Typed Memory Structures for AI Agents: A Formal Framework Based on ContextFS Principles

Matthew Long
Independent Researcher, Chicago, IL
`matthew@yonedaai.com`

The YonedaAI Collaboration
YonedaAI Research Collective

January 8, 2026

Abstract

We present a formal type-theoretic framework for AI memory systems grounded in three foundational principles derived from the ContextFS architecture: (1) schemas shape reasoning—the structure of memory necessarily constrains the structure of possible inferences; (2) types enforce validity—invalid memory states should be unrepresentable in the type system rather than detected post-hoc; and (3) versioning preserves agent continuity—the causal structure of belief revision must be maintained to enable reasoning about why memory changed, not merely that it did. We formalize these principles using dependent type theory, establishing memory types as schema-indexed structures where inhabitants carry proofs of invariant satisfaction. We prove key theorems including the Unrepresentability Theorem (invalid states cannot be constructed), the Schema-Reasoning Correspondence (memory structure determines inference space), and Memory Compositionality (valid memories compose to valid memories). The framework provides compile-time guarantees about memory validity, sound inference over memory contents, explainable belief revision through derivation tracking, and compositional memory architecture. We demonstrate the practical applicability through a complete implementation architecture mapping to TypeScript and provide example schema libraries for common AI agent memory patterns.

Contents

1 Introduction

1.1 The Crisis of Untyped Memory

Contemporary AI systems, particularly large language models and autonomous agents, increasingly rely on persistent memory to maintain context across interactions, track beliefs about the world, and preserve continuity of identity. Yet the predominant architectural approaches to AI memory suffer from a fundamental flaw: they treat memory as *data* rather than *structure*.

Consider the landscape of current implementations:

- **Vector stores**: Embeddings with metadata—no schema, no type safety, semantic similarity as the sole organizational principle
- **Key-value stores**: Arbitrary strings mapping to arbitrary values, with implicit structure enforced only by convention
- **File systems**: Unstructured text with naming conventions, where "memory" is conflated with "storage"
- **Retrieval-Augmented Generation (RAG)**: Chunk-based retrieval with no guarantees about consistency or completeness

These approaches share a common failure mode: they permit the construction of invalid memory states. A vector store can contain contradictory embeddings. A key-value store can reference nonexistent entities. A file system can hold malformed data that violates implicit invariants. The invalidity is discovered—if at all—only at runtime, when the corrupted memory produces erroneous behavior.

This is not merely an engineering inconvenience. It reflects a deeper conceptual confusion about the nature of memory in cognitive systems. Memory is not passive storage; it is the substrate upon which reasoning operates. The structure of memory shapes what can be thought, what can be inferred, what can be remembered and why.

1.2 The ContextFS Principles

The ContextFS project introduces a paradigm shift in how we conceptualize AI memory. Rather than treating memory as files to be read and written, ContextFS treats memory as a typed, versioned structure where validity is guaranteed by construction. Three principles emerge as foundational:

> **Principle 1 (Schemas)**: The structure of memory shapes the structure of reasoning.
>
> **Principle 2 (Types)**: Invalid memory states should be unrepresentable, not just "files with bad data."
>
> **Principle 3 (Versioning)**: Agent continuity means something—you might need to reason about *why* a memory changed, not just *that* it did.

These principles suggest that memory should be formalized as a typed, versioned structure where the type system itself enforces memory validity. This paper develops that formalization rigorously.

1.3 Contributions

This paper makes the following contributions:

1. We formalize AI memory as a dependent type indexed by schemas, proving that invalid memory states are unrepresentable by construction (Section 2).

2. We establish the Schema-Reasoning Correspondence, showing how memory structure determines the space of valid inferences (Section 3).
3. We develop a theory of versioned memory that preserves agent continuity through causal derivation tracking (Section 4).
4. We present the complete ContextFS type system with formation, introduction, and elimination rules (Section 5).
5. We prove Memory Compositionality, enabling modular and hierarchical memory architectures (Section 6).
6. We provide a complete implementation architecture with TypeScript bindings and example schema libraries (Section 7).

1.4 Related Work

Our work draws on several traditions. From dependent type theory [1, 2], we take the insight that types can express rich invariants and that proofs can be data. From the Curry-Howard correspondence [6, 7], we understand that our memory types correspond to propositions and their inhabitants to proofs of validity. From belief revision theory [8, 9], we adopt the concern for rational belief change and the importance of tracking revision history.

The application of type theory to data management has precedent in dependently-typed databases [10] and schema evolution [11]. Our contribution is to extend these ideas specifically to AI memory systems, where the cognitive interpretation—memory as the substrate of reasoning—motivates additional requirements around continuity and derivation.

Version control systems, particularly Git, provide inspiration for our versioning model, but our framework differs in tracking not just *what* changed but *why*, with change reasons that capture epistemic provenance.

1.5 Paper Organization

The remainder of this paper is organized as follows. Section 2 establishes the formal foundations, including the category of memory types and the Unrepresentability Theorem. Section 3 develops the Schema-Reasoning Correspondence. Section 4 presents versioned memory and agent continuity. Section 5 gives the complete type system. Section 6 proves compositionality results. Section 7 provides implementation architecture. Section 8 discusses theoretical implications. Section 9 compares with existing approaches. Section 10 outlines future work, and Section 11 concludes.

2 Formal Foundations

2.1 The Category of Memory Types

We begin by establishing the categorical context for our formalization.

Definition 2.1 (Category **Mem**). Let **Mem** be a category where:

- **Objects** are memory types M
- **Morphisms** $f : M \to M'$ are memory transformations that preserve validity invariants
- **Identity** $\mathrm{id}_M : M \to M$ is the identity transformation
- **Composition** $(g \circ f) : M \to M''$ for $f : M \to M'$ and $g : M' \to M''$

The key insight is that morphisms in **Mem** are not arbitrary functions—they must preserve the validity invariants encoded in the types. This categorical structure will later support composition and schema evolution.

Remark 2.1. The category **Mem** is a subcategory of **Set**, but with restricted morphisms. This restriction is what gives the category its computational significance: morphisms correspond to *safe* memory transformations.

2.2 Memory Types as Dependent Types

Definition 2.2 (Memory Type)**.** A memory type M is a dependent type:

$$M : \mathsf{Schema} \to \mathsf{Type}$$

where Schema specifies the structure and Type is the universe of valid inhabitants.

