

3. Establish Equivalence Using J-Eliminator:

In dependent type theory, the only constructor for the equivalence type $\mathsf{Id}_A(t, t')$ is refl. According to the J-operator (Identity Elimination), if two terms are equivalent under logical reduction, they are indistinguishable under all logical predicates. By mapping each execution action $t \to t'$ of the Kernel to the application process of the J-operator, the kernel is actually continuously constructing a mathematical testimony of "why I changed from t to t").

4. Meta-circular Self-Audit:

There exists a subroutine $\mathsf{Audit} \subset \mathsf{Kernel}$, which takes the proof term π and path $\mathsf{EvalPath}$ as input. Due to the strong normalization property of KOS-TL, Audit can verify in finite steps whether π indeed supports the transition from t to t'. □

Property Discussion: Significance for "Autonomous Systems" Computational Reflexivity elevates KOS-TL to the level of a **"self-aware system": Full-Time Automatic Audit: Traditional systems require external audit logs, while KOS-TL's logs are its execution paths. This means audit is not "post-hoc smoke," but "preemptive proof." Decision Transparency: In autonomous driving or financial transactions, when the system makes a decision (e.g., emergency obstacle avoidance or transaction interception), reflexivity ensures the system can immediately output a human-readable and mathematically valid "compliance explanation report." Logical Basis for Self-Repair: When the system detects a deviation in the hardware refinement mapping, it can pinpoint the conflicting logical operator by reflexively comparing the "expected path" with the "actual path."

Theorem 20. *System-Wide Safety*

Let $\mathcal{S}$ be the system's physical state space, and $\mathsf{Safe} \subseteq \mathcal{S}$ the predefined physical safe subset. If the initial state $s_0 \in \mathsf{Safe}$ of the KOS-TL system, then for any physical evolution sequence $s_0 \xrightarrow{hw} s_1 \xrightarrow{hw} \dots \xrightarrow{hw} s_n$, it always holds that:

$$\forall i \geq 0, \quad s_i \in \mathsf{Safe}$$

Under the premise conditions:

(1) The Core layer satisfies consistency (Consistency).

(2) The Kernel layer satisfies progress (Progress) and evolutionary consistency.

(3) The Runtime layer satisfies refinement fidelity (Refinement Fidelity).

Proof. The proof uses layered induction, mapping physical evolution to reductions of logical proof terms.

1. Base Case

For the initial state s_0, according to Runtime's refinement fidelity:

$$\mathcal{E}(s_0) = ku_0 \quad \text{and} \quad \Gamma \vdash ku_0 : \mathsf{Qualified}(s_0)$$

Since $s_0 \in \mathsf{Safe}$, the corresponding predicate $\mathsf{is\_safe}(\mathsf{proj}_1(ku_0))$ has a proof term p_0 in the Core layer.

2. Inductive Step

Assume the system is in $s_i \in \mathsf{Safe}$ at step i. Consider the migration to step $i + 1$:

A. Physical Disturbance and Refinement:

When the physical environment changes $s_i \xrightarrow{hw} s_{i+1}$ (e.g., sensor value changes or hardware failure), the Runtime immediately captures the change and attempts to construct a new knowledge object ku_{i+1}:

$$\mathcal{E}(s_{i+1}) = ku_{i+1}$$

B. Kernel Logical Judgment:

The kernel submits ku_{i+1} to the unify operator. This produces two branches:

- **Branch 1:** s_{i+1} **Still in Safe:** The kernel can successfully construct p_{i+1} : $\mathsf{is\_safe}(s_{i+1})$ based on Core layer rules. According to the Kernel's evolutionary consistency, the state σ updates to include ku_{i+1}, maintaining safety.
- **Branch 2:** s_{i+1} **Attempts to Cross Safe Boundary:** At this point, no proof term of type $\mathsf{is\_safe}(s_{i+1})$ can be constructed in the Core layer. According to Core layer consistency (cannot prove false propositions), the kernel's logical engine produces a reduction block (or type conflict).

C. Self-healing Loop:

According to Kernel progress, the kernel does not deadlock; it executes find_root_cause and triggers analyze. The Runtime receives the safety instruction π from the kernel, and according to observational adequacy, this instruction enforces execution at the physical layer (e.g., circuit breaker, switch to redundant path), pulling the physical state back to $s'_{i+1} \in \mathsf{Safe}$.

3. Reductio ad Absurdum

Assume there exists some $s_j \notin \mathsf{Safe}$:

(1) This means Runtime must refine a logical item ku_j such that $\Gamma \vdash ku_j : \mathsf{is\_safe}$.

(2) But $s_j \notin \mathsf{Safe}$ means $\mathsf{is\_safe}(s_j)$ is equivalent to $\perp$ (false proposition) in the Core layer.

(3) Then it derives $\Gamma \vdash ku_j : \perp$.

(4) This violates the Core layer consistency theorem (no item for false propositions exists in the system).

(5) Hence $s_j \notin \mathsf{Safe}$ is logically unconstructible.

□

Table 6: The Logical Spectrum of KOS-TL

Property Name	**Belonging Layer**	**Core Value**	**Formal Metaphor / Definition**
Consistency	Core	Root out logical contradictions	$\sigma \nvdash \perp$
Strong Normalization	Core	Ensure real-time response	$\forall t, \exists v$: NormalForm, $t \twoheadrightarrow v$
Progress	Kernel	Continuous self-healing operation	$\mathcal{C} \notin \text{Final} \implies \exists \mathcal{C}' : \mathcal{C} \xrightarrow{small} \mathcal{C}'$
Evolutionary Consistency	Kernel	Safe state evolution	$\sigma \xrightarrow{T} \sigma' \implies \text{TypeCheck}(\sigma') = \text{Success}$
Monotonicity	**System-wide / Kernel**	Causal evidence persistence	$\sigma \subseteq \sigma' \implies (\sigma \Vdash ku \implies \sigma' \Vdash ku)$
Fidelity	Runtime	Lossless physical mapping	$(s, ku) \in \text{SimulationRelation}$
Adequacy	Runtime	Lossless instruction delivery	$\text{Exec}(\mathcal{G}(\mathit{ctrl}), s) \models \text{Refine}^{-1}(\phi)$
Reflexivity	**System-wide**	Full-path audit tracking	$\forall t \to t', \exists \pi : \text{Id}(t, t')$

6.5 Characteristics and Applications

In compliance auditing for multinational banks, processing hundreds of millions of SWIFT transaction records is not merely a *big data* challenge, but fundamentally a challenge of *logical correctness*. Traditional systems typically oscillate between statistical anomaly detection (e.g., identifying frequent high-value transfers) and hard-coded rule engines, which often leads to an overwhelming number of false positives. With the introduction of KOS-TL, compliance auditing is transformed from a "probabilistic black box" into a *formal causal system*.

Below, we elaborate in detail on the logical architecture and execution workflow of this integrated approach.

6.5.1 Logical Abstraction: Defining Anti-Money Laundering Axioms

In KOS-TL, compliance is not a flag in a database, but a *proof goal*.

A. Invariants of Fund Flows At the Core layer, compliant transactions are defined using dependent types. A compliant transaction T must satisfy:

$$\mathsf{ValidTx} \equiv \Sigma(t : \mathsf{TxData}).\ \Sigma(e : \mathsf{Evidence}).\ \mathsf{CheckCompliance}(t, e)$$

where:

- t: SWIFT message data (sender, receiver, amount),
- e: business logic evidence (e.g., hashes of trade contracts, customs declarations),
- $\mathsf{CheckCompliance}$: a logical function requiring semantic alignment between t and e (e.g., consistency between goods value and transfer amount within acceptable tolerance).

B. Topological Axiom: Acyclicity A core feature of money laundering is *layering and integration*, often manifested as funds circulating through multiple entities and eventually returning to the origin. We define a path type $\mathsf{Path}(A, A)$. If an inhabitant of this type can be constructed without any substantive transformation of fund attributes, a logical contradiction is detected.

6.5.2 Integrated Execution: From Massive Data to Logical Evidence

Stage I: Large-scale Filtering (Database Layer – Efficiency First) Underlying databases (e.g., ClickHouse or Neo4j) leverage high concurrency to perform initial graph-based analyses.

- Task: Identify suspicious cycles or high-risk node associations among hundreds of millions of records.
- Result: Approximately 10,000 suspicious transaction chains are extracted. At this stage, they are statistically suspicious but not yet conclusively classified.

Stage II: Evidence Request and Refinement (Runtime Layer – Fidelity) The KOS-TL kernel takes over these 10,000 chains. For each chain, the Runtime module issues evidence backfill requests to business systems.

- Operation: Request underlying contracts and bills of lading for the corresponding SWIFT transactions.
- Fidelity: Evidence is refined into ku_{evidence}, accompanied by immutable timestamps and provenance proofs.

Stage III: Dependent Type Verification (Core/Kernel Layer – Rigor) This is the core step of KOS-TL. The kernel attempts to construct a *compliance proof term* p for each suspicious chain.

Theorem 21. *Transaction Compliance Verification*

For a suspicious chain $L = \{t_1, t_2, \ldots, t_n\}$ to be marked as $\mathsf{Verified}$, a total proof term must be constructed:

$$P_{\mathit{total}} = \langle p_1, p_2, \ldots, p_n \rangle$$

such that each p_i proves that the evidence e_i eliminates the "cyclicity hypothesis" induced by the chain.

6.5.3 Logical Interception

If a transaction corresponds to fictitious trade (e.g., transfer amount does not match the contract hash), the Core layer cannot construct a proof term. By the consistency theorem, the kernel cannot evolve the chain state to $\mathsf{Verified}$.

6.5.4 Case Study: Cooling System Fault Handling

Physical state s: Abnormal voltage fluctuations from the pressure sensor of cooling pump A.

Safety goal Safe: Pressure must remain within $[P_L, P_H]$, and the sensor must possess valid calibration proof.

Step 1: Refinement and Fidelity (Runtime Layer) A raw voltage signal of $2.4V$ is refined:

$$ku_{\text{press}} = \langle 120kPa, p_{\text{calib}} \rangle : \mathsf{Press}$$

Without p_{calib}, no term of type Press can be constructed.

Step 2: Evolution and Monotonicity (Kernel Layer) The kernel updates the global state:

$$\sigma_{\text{new}} = \mathsf{unify}(\sigma, ku_{\text{press}})$$

Previously recorded facts (e.g., "Pump A is active") are preserved.

Step 3: Reduction and Consistency (Core Layer) The kernel evaluates:

$$\mathsf{is\_safe}(v) \equiv (P_L \leq v \leq P_H)$$

With $P_H = 110kPa$, the proof term p_{safe} : $\mathsf{is\_safe}(120)$ cannot be constructed. The system cannot falsely assert safety.

Step 4: Reflexive Audit and Self-Healing A control instruction *ctrl* is generated (e.g., activate pump B, shut down pump A), along with proof:

$$\pi : \mathsf{Id}(\sigma_{\text{fault}}, \sigma_{\text{recovery}})$$

This proof explains the decision in terms of Core-level constraints.

Step 5: Observational Adequacy (Runtime Layer) The logical instruction Close(Pump_A) is refined into concrete bus signals, guaranteeing the intended physical effect.