The dependency on Schema is crucial: it ensures that memory structure is not incidental but *constitutive* of the memory type itself. Two memories with different schemas are different types, even if their underlying data representations coincide.

Definition 2.3 (Schema)**.** A schema S is a record type consisting of:

$$\mathsf{Schema} := \left\{ \begin{array}{ll} \mathsf{Fields} & : \mathsf{List}(\mathsf{Name} \times \mathsf{Type}) \\ \mathsf{Invariants} & : \mathsf{List}(\mathsf{Constraint\ Fields}) \\ \mathsf{Relations} & : \mathsf{List}(\mathsf{Relation\ Fields}) \end{array} \right\}$$

The components serve distinct purposes:

- Fields specifies the data content—what information the memory holds
- $\mathsf{Invariants}$ specifies validity constraints—what configurations are meaningful
- $\mathsf{Relations}$ specifies structural relationships—how fields depend on each other

Definition 2.4 (Constraint)**.** A constraint over fields F is a predicate:

$$\mathsf{Constraint}\ F := \mathsf{Record}\ F \to \mathsf{Prop}$$

where Prop is the universe of propositions.

Definition 2.5 (Relation)**.** A relation over fields F is a specification of dependencies:

$$\mathsf{Relation}\ F := \{\mathsf{from} : \mathsf{FieldRef}\ F, \mathsf{to} : \mathsf{FieldRef}\ F, \mathsf{kind} : \mathsf{RelationKind}\}$$

where $\mathsf{RelationKind} \in \{\mathsf{references}, \mathsf{depends\_on}, \mathsf{implements}, \ldots\}$.

2.3 The Unrepresentability Principle

The central technical contribution of typed memory is the *unrepresentability* of invalid states. This is achieved by defining memory types as dependent pairs (sigma types).

Definition 2.6 (Memory Inhabitant)**.** For a schema S, the type $\mathsf{Mem}[S]$ is defined as:

$$\mathsf{Mem}[S] := \Sigma\ (data : \mathsf{Record}\ S.\mathsf{Fields}).\ \mathsf{AllSat}\ S.\mathsf{Invariants}\ data$$

where $\mathsf{AllSat}\ cs\ d$ is the proposition that all constraints in cs are satisfied by d.

Definition 2.7 (All Satisfied)**.** For constraints $cs = [c_1, \ldots, c_n]$ and data d:

$$\mathsf{AllSat}\ cs\ d := c_1(d) \wedge c_2(d) \wedge \cdots \wedge c_n(d)$$

Any inhabitant of $\mathsf{Mem}[S]$ consists of two components: the data itself, and a *proof* that all invariants hold. This proof is not optional metadata—it is part of the type, required for construction.

Theorem 2.1 (Invalid State Unrepresentability)**.** *For a well-formed memory type* $\mathsf{Mem}[S]$*, there exists no term* $m : \mathsf{Mem}[S]$ *such that* m *violates any constraint in* $S.\mathsf{Invariants}$*.*

Proof. By construction of the type $\mathsf{Mem}[S]$. Suppose for contradiction that there exists $m : \mathsf{Mem}[S]$ with $m = (\mathit{data}, \mathit{pf})$ such that some constraint $c \in S.\mathsf{Invariants}$ is violated by data.

By Definition 2.6, pf has type $\mathsf{AllSat}\ S.\mathsf{Invariants}\ \mathit{data}$. By Definition 2.7, this requires $c(\mathit{data})$ to hold for all $c \in S.\mathsf{Invariants}$.

But we assumed $c(\mathit{data})$ does not hold, so pf cannot exist, contradicting the assumption that m is well-typed. □

This theorem is the cornerstone of our framework. It transforms memory validity from a runtime property ("check if the data is valid") to a static property ("the type ensures validity").

Example 2.1 (Entity Reference Integrity)**.** Consider a memory schema for task tracking:

$$\begin{aligned}
\mathsf{TaskSchema} \coloneqq \{ & \\
\mathsf{Fields} : \ & [(\mathsf{id}, \mathsf{TaskId}), \\
& (\mathsf{assignee}, \mathsf{UserId}), \\
& (\mathsf{status}, \mathsf{Status}), \\
& (\mathsf{completed\_at}, \mathsf{Option\ DateTime})], \\
\mathsf{Invariants} : \ & [\lambda t.\ (t.\mathsf{status} = \mathsf{Completed}) \Leftrightarrow (\mathsf{isSome}\ t.\mathsf{completed\_at}), \\
& \lambda t.\ t.\mathsf{assignee} \in \mathsf{Users}]\ \}
\end{aligned}$$

A value of type $\mathsf{Mem}[\mathsf{TaskSchema}]$ *cannot* represent:

- A task marked complete without a completion timestamp
- A task assigned to a nonexistent user

The type system rejects such constructions at compile time.

2.4 Proof-Relevant Memory

An important feature of our formalization is that memory is *proof-relevant*: the proofs of invariant satisfaction are retained as part of the memory value, not discarded after type-checking.

Definition 2.8 (Proof-Relevant Memory)**.** A memory value $m : \mathsf{Mem}[S]$ carries:

1. The data $\pi_1(m) : \mathsf{Record}\ S.\mathsf{Fields}$
2. The validity proof $\pi_2(m) : \mathsf{AllSat}\ S.\mathsf{Invariants}\ (\pi_1(m))$

Both components are accessible and can be used in subsequent computation.

The proof-relevance enables meta-reasoning: an agent can not only access its memories but also access the *justification* for why those memories are valid.

Proposition 2.2 (Proof Accessibility)**.** *For any* $m : \mathsf{Mem}[S]$ *and any invariant* $c \in S.\mathsf{Invariants}$*, there exists a witness* w *such that* w *proves* $c(\pi_1(m))$*.*

Proof. By Definition 2.8, $\pi_2(m)$ has type $\mathsf{AllSat}\ S.\mathsf{Invariants}\ (\pi_1(m))$. By Definition 2.7, this is a conjunction including $c(\pi_1(m))$. The witness w is the projection of $\pi_2(m)$ onto the c component. □

3 Schemas and Reasoning

3.1 The Schema-Reasoning Correspondence

We now establish the fundamental relationship between memory structure and inference capability.

Proposition 3.1 (Schema-Reasoning Correspondence)**.** *The structure of schema S determines the space of valid inferences over* $\mathsf{Mem}[S]$*.*

To make this precise, we define the type of valid inferences.

Definition 3.1 (Inference Type)**.** For a schema S, define:

$$\mathsf{Inference}[S] := (m : \mathsf{Mem}[S]) \to (\mathsf{query} : \mathsf{Query}[S]) \to \mathsf{Answer}[S]$$

where $\mathsf{Query}[S]$ is the type of well-formed queries over S and $\mathsf{Answer}[S]$ is the type of valid answers.

The schema constrains both dimensions:

1. **What can be queried**: Only fields in $S.\mathsf{Fields}$ can be referenced. A query about a nonexistent field is ill-typed.
2. **What invariants can be assumed**: Inference rules can freely assume that $S.\mathsf{Invariants}$ hold, enabling sound derivations that would be unsound for untyped memory.

Example 3.1 (Inference from Structure)**.** Given a conversation memory schema:

$$\begin{aligned}\mathsf{ConversationSchema} := \{ \\ \mathsf{Fields} : {} & [(\mathsf{messages}, \mathsf{List\ Message}), \\ & (\mathsf{participants}, \mathsf{NonEmpty\ (Set\ UserId)}), \\ & (\mathsf{last\_activity}, \mathsf{DateTime})], \\ \mathsf{Invariants} : {} & [\lambda c.\ \mathsf{length}\ c.\mathsf{messages} > 0 \Rightarrow \\ & \quad c.\mathsf{last\_activity} \geq \mathsf{max}\ (\mathsf{map\ timestamp}\ c.\mathsf{messages})]\ \}\end{aligned}$$

The schema enables the inference rule:

$$\frac{c : \mathsf{Mem}[\mathsf{ConversationSchema}] \quad c.\mathsf{messages} \neq []}{c.\mathsf{last\_activity} \geq \max(\mathsf{timestamps}(c.\mathsf{messages}))}\ \text{LastActivity-Sound}$$

This inference is *sound by construction*—no runtime check is needed because the type system guarantees the invariant holds.

3.2 Query Languages and Type Safety

The Schema-Reasoning Correspondence suggests that query languages over typed memory should themselves be typed.

Definition 3.2 (Typed Query)**.** For schema S, a typed query has the form:

$$\mathsf{Query}[S] := \Sigma\ (f : \mathsf{Name}).\ f \in S.\mathsf{Fields}$$

This definition ensures that queries can only reference fields that exist. An attempt to query a nonexistent field is a type error, caught at compile time.

Lemma 3.2 (Query Totality)**.** *Every well-typed query over* $\mathsf{Mem}[S]$ *produces a well-typed answer.*

Proof. Let $q : \mathsf{Query}[S]$ and $m : \mathsf{Mem}[S]$. By Definition 3.2, $q = (f, pf)$ where pf proves $f \in S.\mathsf{Fields}$.

Since $\pi_1(m) : \mathsf{Record}\ S.\mathsf{Fields}$, the field f is guaranteed to exist in $\pi_1(m)$. Projection is therefore total, and the answer $\pi_1(m).f$ is well-typed with type $S.\mathsf{Fields}[f]$. □

3.3 Inference Rules as Types

Following the Curry-Howard correspondence, we can represent inference rules as types.

Definition 3.3 (Inference Rule Type)**.** An inference rule over schema S is a type of the form:

$$\mathsf{Rule} := \Pi\ (m : \mathsf{Mem}[S]).\ P(m) \to Q(m)$$

where P and Q are predicates over memories.

An inhabitant of this type is a *proof* of the rule's soundness—a function that transforms evidence for $P(m)$ into evidence for $Q(m)$ for any memory m.

Example 3.2 (Soundness Proof as Program)**.** Consider an inference rule: "If a task is assigned and overdue, it requires attention."

$$\mathsf{RequiresAttention} : \Pi\ (t : \mathsf{Mem}[\mathsf{TaskSchema}]).$$

$$(\mathsf{isAssigned}(t) \wedge \mathsf{isOverdue}(t)) \to \mathsf{NeedsAttention}(t)$$

An implementation of this type is a constructive proof of the rule's validity.

3.4 The Inference Lattice

Different schemas induce different inference capabilities. We can partially order schemas by their inference power.

Definition 3.4 (Schema Ordering)**.** Define $S_1 \preceq S_2$ ("S_1 is weaker than S_2") iff every inference valid over $\mathsf{Mem}[S_1]$ is also valid over $\mathsf{Mem}[S_2]$.

Proposition 3.3 (Inference Lattice)**.** *The relation $\preceq$ forms a partial order on schemas, with:*

- *Join: $S_1 \sqcup S_2$ is the schema enabling inferences valid in either*
- *Meet: $S_1 \sqcap S_2$ is the schema enabling only inferences valid in both*

This lattice structure allows reasoning about the relative expressiveness of different memory organizations.

4 Versioned Memory and Agent Continuity

4.1 The Temporal Structure of Belief

Memory is not static. Beliefs evolve as agents receive new information, make inferences, correct errors, and forget irrelevant details. Agent continuity requires not just the current state but the *history* of how beliefs changed.

This requirement goes beyond simple version control. We need to track not only *what* changed but *why*—the epistemic provenance of each memory state.

Definition 4.1 (Versioned Memory)**.** A versioned memory over schema S is a structure:

$$\begin{aligned}\mathsf{Versioned}[\mathsf{Mem}[S]] := \{ & \\ & \mathsf{Timeline} : \mathsf{List}\ (\mathsf{Timestamped}\ (\mathsf{Mem}[S] \times \mathsf{ChangeReason})) \\ & \mathsf{current} : \mathsf{Mem}[S] \\ & \mathit{proof} : \mathsf{last}\ \mathsf{Timeline} = \mathsf{current}\ \}\end{aligned}$$

The Timeline records the complete history of memory states, each paired with the reason for the transition. The current field provides efficient access to the present state. The *proof* field ensures consistency between timeline and current state.

Definition 4.2 (Change Reason)**.** The type ChangeReason captures why memory changed:

$$\begin{aligned}\mathsf{ChangeReason} := &\mid \mathsf{Observation}\ (\mathsf{source} : \mathsf{Source},\ \mathsf{evidence} : \mathsf{Evidence})\\ &\mid \mathsf{Inference}\ (\mathsf{rule} : \mathsf{InferenceRule},\ \mathsf{premises} : \mathsf{List}\ \mathsf{Mem}[S])\\ &\mid \mathsf{Correction}\ (\mathsf{previous} : \mathsf{Mem}[S],\ \mathsf{reason} : \mathsf{String})\\ &\mid \mathsf{Decay}\ (\mathsf{factor} : \mathsf{Float} \to \mathsf{Float})\end{aligned}$$

These constructors capture the fundamental modes of belief change:

- Observation: Memory changed because of new sensory or informational input
- Inference: Memory changed because of an inference from existing beliefs
- Correction: Memory changed because a previous state was recognized as erroneous
- Decay: Memory changed through gradual confidence reduction (forgetting)

4.2 Derivation Trees

The history of memory states forms a derivation structure that captures the causal dependencies between beliefs.