7 Application of KOS-TL

7.1 Application Background: Quality Anomaly Traceability in Manufacturing

Consider a large discrete manufacturing enterprise whose core challenge is the following:

> *When a certain batch of products exhibits severe quality defects, can the system automatically trace its production process, identify potential anomalies related to equipment, personnel, or raw materials, and produce an executable and explainable causal chain?*

This problem exhibits several typical characteristics:

- Heterogeneous data sources (work orders, equipment logs, personnel schedules, quality inspection records);
- Strong temporal ordering and causal constraints;
- Inference results must directly support production decisions and responsibility attribution.

The system involves the following core tables (originating from different subsystems):

(1) Product records: `Product(ProductID, Model)`
(2) Batch records: `Batch(BatchID, ProductID, ProduceDate)`
(3) Production lines: `ProductionLine(LineID, Factory)`
(4) Process routes: `ProcessRoute(Model, StepName, StepOrder, TargetLineType)`
(5) Process thresholds: `ProcessThreshold(Model, StepName, ParamName, MinValue, MaxValue)`
(6) Step execution details: `StepExecution(WOID, StepName, StartTime, EndTime, EquipID)`
(7) Sensor time series: `SensorTimeSeries(EquipID, ParamName, Value, Timestamp, DeviceStatus)`
(8) Work orders: `WorkOrder(WOID, BatchID, LineID)`
(9) Operators: `Operator(OperatorID, Name, Role)`
(10) Operation logs: `OperationLog(LogID, WOID, OperatorID, Time)`
(11) Equipment: `Equipment(EquipID, LineID)`
(12) Equipment status: `EquipmentStatus(EventID, EquipID, Status, Time)`
(13) Process parameters: `ProcessParam(LogID, ParamName, Value)`
(14) Quality inspections: `QualityInspection(InspectID, BatchID, Result, Time)`
(15) Defect reports: `DefectReport(ReportID, BatchID, DefectType)`
(16) Supply chain records: `SupplierPart(PartID, SupplierID, BatchID)`

To illustrate the approach, we trace a causal reasoning workflow for quality anomaly analysis in bearing production, as shown in Table 7.

By applying KOS-TL reasoning, the system does not return a simple SQL query result to the user, but rather a *logical proof package*. When the user opens the report, the system can expand prf_{causal} and directly locate the original PLC logs corresponding to the temperature fluctuation, since these logs are integral components in the construction of the report r.

Table 7: Causal Reasoning for Quality Traceability in Bearing Manufacturing

Step	System Action	Concrete Data Example
Input	Quality inspection reports anomaly	Batch_202310-01 detected “non-uniform hardness” at 10:00 on 2023-10-10.
Type instantiation	Construct f_{fail}	f_{fail} : $\mathsf{FailureEvent} = \langle$"B2310", "HARD_ERR", 10:00$\rangle$
Kernel reasoning	Search for causal evidence	Retrieved that the batch passed through furnace HeatTreatment_03 at 08:00, and a temperature fluctuation a_{temp} occurred at 07:55.
Logical synthesis	Construct causal chain	$r = \langle f_{fail}, a_{temp}, \text{prf}_{causal}\rangle$. The proof term prf_{causal} automatically verifies $07{:}55 < 10{:}00$.

7.2 The Application Workflow of KOS-TL

In practical deployment, KOS-TL mainly involves the following stages:

1. Definition of initial atomic types, predicate types, events, and constraints. This part belongs to the *Core* layer. Such definitions essentially determine the boundary of logical validity of the system being modeled.
2. The *Runtime* layer serves as the interface between the system and the external world. Through the runtime, the KOS-TL system acquires data and refines it into typed objects (logically operable entities).
3. The *Kernel* layer is responsible for concrete knowledge operations.

7.2.1 Kernel Layer: Rule Definitions and Logical Constraints

For the problem of *causal reasoning in bearing production quality traceability*, the Kernel layer defines the corresponding types and constraints.

(1) Basic Atomic Types The basic atomic types are shown in Table 8, including $\mathsf{BatchID}$, $\mathsf{Machine}$, and Time.

Table 8: Domain Concepts and Type Refinement Relations

Domain Concept	Logical Type (Core)	Kernel Atomic Type	Refinement Logic
Time	Time	Float / UInt64	Direct mapping, representing Unix timestamps or logical clocks.
BatchID	BatchID	Val / String	$\Sigma(s : Val)$.Proof$(isIDFormat(s))$
Machine	Machine	Val / Enum	$\Sigma(v : Val)$.Proof$(v \in EquipRegistry)$

(2) Predicate Types Predicate types include:

- $\mathsf{InRoute}(b, m)$, which defines whether batch b is allowed to be processed on machine m.
- $\mathsf{Overlap}(t, dur)$, which defines whether time point t falls within duration dur.

(3) Events and Constraints

(i) **Failure Event Type** ($\mathsf{FailEvt}$)

$$\mathsf{FailEvt} \equiv \Sigma(b : \mathsf{BatchID}).\Sigma(err : \mathsf{ErrorCode}).\Sigma(t : \mathsf{Time}).\mathsf{Proof}(t \in \mathsf{Shift}_{QA})$$

This type not only records which batch failed, but also enforces a proof that the inspection time lies within the QA shift.

(ii) **Process Step Type** ($\mathsf{ProcStep}$)

$$\mathsf{ProcStep} \equiv \Sigma(b : \mathsf{BatchID}).\Sigma(m : \mathsf{Machine}).\Sigma(dur : \mathsf{Time} \times \mathsf{Time}).\mathsf{Proof}(\mathsf{InRoute}(b, m))$$

The predicate $\mathsf{InRoute}$ guarantees that the batch is processed on machine m according to the defined process route.