Definition 4.3 (Memory Derivation)**.** A derivation D of memory state m is a tree where:

- **Nodes** are memory states m_i : $\mathsf{Mem}[S]$
- **Edges** (m_i, m_j) are labeled with ChangeReason values
- **Root** is the initial memory state m_0
- **Leaves** include the current state m

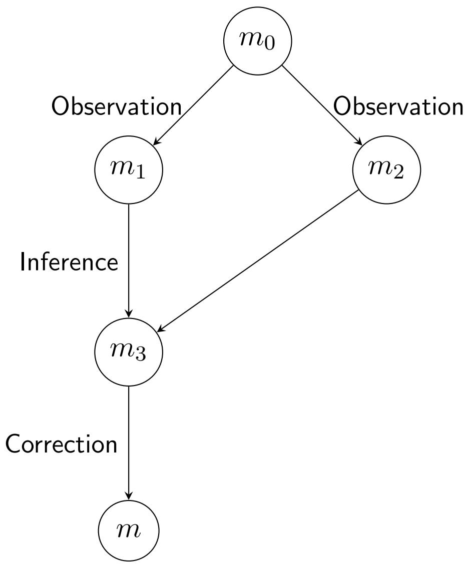
$$\begin{tikzcd} & m_0 \arrow[dl, "\mathsf{Observation}"'] \arrow[dr, "\mathsf{Observation}"] & \\ m_1 \arrow[d, "\mathsf{Inference}"'] & & m_2 \arrow[dll] \\ m_3 \arrow[d, "\mathsf{Correction}"'] & & \\ m & & \end{tikzcd}$$

Figure 1: A memory derivation tree showing how state m derives from initial state m_0 through observations, inference, and correction.

Theorem 4.1 (Derivation Soundness). *For any derivation D of $m : \mathsf{Mem}[S]$, every intermediate state m_i in D is also of type $\mathsf{Mem}[S]$.*

Proof. By induction on the structure of derivation D.

Base case: The root $m_0 : \mathsf{Mem}[S]$ by assumption (initial state is well-typed).

Inductive case: Suppose $m_i : \mathsf{Mem}[S]$ and there is an edge (m_i, m_j) labeled with change reason r. We must show $m_j : \mathsf{Mem}[S]$.

By definition of ChangeReason, each constructor produces a valid memory transformation:

- Observation: The observation handler must produce $\mathsf{Mem}[S]$ (by typing of observation handlers)
- Inference: Inference rules are typed as $\mathsf{Mem}[S] \to \mathsf{Mem}[S]$
- Correction: Corrections replace one valid state with another valid state
- Decay: Decay transforms confidence values while preserving type validity

In all cases, $m_j : \mathsf{Mem}[S]$. □

This theorem ensures that the entire history of an agent's memory consists of valid states—there are no "corrupted intermediate states" hidden in the timeline.

4.3 Reasoning About Memory Change

The versioning structure enables powerful meta-reasoning capabilities.

Definition 4.4 (Belief Revision Query). A belief revision query has the form:

$$\mathsf{WhyChanged} : (f : \mathsf{Name}) \to (v_1 : \mathsf{Value}) \to (v_2 : \mathsf{Value}) \to$$

$$\mathsf{Versioned}[\mathsf{Mem}[S]] \to \mathsf{Option\ ChangeReason}$$

Example 4.1 (Tracing Belief Change). Suppose an agent's memory includes project information:

$$\begin{aligned}\mathsf{ProjectMemory} := \{ \\ \mathsf{Fields} : [&(\mathsf{deadline}, \mathsf{DateTime}), \\ &(\mathsf{confidence}, \mathsf{Float}), \\ &(\mathsf{sources}, \mathsf{List\ Source})]\ \}\end{aligned}$$

If deadline changes from January 15 to January 22, the agent can query:

$$\begin{aligned}&\mathsf{WhyChanged}\ \text{"deadline"}\ (\text{Jan } 15)\ (\text{Jan } 22)\ \textit{memory} \\ &\quad \mapsto \mathsf{Some}\ (\mathsf{Observation}\ (\text{"email from PM"}, \text{"reschedule notice"}))\end{aligned}$$

This enables natural language explanations: "I updated the deadline because I received an email from the project manager announcing a schedule change."

4.4 Causal Consistency

Versioned memory must maintain causal consistency—the recorded history must reflect actual causal relationships.

Definition 4.5 (Causal Consistency). A versioned memory $v : \mathsf{Versioned}[\mathsf{Mem}[S]]$ is causally consistent iff for every edge (m_i, m_j) in its derivation with label r:

- If $r = \mathsf{Observation}(s, e)$, then evidence e from source s justifies the change from m_i to m_j
- If $r = \mathsf{Inference}(\rho, ps)$, then m_j follows from applying rule ρ to premises ps
- If $r = \mathsf{Correction}(m_i, \text{reason})$, then m_j corrects an identified error in m_i
- If $r = \mathsf{Decay}(f)$, then m_j is m_i with confidence values transformed by f

Theorem 4.2 (Causal Closure). *If versioned memory v is causally consistent, then every belief in $\mathsf{current}(v)$ has a traceable causal chain to observations or axioms.*

Proof. By induction on the derivation tree. Each node either is the root (an axiom/initial state) or has an incoming edge with a justified ChangeReason. Following these edges backward from any belief eventually reaches the root. □

4.5 Agent Identity Through Time

Versioned memory provides a formal account of agent identity that persists through belief change.

Definition 4.6 (Agent Continuity). An agent exhibits *continuity* over time interval $[t_1, t_2]$ iff:

1. Its memory evolution forms a valid derivation (Theorem 4.1)
2. Each transition has a justified ChangeReason (Definition 4.5)
3. The derivation is causally closed (Theorem 4.2)

This definition rules out several pathological cases:

- **Discontinuous belief change**: Beliefs appearing without cause (no incoming edge in derivation)
- **Incoherent belief states**: Memory violating invariants (ruled out by type system)
- **Untraceable reasoning**: Cannot explain why beliefs changed (no ChangeReason labels)

5 The ContextFS Type System

We now present the complete type system for ContextFS memory.

5.1 Type Grammar

Definition 5.1 (Type Grammar).

$$
\begin{aligned}
\mathsf{BaseType} &::= \mathsf{String} \mid \mathsf{Int} \mid \mathsf{Float} \mid \mathsf{Bool} \mid \mathsf{DateTime} \mid \mathsf{UUID} \\
\mathsf{EntityType} &::= \mathsf{Entity\ Name\ Schema} \\
\mathsf{RefType} &::= \mathsf{Ref\ EntityType} \\
\mathsf{OptionType} &::= \mathsf{Option\ Type} \\
\mathsf{ListType} &::= \mathsf{List\ Type} \\
\mathsf{SetType} &::= \mathsf{Set\ Type}\ \text{where}\ \mathsf{Ord\ Type} \\
\mathsf{MapType} &::= \mathsf{Map\ KeyType\ ValueType} \\
\mathsf{UnionType} &::= \mathsf{Type}_1 \mid \mathsf{Type}_2 \mid \cdots \mid \mathsf{Type}_n \\
\mathsf{RecordType} &::= \{f_1 : \mathsf{Type}_1, \ldots, f_n : \mathsf{Type}_n\} \\
\mathsf{MemoryType} &::= \mathsf{Mem\ Schema} \\
\mathsf{VersionedType} &::= \mathsf{Versioned\ MemoryType}
\end{aligned}
$$

5.2 Formation Rules

Formation rules specify when a type expression is well-formed.

$$\frac{S : \mathsf{Schema}}{\mathsf{Mem}[S] : \mathsf{Type}}\ \text{Mem-Form}$$

$$\frac{M : \mathsf{MemoryType}}{\mathsf{Versioned}[M] : \mathsf{Type}}\ \text{Ver-Form}$$

$$\frac{\mathsf{Fields} : \mathsf{List}(\mathsf{Name} \times \mathsf{Type}) \quad \mathsf{Invariants} : \mathsf{List}(\mathsf{Constraint\ Fields})}{S : \mathsf{Schema}}\ \text{Schema-Form}$$

$$\frac{E : \mathsf{EntityType}}{\mathsf{Ref}\ E : \mathsf{Type}}\ \text{Ref-Form}$$

5.3 Introduction Rules

Introduction rules specify how to construct values of a type.

$$\frac{data : \mathsf{Record}\ S.\mathsf{Fields} \quad pf : \mathsf{AllSat}\ S.\mathsf{Invariants}\ data}{\mathsf{MkMem}\ data\ pf : \mathsf{Mem}[S]}\ \text{Mem-Intro}$$

$$\frac{m : \mathsf{Mem}[S] \quad h : \mathsf{List}\ (\mathsf{Timestamped}\ \Delta) \quad \mathsf{last}(h) = m}{\mathsf{MkVer}\ m\ h : \mathsf{Versioned}[\mathsf{Mem}[S]]}\ \text{Ver-Intro}$$

$$\frac{s : \mathsf{Source} \quad e : \mathsf{Evidence}}{\mathsf{Observation}\ s\ e : \mathsf{ChangeReason}}\ \text{Obs-Intro}$$

$$\frac{\rho : \mathsf{InferenceRule} \quad ps : \mathsf{List\ Mem}[S]}{\mathsf{Inference}\ \rho\ ps : \mathsf{ChangeReason}}\ \text{Inf-Intro}$$

5.4 Elimination Rules

Elimination rules specify how to use values of a type.

$$\frac{m : \mathsf{Mem}[S] \quad f \in \mathsf{dom}(S.\mathsf{Fields})}{m.f : S.\mathsf{Fields}[f]} \text{ Mem-Elim}$$

$$\frac{v : \mathsf{Versioned}[M]}{\mathsf{current}\ v : M} \text{ Ver-Elim-Current}$$

$$\frac{v : \mathsf{Versioned}[M]}{\mathsf{history}\ v : \mathsf{Timeline}} \text{ Ver-Elim-History}$$

$$\frac{m : \mathsf{Mem}[S]}{\mathsf{validityProof}\ m : \mathsf{AllSat}\ S.\mathsf{Invariants}\ (\pi_1(m))} \text{ Proof-Elim}$$

5.5 Computation Rules

Computation rules specify how elimination interacts with introduction (beta reduction).

$$(\mathsf{MkMem}\ data\ pf).f \equiv data.f$$

$$\mathsf{current}\ (\mathsf{MkVer}\ m\ h) \equiv m$$

$$\mathsf{history}\ (\mathsf{MkVer}\ m\ h) \equiv h$$

5.6 Subtyping and Schema Evolution

Schemas evolve over time as requirements change. We need a notion of safe schema evolution.

Definition 5.2 (Schema Subtyping)**.** Schema S' is a subtype of S (written $S' <: S$) iff:

1. $S'.\mathsf{Fields} \supseteq S.\mathsf{Fields}$ (extension with new fields)
2. $S'.\mathsf{Invariants} \supseteq S.\mathsf{Invariants}$ (additional constraints)
3. $S'.\mathsf{Relations} \supseteq S'.\mathsf{Relations}$ (additional relations)

Theorem 5.1 (Safe Schema Evolution)**.** *If $S' <: S$ then there exists a safe projection:*

$$\mathsf{project} : \mathsf{Mem}[S'] \to \mathsf{Mem}[S]$$

Proof. Let $m' : \mathsf{Mem}[S']$ with $m' = (data', pf')$.

Define $data = data'|_{S.\mathsf{Fields}}$ (restriction to S's fields). This is well-defined because $S'.\mathsf{Fields} \supseteq S.\mathsf{Fields}$.

For each constraint $c \in S.\mathsf{Invariants}$, we have $c \in S'.\mathsf{Invariants}$ (by subtyping). Since pf' proves all constraints in $S'.\mathsf{Invariants}$ hold for $data'$, it proves c holds for $data'$.

Since c only references fields in $S.\mathsf{Fields}$ and $data = data'|_{S.\mathsf{Fields}}$, we have $c(data)$ holds.

Thus $\mathsf{MkMem}\ data\ pf : \mathsf{Mem}[S]$ where pf is constructed from the relevant projections of pf'. □

This theorem enables schema migration: data in the new schema can be safely projected to the old schema, preserving type safety.