(iii) **Environmental Anomaly Type** ($\mathsf{Anomaly}$)

$$\mathsf{Anomaly} \equiv \Sigma(m : \mathsf{Machine}).\Sigma(p : \mathsf{Param}).\Sigma(v : \mathsf{Val}).\Sigma(t : \mathsf{Time})$$

(iv) **Causal Validity Constraint** ($\mathsf{CausalProof}(a, f)$)

$$\mathsf{isBefore}(t(a), t(f)) \wedge \mathsf{isSameResource}(\mathsf{location}(a), \mathsf{process}(f))$$

Traceability is defined as a proof search problem:

$$\forall f : \mathsf{Failure}, \exists(a, \pi) : \Sigma(a : \mathsf{Anomaly}).\mathsf{CausalProof}(a, f)$$

(v) **Causal Proof** ($\mathsf{CausalProof}$)

$$\mathsf{CausalProof}(a, f) \equiv \Sigma(e : \mathsf{ProcStep}).\mathrm{Prop}_{causal}(a, e, f)$$

where Prop_{causal} enforces:

- Temporal logic: $a.t \in e.dur \wedge e.dur.end < f.t$;
- Spatial logic: $a.m = e.m$;
- Batch consistency: $e.b = f.b$.

(vi) **Root Cause Report** ($\mathsf{RootCauseReport}$)

$$\mathsf{RootCauseReport} \equiv \Sigma(f : \mathsf{FailEvt}).\Sigma(a : \mathsf{Anomaly}).\mathsf{CausalProof}(a, f)$$

Semantically, this definition encodes:

- the existence of a failure f;
- the existence of a physical anomaly a;
- a non-Boolean causal proof witnessing their relation.

7.2.2 Runtime Layer: Data Acquisition and Elaboration

The Runtime layer extracts data from external databases and refines it into Kernel-level objects, thereby *logicalizing* raw data.

Typical source tables include:

- `Product_Master`: process routes;
- `Execution_Log`: work order execution records;
- `IoT_Sensor_Stream`: sensor streams;
- `Quality_Report`: inspection results.

These data are elaborated into proof-carrying objects such as:

$$f_0 = \text{mkFailure}(\text{Batch\_202310-01}, \text{Hardness\_Issue}, 10{:}00, \pi_{QA})$$

7.2.3 Core Layer: Proof Construction and Small-Step Evolution

The Core layer executes proof construction following Kernel rules. It operates via small-step semantics, gradually evolving configurations until a *RootCauseReport* is materialized in the global knowledge base.

In summary, KOS-TL integrates runtime data acquisition, kernel-level logical constraints, and core-level proof synthesis into a unified, type-safe reasoning pipeline. Rather than producing opaque query results, the system yields formally verified, explainable causal reports whose correctness is guaranteed by construction.

7.3 KOS-TL Adapting to Changes in Business Rules

We continue with the same example from the previous subsection—*bearing heat treatment*—and assume that the business rules change as follows:

To prevent temper brittleness, if a *voltage anomaly* occurs during the heat treatment process, the system must additionally check whether the *cooling water circulation pressure* during the same time period is abnormal. Only when *both* anomalies are present can the situation be classified as a severe quality defect.

In the KOS-TL framework introduced in the previous subsection, this change requires modifying *only* the type definition of $\mathsf{CausalProof}$ in the **Core** layer.

7.3.1 Type definition before modification

$$\mathsf{CausalProof}(a, f) \;\equiv\; \Sigma(e : \mathsf{ProcStep}).\, \mathsf{Prop}_{time}(a, e, f)$$

7.3.2 Type definition after modification (injecting the new rule)

We redefine CausalProof as a dependent type that must contain evidence of a *dual anomaly*:

$$\mathsf{CausalProof}(a, f) \;\equiv\; \Sigma(e : \mathsf{ProcStep}).\, \Sigma(w : \mathsf{WaterPressureAnomaly}).\, \mathsf{Prop}_{joint}(a, w, e, f)$$

The new constraint requires not only the presence of a voltage anomaly a, but also enforces the existence of a water pressure anomaly w within the same production process e.

Once the type in the Core layer changes, a chain reaction is automatically triggered in the **Kernel** layer through its small-step reduction logic:

(1) **Constructor invalidation.** The original constructor mkCausalProof immediately fails type checking in the Kernel due to missing parameters (namely w and its corresponding proof).

(2) **Automatic triggering of new search.** Upon detecting that the target type requires a WaterPressureAnomaly, the Kernel automatically initiates a search over water pressure data in the environment σ.

7.3.3 Reorganization of proof synthesis paths

Case 1 (Voltage anomaly only). The Kernel cannot find a matching water pressure anomaly w, and therefore fails to construct a complete π_{causal} term. The inference result is automatically classified as *invalid*.

Case 2 (Dual anomalies). The Kernel automatically composes the voltage anomaly, the water pressure anomaly, and the production process into a new result r_{final}.

The essence of this capability is an extreme form of *Type-Directed Development*, as summarized in Table 9.

Table 9: Mechanisms of KOS-TL

Property	Mechanism	Implication
Self-healing	If the runtime data source does not provide water pressure data, the Kernel raises a “missing type” error rather than producing an incorrect conclusion.	Strictly guarantees the safety of conclusions and prevents blind traceability.
Push-down logic	New rules are propagated downward through the signature of mkCausalProof; the Kernel’s search algorithm automatically detects new parameter requirements.	Developers do not need to rewrite search algorithms; algorithms adapt automatically to type changes.
Zero redundancy	Existing traceability code does not need to be removed; once the referenced types are updated, its behavior changes automatically.	Achieves true “configuration-as-logic.”