6 Memory Composition

6.1 The Compositional Property

Real agent memories are not monolithic—they consist of multiple interacting subsystems. We need compositional operators that preserve type safety.

Definition 6.1 (Schema Product)**.** For schemas S_1 and S_2 with disjoint field names, define:

$$S_1 \times S_2 := \left\{ \begin{array}{l} \mathsf{Fields} : S_1.\mathsf{Fields} \cup S_2.\mathsf{Fields} \\ \mathsf{Invariants} : S_1.\mathsf{Invariants} \cup S_2.\mathsf{Invariants} \\ \mathsf{Relations} : S_1.\mathsf{Relations} \cup S_2.\mathsf{Relations} \end{array} \right\}$$

Theorem 6.1 (Memory Compositionality)**.** *If $m_1 : \mathsf{Mem}[S_1]$ and $m_2 : \mathsf{Mem}[S_2]$ are valid memories over schemas with disjoint field names, their composition:*

$$m_1 \otimes m_2 : \mathsf{Mem}[S_1 \times S_2]$$

is also valid.

Proof. Let $m_1 = (data_1, pf_1)$ and $m_2 = (data_2, pf_2)$.

Define $data = data_1 \cup data_2$ (disjoint union of fields).

For invariants in $S_1.\mathsf{Invariants}$: each c only references fields in $S_1.\mathsf{Fields}$, which are present in $data$. Since pf_1 proves $c(data_1)$ and $data|_{S_1.\mathsf{Fields}} = data_1$, we have $c(data)$.

Symmetrically for invariants in $S_2.\mathsf{Invariants}$.

Thus all invariants in $(S_1 \times S_2).\mathsf{Invariants} = S_1.\mathsf{Invariants} \cup S_2.\mathsf{Invariants}$ are satisfied.

Therefore $m_1 \otimes m_2 := \mathsf{MkMem}\ data\ pf : \mathsf{Mem}[S_1 \times S_2]$ is well-typed. □

6.2 Applications of Compositionality

The compositionality theorem enables several architectural patterns:

1. **Modular memory design**: Domain-specific memories (episodic, semantic, working) can be developed independently and composed.
2. **Safe memory sharing**: Multiple agents with compatible schemas can share memory components.
3. **Hierarchical structures**: Memories can be nested through repeated composition.

Example 6.1 (Composite Agent Memory)**.** An agent might have composite memory:

$$\mathsf{AgentMemory} = \mathsf{IdentityMem} \times \mathsf{EpisodicMem} \times \mathsf{SemanticMem} \times \mathsf{WorkingMem}$$

Each component is developed and validated independently; composition preserves all guarantees.

6.3 Memory Morphisms

Definition 6.2 (Memory Morphism)**.** A memory morphism $\phi : \mathsf{Mem}[S_1] \to \mathsf{Mem}[S_2]$ is a function that:

1. Transforms data: $\phi_{data} : \mathsf{Record}\ S_1.\mathsf{Fields} \to \mathsf{Record}\ S_2.\mathsf{Fields}$
2. Preserves validity: if $\mathsf{AllSat}\ S_1.\mathsf{Invariants}\ d$ then $\mathsf{AllSat}\ S_2.\mathsf{Invariants}\ (\phi_{data}(d))$

Proposition 6.2 (Morphism Composition)**.** *Memory morphisms compose: if $\phi : \mathsf{Mem}[S_1] \to \mathsf{Mem}[S_2]$ and $\psi : \mathsf{Mem}[S_2] \to \mathsf{Mem}[S_3]$, then $\psi \circ \phi : \mathsf{Mem}[S_1] \to \mathsf{Mem}[S_3]$.*

Proof. Let $m_1 : \mathsf{Mem}[S_1]$. Then $\phi(m_1) : \mathsf{Mem}[S_2]$ by assumption on ϕ. Then $\psi(\phi(m_1)) : \mathsf{Mem}[S_3]$ by assumption on ψ. Define $(\psi \circ \phi)(m_1) = \psi(\phi(m_1))$. □

This confirms that **Mem** is indeed a category (Definition 2.1).

6.4 Functorial Properties

Proposition 6.3 (Versioning is a Functor). *The operation* $\mathsf{Versioned}[-]$ *extends to a functor* $\mathsf{Versioned} : \mathbf{Mem} \to \mathbf{Mem}$.

Proof. For objects: $\mathsf{Versioned}$ maps memory types to versioned memory types.

For morphisms: Given $\phi : \mathsf{Mem}[S_1] \to \mathsf{Mem}[S_2]$, define $\mathsf{Versioned}(\phi) : \mathsf{Versioned}[\mathsf{Mem}[S_1]] \to \mathsf{Versioned}[\mathsf{Mem}[S_2]]$ by applying ϕ to each state in the timeline and to the current state.

Identity preservation: $\mathsf{Versioned}(\mathrm{id}) = \mathrm{id}$ (applying identity pointwise is identity).

Composition preservation: $\mathsf{Versioned}(\psi \circ \phi) = \mathsf{Versioned}(\psi) \circ \mathsf{Versioned}(\phi)$ (composing pointwise applications). □

7 Implementation Architecture

7.1 The ContextFS Memory Store Interface

We now present a practical implementation mapping our formal framework to TypeScript.

Listing 1: ContextFS Store Interface

```typescript
interface ContextFSStore<S extends Schema> {
  // Type-safe field operations
  get<K extends keyof S['fields']>(key: K): S['fields'][K];
  set<K extends keyof S['fields']>(
    key: K,
    value: S['fields'][K]
  ): void;

  // Invariant-checked bulk update
  update(
    patch: Partial<S['fields']>
  ): Result<void, InvariantViolation>;

  // Versioned operations
  commit(reason: ChangeReason): Version;
  checkout(version: Version): Memory<S>;

  // Provenance queries
  whyChanged(field: keyof S['fields']): ChangeHistory;
  derivation(): DerivationTree;
}
```

The interface enforces type safety: the `get` and `set` methods are parameterized by field name, and the return/argument types are automatically inferred from the schema.