In this example, neither the analyze function nor the getProductionContext function is modified. By changing only the type signatures in the Core layer, one redefines the physical boundary of *what counts as truth*.

The Kernel layer behaves like a fully automated puzzle-solving machine: once the puzzle template (Core) is changed, it automatically adjusts its strategy for searching puzzle pieces (Data) and the final assembled picture (Result).

This is the core value of KOS-TL when dealing with complex and evolving industrial environments: ensuring atomic-level consistency between the evolution of system logic and changes in business rules.

7.4 KOS-TL Enabling Cross-Domain Logical Consistency

KOS-TL can further achieve cross-domain logical consistency through a *shared logical kernel* and *cross-domain type references*.

In KOS-TL, neither the financial domain nor the quality domain directly accesses raw databases. Instead, both domains subscribe to the same *knowledge objects* materialized by the Kernel layer.

Continuing the previous example of *bearing heat treatment*, consider the following scenario:

If an abnormal heat-treatment voltage is detected (a quality risk), then the payment to the raw material supplier associated with the affected batch must automatically enter a "pending audit" state, and the corresponding financial voucher must include evidence of the quality anomaly (financial risk control).

KOS-TL realizes this requirement by defining *cross-dependent types* at the Core layer.

7.4.1 Logical Extension in the Quality Domain

At the quality Core layer, we have already defined

$$r_{quality} : \mathsf{RootCauseReport},$$

which certifies that voltage fluctuation caused the quality defect.

7.4.2 Type Definition in the Financial Domain (New)

At the financial Core layer, we introduce a new type AuditLock:

$$\mathsf{AuditLock} \equiv \Sigma(inv : \mathsf{Invoice}).\Sigma(r : \mathsf{RootCauseReport}).\mathsf{Proof}(inv.batch = r.f.b)$$

The constructor of the financial object AuditLock *mandatorily* requires an input r : RootCauseReport. Without a generated quality report r, the financial layer cannot construct a valid audit-lock object.

Once quality traceability is completed, a new item r_{final} is added to the Kernel state σ. At this moment, the cross-domain logical engine automatically triggers the next reduction step:

- **Financial Observer Triggered:** The financial system's analyzeAudit procedure detects that a quality anomaly report r_{final} exists in σ with the same batch as invoice inv_{01}.
- **Cross-Domain Item Composition:** Through unification, the Kernel directly injects the quality-domain evidence r_{final} into the financial-domain AuditLock object.
- **Atomic Update:**

  $$\langle \sigma, \mathsf{FinanceGoal} \rangle \xrightarrow{\text{small}} \langle \sigma \cup \{\mathsf{lock}_{01}\}, \mathsf{unit} \rangle$$

  where

  $$\mathsf{lock}_{01} = \langle inv_{01}, r_{final}, \pi_{match} \rangle.$$

In the cross-domain architecture of KOS-TL, the financial procedure analyzeAudit is not an isolated software module. Instead, it is a *predicate listener* mounted on the shared logical Kernel. Its core responsibility is to monitor the evolution of the state σ and to automatically derive audit actions whenever specific *financial–quality cross-constraints* are satisfied.

7.4.3 The analyzeAudit Function

At the Core layer, analyzeAudit is defined as a higher-order function whose purpose is to construct an audit-lock item:

$$\mathsf{analyzeAudit} : \Pi(inv : \mathsf{Invoice}) \to \mathsf{Option}(\mathsf{AuditLock})$$

Its internal derivation logic follows the rule:

$$\frac{inv \in \sigma \quad \exists r : \mathsf{RootCauseReport} \text{ s.t. } \mathsf{Unify}(inv.batch, r.f.b)}{\mathsf{derive}(\mathsf{AuditLock}(inv, r))}$$

- **Input:** A financial invoice inv pending processing.
- **Trigger Condition:** A quality report r exists in the Kernel state whose batch unifies with the invoice batch.
- **Output:** If the condition holds, an audit object containing quality evidence is produced; otherwise, None is returned.

When a voltage fluctuation occurs on the production line and a quality report is generated, analyzeAudit undergoes the following evolution at the Kernel layer:

Step 1: Cross-Domain Detection The Kernel detects the newly materialized r_{final}. Since analyzeAudit subscribes to changes in the BatchID type, it is immediately activated.

Step 2: Dependency Extraction Using the index `inv.batch`, the program locates the associated financial invoice. For example:

$$\text{Batch\_10-01} \longrightarrow \text{Inv\_2023\_009 (Supplier: SteelCo)}$$

Step 3: Proof Transparency This is the most critical step. $\mathsf{analyzeAudit}$ does not merely flag an issue; it directly *references* the proof object π_{causal} from the quality domain:

$$\mathsf{lock}_{item} = \langle \mathsf{Inv}_{009}, r_{final}, \pi_{match} \rangle$$

This means that the financial system now holds the *physical evidence* of the quality anomaly, such as voltage curves and computational records.

Step 4: Action Materialization Once lock_{item} is instantiated in the Kernel, the Runtime-layer financial plugin captures this state change and immediately executes the following actions in the ERP system:

- **Payment Suspension:** Freeze settlement of $\mathsf{Inv\_2023\_009}$.
- **Audit Endorsement:** Automatically attach the logical trace of r_{final} to the invoice for review.