7.2 Schema Definition Language

Listing 2: Schema Definition

```typescript
const AgentMemorySchema = defineSchema({
  name: 'AgentMemory',
  fields: {
    identity: { type: 'string', required: true },
    beliefs: {
      type: 'map',
      keyType: 'string',
      valueType: 'BeliefRecord'
    },
    goals: { type: 'list', elementType: 'Goal' },
    context: { type: 'ref', target: 'ConversationContext' }
  },
  invariants: [
    // At least one goal must exist
    (m) => m.goals.length > 0,
    // Belief confidence must be in [0, 1]
    (m) => Object.values(m.beliefs)
             .every(b => b.confidence >= 0 &&
                         b.confidence <= 1)
  ]
});

// Type is inferred from schema
type AgentMemory = InferMemory<typeof AgentMemorySchema>;
```

The InferMemory type-level function extracts the memory type from the schema definition, ensuring that the TypeScript type system tracks our invariants.

7.3 Versioning Implementation

Listing 3: Versioned Memory Implementation

```
class VersionedMemory<S extends Schema> {
  private timeline: Array<{
    timestamp: DateTime,
    state: Memory<S>,
    reason: ChangeReason,
    hash: Hash
  }>;

  commit(state: Memory<S>, reason: ChangeReason): Version {
    const previous = this.timeline.at(-1);
    const entry = {
      timestamp: now(),
      state,
      reason,
      hash: computeHash(state, previous?.hash ?? GENESIS)
    };
    this.timeline.push(entry);
    return entry.hash;
  }

  // Git-like operations
  diff(v1: Version, v2: Version): MemoryDiff<S> { ... }
  merge(branch: VersionedMemory<S>): MergeResult<S> { ... }
  rebase(onto: Version): RebaseResult<S> { ... }

  // Provenance queries
  whyChanged(field: keyof S['fields']): ChangeReason[] {
    return this.timeline
      .filter(e => fieldChanged(e.state, field))
      .map(e => e.reason);
  }
}
```

The implementation provides Git-like operations (diff, merge, rebase) while preserving type safety and provenance tracking.

7.4 Example Schema Library

We provide a library of common memory schemas for AI agents.

Listing 4: Identity Schema

```
export const IdentitySchema = defineSchema({
  name: 'Identity',
  fields: {
    agentId: { type: 'uuid', required: true },
    name: { type: 'string', required: true },
    capabilities: { type: 'set', elementType: 'Capability' },
    constraints: { type: 'list', elementType: 'Constraint' }
  }
});
```

Listing 5: Episodic Memory Schema

```
export const EpisodicMemorySchema = defineSchema({
  name: 'EpisodicMemory',
  fields: {
    episodeId: { type: 'uuid', required: true },
    timestamp: { type: 'datetime', required: true },
    participants: { type: 'set', elementType: 'AgentRef' },
    events: { type: 'list', elementType: 'Event' },
    summary: { type: 'string', optional: true },
    salience: { type: 'float', required: true }
  },
  invariants: [
    (m) => m.salience >= 0 && m.salience <= 1,
    (m) => m.events.length > 0
  ]
});
```

Listing 6: Semantic Memory Schema

```
export const SemanticMemorySchema = defineSchema({
  name: 'SemanticMemory',
  fields: {
    concept: { type: 'string', required: true },
    definition: { type: 'string', required: true },
    relations: {
      type: 'map',
      keyType: 'RelationType',
      valueType: 'list<ConceptRef>'
    },
    confidence: { type: 'float', required: true },
    sources: { type: 'list', elementType: 'Source' }
  },
  invariants: [
    (m) => m.confidence >= 0 && m.confidence <= 1,
    (m) => m.sources.length > 0
  ]
});
```

Listing 7: Working Memory Schema

```
export const WorkingMemorySchema = defineSchema({
  name: 'WorkingMemory',
  fields: {
    currentGoal: { type: 'ref', target: 'Goal', required: true },
    activeContext: { type: 'ref', target: 'Context' },
    scratchpad: { type: 'map', keyType: 'string', valueType: 'any'
      },
    attentionStack: { type: 'list', elementType: 'AttentionItem' }
  },
  invariants: [
    (m) => m.attentionStack.length <= 7 // cognitive limit
  ]
});
```

8 Theoretical Implications

8.1 Memory as Proof-Carrying Data

Our framework reveals an important interpretation: memory states are *proof-carrying data*. Each $\mathsf{Mem}[S]$ value carries:

1. The data itself
2. A proof that all schema invariants hold
3. (In versioned form) A derivation of how the state was reached

This connects to the Curry-Howard correspondence: memory types correspond to propositions, and memory inhabitants correspond to proofs.

Proposition 8.1 (Curry-Howard for Memory). *There is a correspondence:*

Type Theory	***Logic***
$\mathsf{Mem}[S]$	*Proposition "Memory satisfying S exists"*
$m : \mathsf{Mem}[S]$	*Proof that such memory exists*
$\mathsf{AllSat\ Invariants}\ d$	*Conjunction of validity constraints*
$\mathsf{Versioned}[\mathsf{Mem}[S]]$	*History-indexed proposition*

8.2 Agent Continuity as Narrative Coherence

Versioned memory provides a formal account of agent continuity that goes beyond mere persistence.

Definition 8.1 (Narrative Coherence). An agent's memory history exhibits *narrative coherence* iff:

1. Every belief has a traceable origin (Theorem 4.2)
2. Every change has a justified reason (Definition 4.5)
3. The overall trajectory forms a comprehensible story

This notion of coherence captures something important about agent identity: an agent is not just its current state, but the history of how it came to be in that state.

8.3 The Epistemological Interpretation

Our framework can be interpreted epistemologically:

- **Schemas** correspond to conceptual frameworks—the categories through which an agent understands its domain
- **Types** correspond to the laws of thought—what belief configurations are coherent
- **Invariants** correspond to background assumptions—the constraints that all beliefs must satisfy
- **Versioning** corresponds to rational belief revision—the norms governing how beliefs change in response to evidence

This interpretation suggests that typed memory is not merely an engineering convenience but captures something fundamental about the structure of rational cognition.

8.4 Computability and Decidability

Proposition 8.2 (Invariant Checking Decidability). *For schemas with decidable invariants, membership in* $\mathsf{Mem}[S]$ *is decidable.*

Proof. To check if data d can form a valid memory, we need to verify $\mathsf{AllSat}\ S.\mathsf{Invariants}\ d$. This is a finite conjunction of decidable predicates, hence decidable. □

Remark 8.1. In practice, invariants should be restricted to decidable predicates (computable functions returning boolean). Undecidable invariants would make type-checking impossible.