Algorithm 1 KOS-TL Cross-Domain Audit Procedure: $\mathsf{analyzeAudit}(r_{final})$

```
Require: Newly materialized root-cause report $r_{final} = \langle f, a, e, \pi_{causal} \rangle$ : $\mathsf{RootCauseReport}$
Ensure: Audit-lock item $\mathsf{AuditLock}$ or $\emptyset$
Step 1: Cross-Domain Detection
Monitor Kernel state $\sigma$; trigger upon materialization of $r_{final}$.
Extract batch index: $b \leftarrow r_{final}.f.b$ {$BatchID$ serves as the unique key for cross-domain unification}
Step 2: Dependency Extraction
Retrieve related invoices from financial state $\sigma_{fin}$:
$inv \leftarrow \{i \in \sigma_{fin} \mid i : \mathsf{Invoice} \wedge \mathsf{Unify}(i.batch, b)\}$
if $inv = \emptyset$ then

    return $\emptyset$
end if
Step 3: Proof Transparency
Construct proof predicate $\pi_{match}$ linking $inv$ and $r_{final}$.
Reference quality proof: $\pi_{ref} \leftarrow r_{final}.\pi_{causal}$
Instantiate audit lock: $lock_{item} \leftarrow \langle inv, r_{final}, \pi_{match}, \pi_{ref} \rangle$
Step 4: Action Materialization
Atomic materialization: $\sigma \leftarrow \sigma \cup \{lock_{item}\}$
Execute $\mathsf{FreezePayment}(inv)$
Execute $\mathsf{AttachEvidence}(inv, r_{final}.trace)$
return $lock_{item}$
```

Suppose an additional financial rule is introduced: *"An audit lock is triggered only if the estimated quality loss exceeds* 20% *of the invoice amount."*

In KOS-TL, this requires only adding a logical predicate to the definition of $\mathsf{analyzeAudit}$:

$$\mathsf{Proof}(r.loss\_estimate > inv.total \times 0.2)$$

Even if a quality report is generated, if the estimated loss is insufficient, the unification in $\mathsf{analyzeAudit}$ fails, and the financial system automatically maintains a *normal settlement* state. This form of logical pushdown ensures that financial decision-making remains mathematically consistent with the physical realities of the production line.

7.5 Counterfactual Reasoning in KOS-TL

In KOS-TL, counterfactual reasoning is not realized through *guessing*, but through kernel-level simulation over **parallel state spaces**.

7.5.1 How KOS-TL Defines Counterfactuals

In traditional AI, counterfactuals are typically expressed as probabilistic queries of the form $P(y \mid do(x))$. In contrast, KOS-TL defines a counterfactual as the evaluation of a virtual configuration:

"If, in the state σ_0, the fact a (voltage fluctuation) had not occurred, would r_{final} (the failure proof) still be instantiable in the kernel?"

7.5.2 Implementation Mechanism: Virtual Context

KOS-TL realizes counterfactual reasoning through *branching* of the context Γ.

- **Shadow State Construction (σ'):** The kernel copies the current knowledge base σ_0, but selectively removes or modifies a specific fact (e.g., removing a_{volt}).
- **Hypothetical Evaluation:** Small-step evaluation is restarted under the new configuration

$$\langle \Gamma, \sigma', \mathsf{analyze}(f_0) \rangle.$$

- **Lemma Comparison:** If the evaluation reduces to $\perp$ (the empty type), then a_{volt} is a necessary condition for f_0 (necessary causation). If the evaluation can still generate r'_{final}, then redundant causation exists, or a_{volt} is merely confounding noise.

7.5.3 Three Application Scenarios of Counterfactual Reasoning

A. Root-Cause Sensitivity Analysis *Question:* If the voltage fluctuation were only 2% instead of 10%, would the hardness still fail to meet specifications?

Kernel Action: Modify the value of a_{volt} in σ and observe whether $\mathsf{mkCausalProof}$ still passes the physical threshold checks defined at the Core layer.

B. Liability Attribution *Question:* If the batch of raw material supplied by Vendor A had not been used, would fluctuations in furnace M_03 alone still cause the failure?

Kernel Action: Remove the raw-material fact in a virtual environment and observe whether the proof chain collapses. This directly supports the recovery and compensation logic in the upper-layer $\mathsf{analyzeAudit}$ procedure.

C. Preemptive Simulation *Question:* If the power of furnace M_03 were increased by 5% tomorrow, would it trigger a similar failure f_0?

Kernel Action: This form of *forward-looking counterfactual* allows the system to complete a "virtual accident" in logical space before any physical incident occurs.

7.5.4 Logical Formulation

In the formal language of KOS-TL, counterfactual reasoning is typically written as:

$$\mathcal{C} \vdash \neg a \;\Rightarrow\; \neg(\exists \pi : \mathsf{Proof}(f))$$

That is, under the current configuration, one proves that *if a does not exist, then no proof object for f can be instantiated.*

7.5.5 Decoupling Construction and Validation

In the $\mathsf{analyze}$ function, r_{final} is indeed a candidate construction that includes a_{volt}. Counterfactual reasoning instead asks:

"If the atomic fact a_{volt} is removed from the axiom system, can another valid proof π' still be constructed that points to f_0?"

- If yes, then a_{volt}, although present in r_{final}, is only a *sufficient but non-necessary condition* (or part of multiple causation).
- If no, then a_{volt} is a *necessary condition*.

7.5.6 The Kernel's Parallel-Space Mechanism: σ vs. $\sigma \setminus \{a\}$

During counterfactual reasoning, the Kernel does not operate on a single triple, but generates an environmental slice.

- **Actual trajectory:**
$$\langle \Gamma, \sigma, f_0 \rangle \;\Rightarrow\; r_{\mathit{final}} \quad \text{(containing } a_{volt}\text{)}$$
- **Counterfactual trajectory:**
$$\langle \Gamma, \sigma \setminus \{a_{volt}\}, f_0 \rangle \;\Rightarrow\; \bot \quad \text{(derivation collapses)}$$

When the counterfactual trajectory reduces to $\bot$ (the bottom type / empty set), it logically reinforces the authority of the actual trajectory. In logic, this is known as *affirmation by negation*.

7.5.7 Elimination Rules in Type Theory

Counterfactual reasoning effectively computes *contribution*:

$$\mathsf{Contrib}(a, f) \iff (\sigma \vdash f) \land (\sigma \setminus \{a\} \nvdash f)$$

Here, $\nvdash$ denotes that, without a, the remaining knowledge base cannot derive a proof of the failure f.

If this condition holds, it rules out the possibility that *even without voltage fluctuation, hardness would still degrade due to other causes*.

7.5.8 Multiple-Cause Illustration

Suppose that, in addition to voltage fluctuation (a_1), there is also excessive impurity in the raw material (a_3).

- **Initial derivation:** Two candidate reports are obtained: $r_1(a_1)$ and $r_3(a_3)$.
- **Counterfactual tests:**
  - Remove a_1; if a_3 can still derive the failure, then a_1 is not the unique root cause.
  - Remove a_3; if the derivation still holds, then a_3 is likely only noise.

7.6 Logic as the System Kernel

Through the reasoning processes above, KOS-TL demonstrates its core strengths as a logical system:

1. **Construction as Evidence**: The generated $\mathsf{RootCauseReport}$ is not merely a piece of text. It contains pointers to the original equipment logs and quality inspection records, together with a complete logical proof chain, thereby providing a *hard guarantee* of explainability.
2. **State Preservation**: At every Small-step transition, the kernel verifies the $Post$ constraints. If a tracing result violates physical logic (e.g., a cause occurring after its effect), the transition fails, ensuring that the system state always remains within a logically consistent space.
3. **Computational Completeness**: Unlike Description Logic with its Open World Assumption, KOS-TL leverages strong normalization at the kernel level to guarantee that any tracing program will, in finitely many steps, either yield a definitive causal conclusion or report insufficient evidence.

In this example, KOS-TL transforms quality tracing from *post-hoc auditing* into *real-time logical calculus*. The system is not asking "why did it fail," but instead attempts to construct a complete logical proof of "the failure itself." This paradigm shift—from **truth judgment** to **proof construction**—endows complex knowledge management in manufacturing with operating-system-level rigor.

This section, through a large-scale manufacturing quality anomaly analysis scenario, illustrates how KOS-TL maps heterogeneous business tables into dependent types and realizes automatic knowledge derivation and causal tracing via Small-step operational semantics.

> *"KOS-TL is not designed merely to describe the world, but to run a logically closed-loop operational control system with knowledge as its kernel."*

In KOS-TL, knowledge derivation is essentially a process of constructing and applying higher-order functions.

Traditional systems *use hard-coded functions to process data*. KOS-TL, by contrast, embodies the idea that *data drives the synthesis of logical terms, which in turn form derived function bodies endowed with reasoning capability*. This mechanism of "data-derived functions" can be understood along the following three dimensions.

Table 10: Fundamental Differences Between Description Logic (DL) and KOS-TL

Dimension	**Description Logic (DL)**	**KOS-TL**
Core Paradigm	Static ontological consistency	Dynamic operational semantics
Logical Property	Truth evaluation	Proof construction
Temporal Handling	External extensions (Temporal DL)	Intrinsic temporal ordering constraints
Application Goal	Knowledge description and querying	Knowledge operating system kernel

7.6.1 From Data Tuples to Proof Terms

At the Runtime layer, raw data such as `(Batch_202310-01, 10:00)` is merely passive information. During elaboration, however, it is encapsulated into Σ-terms with logical signatures.

Essence: From the Kernel's perspective, these terms are no longer simple "values," but small, composable function fragments. For example, the proof π_{route} carried by e_{proc} is itself a logical function capable of proving "route validity."

7.6.2 Dynamically Generated Causal Chains as Function Composition

What appears as the analyze function or the construction of a RootCauseReport is, in reality, the Kernel dynamically composing a new logical function based on real-time data.

Process: When the Kernel detects a match between a_{volt} and f_0, it does not simply connect them. Instead, it constructs a new lambda term:

$$\lambda(t).\ \mathsf{proof\_of\_overlap}(t, e_{proc}.\mathsf{dur})$$

Implication: The newly generated logical term π_{final} is a *specialized function*, tailored specifically to explain this particular hardness defect occurring at a specific time on a specific machine.

7.6.3 Division of Derivation Responsibilities Across the Three-Layer Architecture

This example illustrates how functions "flow" across different layers and become "materialized":

Table 11: Layers and the Essence of Derivation in the Bearing Case

Layer	**Essence of Derivation**	**Manifestation in the Bearing Case**
Core	Defines the "space" of functions	Defines the higher-order template mkCausalProof, specifying required input and output types.
Runtime	Provides the "operands" of functions	Extracts concrete values from SQL tables and elaborates them into logically meaningful atomic facts (e.g., f_0, a_{volt}).
Kernel	Performs function "evaluation"	Through Small-step semantics, fills fragmented data into Core-layer templates, materializing a concrete causal proof function R.

7.6.4 Why Is This Called "Derivation"?

In traditional software, adding a tracing logic such as "voltage fluctuation causes hardness non-uniformity" would require manually writing a function like `checkVoltage()`.

In KOS-TL, by contrast, one only defines general causal principles at the Core layer (e.g., $t(a) < t(f)$ together with physical correlation). When the specific voltage data for `Batch_202310-01` enters the system, the Kernel uses this data to *derive* a batch-specific, specialized proof function instance.