9 Comparison with Existing Approaches

Table 1: Comparison of Memory Architectures

Aspect	Vector	KV	Files	RAG	Typed
Schema	None	None	Conv.	None	**Compile**
Invalid state	Runtime	Silent	Parse	Runtime	**Unrep.**
Provenance	Meta	Manual	Git	None	**Built-in**
Type safety	None	None	None	None	**Full**
Revision	Overwrite	Overwrite	Manual	Overwrite	**Causal**
Query sound	Stat.	None	None	Stat.	**Constr.**
Composition	Ad hoc	Ad hoc	Dir.	Ad hoc	**Typed**

9.1 Vector Stores

Vector stores (e.g., Pinecone, Weaviate, ChromaDB) organize memory by embedding similarity. While effective for retrieval, they lack structural guarantees, typed queries, and provenance tracking.

9.2 Key-Value Stores

Key-value stores (e.g., Redis, DynamoDB) provide fast access but offer no schema, allow silent corruption, and cannot enforce relations.

9.3 File Systems

File-based memory provides human-readability but relies on convention-based structure, external versioning, and parse-time validation.

9.4 Retrieval-Augmented Generation

RAG systems combine retrieval with generation but are chunk-based with no global consistency, use statistical retrieval, and have no persistent memory update mechanism.

Our typed memory framework addresses all these limitations through its foundational principles.

10 Future Work

10.1 Distributed Agent Memories

Extending the framework to distributed settings raises interesting challenges around eventual consistency, conflict resolution, and consensus protocols that preserve type safety.

10.2 Probabilistic Types

Many agent beliefs are uncertain. Incorporating probabilistic types would enable confidence intervals, probabilistic invariants, and Bayesian versioning.

10.3 Proof Assistants for Schema Design

Designing correct schemas is challenging. Future work includes interactive schema editors, schema inference, and migration verification.

10.4 Integration with Neural Systems

Bridging typed memory with neural networks through differentiable types, neural invariant learning, and hybrid architectures.

11 Conclusion

We have presented a formal framework for AI memory as a typed structure, grounded in three principles from ContextFS:

1. **Schemas shape reasoning**: Memory structure determines the space of valid inferences (Proposition 3.1)
2. **Types enforce validity**: Invalid memory states are unrepresentable by construction (Theorem 2.1)
3. **Versioning preserves continuity**: The causal structure of belief revision is maintained (Theorem 4.1)

The framework provides compile-time guarantees about memory validity, sound inference over memory contents, explainable belief revision through derivation tracking, compositional memory architecture (Theorem 6.1), and safe schema evolution (Theorem 5.1).

Beyond engineering benefits, the framework suggests that the structure of memory is not incidental to cognition but constitutive of it. What an agent can think depends on how its memory is organized. What beliefs are coherent depends on what invariants are enforced. What reasoning is valid depends on what schemas are in place.

Typed memory is thus not merely "better storage"—it is a foundation for principled AI cognition.

Acknowledgments

The author thanks the YonedaAI Research Collective for ongoing discussions on the foundations of AI memory systems.

References

[1] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, 1984.

[2] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013.

[3] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[4] Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.

[5] Edwin Brady. Idris, a general-purpose dependently typed programming language. *Journal of Functional Programming*, 23(5):552–593, 2013.

[6] Haskell B. Curry, Robert Feys, and William Craig. *Combinatory Logic*, volume 1. North-Holland, 1958.

[7] William A. Howard. The formulae-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic*. Academic Press, 1980.

[8] Carlos E. Alchourrón, Peter Gärdenfors, and David Makinson. On the logic of theory change. *Journal of Symbolic Logic*, 50(2):510–530, 1985.

[9] Peter Gärdenfors. *Knowledge in Flux*. MIT Press, 1988.

[10] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *Haskell Symposium*, 2012.

[11] Carlo Curino, Hyun Jin Moon, and Carlo Zaniolo. Graceful database schema evolution. *VLDB*, 1(1):761–772, 2008.

[12] Philip Wadler. Propositions as types. *Communications of the ACM*, 58(12):75–84, 2015.

[13] Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*. Springer, 2009.

A Complete Type Definitions

Listing 8: Core Memory Types in Haskell

```haskell
{-# LANGUAGE GADTs, DataKinds, TypeFamilies #-}

-- Core memory type
data Memory (s :: Schema) where
```

```
  MkMemory :: (data_ :: Record (Fields s))
           -> (proof :: AllSat (Invariants s) data_)
           -> Memory s

-- Versioned memory with causal history
data Versioned (m :: *) where
  MkVersioned :: (current_ :: m)
              -> (timeline :: [Timestamped (m, ChangeReason)])
              -> (valid :: ValidTimeline timeline current_)
              -> Versioned m

-- Change reasons
data ChangeReason where
  Observation :: Source -> Evidence -> ChangeReason
  Inference   :: InferenceRule -> [Memory s] -> ChangeReason
  Correction  :: Memory s -> Text -> ChangeReason
  Decay       :: (Float -> Float) -> ChangeReason

-- Schema definition
data Schema where
  MkSchema :: (fields :: [(Symbol, *)])
           -> (invariants :: [Constraint fields])
           -> (relations :: [Relation fields])
           -> Schema

-- Type families for schema access
type family Fields (s :: Schema) :: [(Symbol, *)]
type family Invariants (s :: Schema) :: [Constraint (Fields s)]
type family AllSat (cs :: [Constraint fs]) (d :: Record fs) ::
    Constraint
```

B Proof of Compositionality

Detailed Proof of Theorem 6.1. Let S_1 and S_2 be schemas with disjoint field names:

$$S_1.\mathsf{Fields} \cap S_2.\mathsf{Fields} = \emptyset$$

Let $m_1 : \mathsf{Mem}[S_1]$ and $m_2 : \mathsf{Mem}[S_2]$ with:

$$m_1 = (data_1, pf_1)$$
$$m_2 = (data_2, pf_2)$$

Step 1: Data construction. Define $data = data_1 \cup data_2$. This is well-defined by field disjointness.

Step 2: Invariant satisfaction for S_1. Let $c \in S_1.\mathsf{Invariants}$. Since c only references $S_1.\mathsf{Fields}$ and $data|_{S_1.\mathsf{Fields}} = data_1$, we have $c(data) = c(data_1)$. By pf_1, this holds.

Step 3: Invariant satisfaction for S_2. Symmetric to Step 2.

Step 4: Construction. $m_1 \otimes m_2 \coloneqq \mathsf{MkMem}\ data\ pf$ where pf combines pf_1 and pf_2. □