7.6.5 Conclusion

Functions are not pre-written and fixed; they are dynamically derived logical results based on the current system state σ and the observed fact f.

This example perfectly embodies the philosophy of *"Proof as Program"*: deriving a proof of a quality root cause is equivalent to deriving a logical function capable of explaining that failure. Data (voltage, time, batch) are no longer merely objects to be processed; they become the building blocks of the very *logical function* that explains the event.

8 Conclusion

8.1 Philosophical Paradigm: From "Theories of Truth" to "Executable Norms"

Since Frege, the core of traditional logic (first-order logic, description logic) has been static truth valuation ($\mathcal{M} \models \varphi$), aiming to characterize *what the world is like*. KOS-TL explicitly rejects this "single-model centrism" and realizes a fundamental transformation in the role of logic:

- **State-based Semantics**: The basic unit of semantics is no longer an eternal, immutable model, but a dynamically evolving sequence of states ($\sigma_0 \to \sigma_1 \to \dots$). The focus of logical judgment shifts from "whether a proposition is true" to "whether a state transition is valid."
- **Constructivist Stance**: Inheriting the essence of Martin-Löf Type Theory, the principle of "propositions as types, proofs as programs" is generalized to "knowledge as types, operations as programs, and events as constructors." In KOS-TL, the existence of knowledge is determined by whether it can be constructed, making the system philosophically self-consistent and transparent.
- **Unity of Knowing and Acting**: Logic is no longer merely descriptive, but becomes an executable normative system. It not only characterizes truth, but also specifies how knowledge is operated on, updated, and executed, bridging the gap between logical reasoning and physical action.

8.2 Logical Characteristics: The Fusion of Event-Driven Reasoning and Operational Semantics

The essential innovation of KOS-TL lies in introducing operational semantics into the logical core, thereby enabling "computation as reasoning":

- **Events as First-class Constructors**: Unlike the passive fact records of description logic, events in KOS-TL serve as the nexus between understanding and action. They are constructors in type theory, defining the legitimate mechanisms of object generation and providing a foundation for handling temporal sensitivity, causality, and state transitions.
- **Operational Derivation Rules**: Derivation rules are no longer purely truth-preserving; they become operational norms. They define how the system may *legally* produce new semantic annotations (e.g., risk alerts) under specific contextual and temporal constraints. This normative stance allows the system to exhibit different reasoning strategies across scenarios, with high flexibility and explainability.
- **Small-step Operational Semantics**: Logical judgment takes the formal shape $\langle \Sigma, c \rangle \to \Sigma'$. This fine-grained evolution enables KOS-TL to precisely manage monotonicity and resource consumption within the knowledge base, making it closer to a "causal reasoning computer."

8.3 System Capabilities: Computation-Reflective Autonomy and Auditability

Reflexivity in KOS-TL is elevated from a programming technique to a form of formal self-introspection:

- **Self-certifying Computation as Proof**: Based on the Curry–Howard correspondence, at every computational step (reduction), the kernel simultaneously synthesizes an equivalence proof (an Id proof). Thus, each step leaves an immutable logical footprint, certifying that system behavior conforms to predefined axioms.
- **End-to-end Logical Auditability**: With reflexivity, auditing no longer relies on external textual logs, but becomes a real-time mathematical verification process. The completeness of proof chains directly determines the legitimacy of system states, enforcing "transparent governance."
- **Logical Self-inspection and Healing**: Reflexivity allows the kernel to "look back" over decision paths, locate axiom conflicts when contradictions arise, and trigger self-healing operators, providing a foundation for autonomous system operation.

8.4 Engineering Paradigm: Expanding the Boundaries of Type-Driven Development

KOS-TL elevates the type system from a "memory safety tool" to a set of "axioms of system autonomy," offering key insights for modern system design:

- **From Type Safety to Logical Determinacy**: Through bidirectional physical–logical refinement types, physical laws and business rules are internalized as type properties. These "evidence-carrying types" ensure that illegal states are unrepresentable at the design level.

- **Persistent Dependent-Type Storage**: Breaking away from databases as raw byte heaps, storage becomes a runtime extension of the type system. Monotonicity constraints on types manage data lifecycles and ensure causal consistency in knowledge evolution.
- **Cross-layer Logical Lenses (Refinement Lenses)**: Through rigorous bidirectional refinement mappings, KOS-TL bridges the gap between high-level business entities (e.g., "compliant transfers") and low-level physical storage (e.g., SQL records), ensuring that every physical action faithfully reflects high-level logical intent.

8.5 The Essence of KOS-TL

KOS-TL is a logical system that seamlessly integrates the constructive semantics of intuitionistic type theory, operational semantics, and knowledge engineering practice. It is not merely a collection of algorithms, but a **"constitution for systems."**

It demonstrates that, through dependent types (Π and Σ), event constructors, and reflexivity, one can build an intelligent autonomous system that is logically self-consistent, causally traceable, and tightly aligned with the physical world. In this framework, proofs evolve from simple correctness checks into the central driving force behind decision-making in complex real-world systems.

References

@InCollectionDavidson1967, author = Davidson, Donald, title = The Logical Form of Action Sentences, booktitle = The Logic of Decision and Action, editor = Rescher, Nicholas, publisher = University of Pittsburgh Press, year = 1967, pages = 81–95 @bookMartinLof1984, author = Martin-Löf, Per, title = Intuitionistic Type Theory: Notes by Giovanni Sambin, publisher = Bibliopolis, year = 1984 @bookGirard1989, author = Girard, J.-Y. and Lafont, Y. and Taylor, P., title = Proofs and Types, publisher = Cambridge University Press, year = 1989