Schema-Factored Reasoning

Cross-Domain Procedural Transfer via Structure–Content Decomposition

Aryan Gupta
aryan.cs.app@gmail.com

May 19, 2026

Abstract

Current large language models simulate multi-step reasoning by autoregressive token prediction over surface representations. This collapses on tasks requiring transfer of an abstract procedure across content domains: the same algorithm applied to letter-string puzzles in a novel alphabet, the same equilibrium argument applied across physics and economics, the same inductive proof pattern applied across number theory and graph theory. The failure has a structural diagnosis. A standard transformer does not factorise the relational scaffold of a problem from its specific content, and so has no representation that can be transferred independently of content.

We propose the Schema-Factored Reasoning Architecture (SFRA), motivated by the structure–content factorisation observed in the hippocampal–entorhinal cognitive map [1, 2, 3] and by the structure-mapping account of analogy [7, 8]. SFRA maintains an explicit library of *schemas*: parameterised relational templates over typed operations, induced from few examples, stored as content-independent abstractions, and applied to new problems via structure-mapping. An LLM serves as the content engine that parses problems and supplies values; a separate structural layer maintains the schema library and selects, binds, and instantiates schemas; the resulting concrete computation runs on a deterministic graph executor (the Graph-Structured Recurrent Executor of the Notion-Based Reasoning System, retained as substrate). Library growth is driven by a replay-style consolidation process that abstracts schemas from successful traces.

The technical contributions of this document are: (i) formal definitions of schemas, structural binding, and analogical mapping over typed notion DAGs; (ii) the architectural specification of SFRA as a five-component system (content layer, structure extractor, schema library, analogy engine, execution substrate) with the consolidation process that grows the library; (iii) execution guarantees for instantiated schemas inherited from the underlying executor (determinism, termination, soundness, depth-decoupling); (iv) a precise statement of the transfer property the architecture is designed to achieve, with the conditions under which it holds; (v) a falsifiable empirical programme targeting cross-domain analogical transfer benchmarks where contemporary LLMs systematically fail.

1 Introduction

1.1 The transfer gap

The capability that most distinguishes human reasoning from contemporary large language models is not depth, breadth, or factual coverage. It is *transfer*: the ability to recognise that a problem in a new domain has the same structure as a problem already solved in some other domain, and to apply the previously-learned procedure to the new content without further training. A child who has learned long-division on integers can apply the same algorithm to polynomials with minimal additional instruction. A physicist who recognises that supply–demand equilibrium has the structure of mechanical equilibrium can transfer intuitions about damped oscillations to market dynamics. A reader who absorbs the moral of a parable can

identify the same structural pattern in a story with different characters, settings, and surface content.

Recent stress tests of analogical reasoning in frontier LLMs document the gap precisely. Lewis and Mitchell [9] and Webb et al. [11] show that performance on letter-string analogies, Raven's Progressive Matrices, and digit-pattern problems collapses when test stimuli are constructed to be unlikely in training data: GPT-class models fail on novel alphabets where humans, including children, generalise robustly. The failure mode is consistent across studies: LLMs match on surface features, not on relational structure. Where the abstract pattern requires a content-independent representation, the model has nothing to fall back on.

1.2 The neurobiological evidence for structure–content factorisation

The brain solves the transfer problem with an architectural commitment that contemporary neural networks do not make. The hippocampal–entorhinal system maintains *factorised* representations: structural codes encode the relational scaffold of a task or environment (the graph of transitions, the relative positions, the type relationships), while sensory codes encode the specific content (which objects are at which positions, which values fill which slots). Conjunctive cells in the hippocampus combine them.

The factorisation is striking enough that the same grid-cell apparatus that organises physical spatial navigation also organises navigation through abstract conceptual spaces. Constantinescu, Behrens, and colleagues [3] demonstrated that human subjects navigating a continuous two-dimensional space of bird shapes (defined by neck length and leg length) exhibit the same six-fold hexagonal fMRI signature that grid cells produce for physical space. The brain's structural representation is domain-general; what changes between physical and conceptual tasks is the content bound to that structure, not the structure itself.

The Tolman–Eichenbaum Machine [1] formalises this as a computational model. After training on diverse environments, its medial-entorhinal cells learn to represent the abstract transition structure shared across environments; its lateral-entorhinal cells encode the sensory content; its hippocampal cells encode conjunctions. Transfer to a new environment requires re-binding sensory content to an already-learned structure, not relearning structure from scratch. Recent extensions and ablations [4, 5] show that the factorisation is necessary for the model's transfer behaviour: removing it removes the generalisation property.

1.3 The proposal

We propose the Schema-Factored Reasoning Architecture (SFRA): a neurosymbolic system that operationalises the cognitive-map factorisation for procedural reasoning. SFRA maintains a library of *schemas*—parameterised relational templates over typed operations—that are induced from few examples, stored as content-independent abstractions, and applied to new problems by mapping the problem's relational structure onto the schema and binding the schema's variables to problem content. An LLM provides the content layer (parsing problems, supplying values, generating natural-language output); a structural layer parses content into relational graphs and operates the schema library; instantiated schemas execute on a deterministic typed-DAG executor inherited from prior work on the Notion-Based Reasoning System; a consolidation process driven by replay over successful traces grows the library over time.

The technical content of this document is in five parts. Section 2 fixes the conceptual framework: the structure–content distinction, schemas as the unit of transfer, the relationship to cognitive maps. Sections 3 and 4 give the formal preliminaries and define schemas, instantiation, and structural matching. Section 5 specifies the five-component architecture. Section 6 compresses the execution substrate (the Graph-Structured Recurrent Executor; full operational properties retained in Appendix A). Sections 7 and 8 develop the schema-induction and analogical-transfer mechanisms, including conditions for soundness. Section 10 states the empirical programme:

a tiered set of cross-domain transfer benchmarks on which the architecture is to be evaluated against contemporary LLM baselines.

2 Conceptual Framework

2.1 The structure–content distinction

A reasoning problem has two separable aspects. The *content* consists of the specific values, entities, types, and surface features that the problem mentions. The *structure* consists of the relational scaffold: which entities depend on which others, what operations connect them, what invariants hold, what the goal is in relation to the givens.

Two problems share content when they mention the same things; they share structure when the dependency graphs of their solutions are isomorphic up to relabelling. Contemporary LLMs blend these in a single token stream and a single attention computation. When training distributions cover the content–structure joint distribution densely, this works; when test problems combine familiar structure with unfamiliar content, it breaks.

2.2 Schemas as the unit of transfer

We define a *schema* as a parameterised relational template: a typed notion-DAG in which leaves and constants are replaced by typed variables. Concretely, a schema for the binary-search procedure has the form "given a sorted sequence x of type T^* and a target value y of type T, iteratively halve the search interval according to a comparison primitive $\mathsf{cmp} : T \times T \to \mathsf{Bool}$." This template has no commitment to what T is, what specific elements x contains, or what specific value y takes. Instantiating the schema requires binding T, x, y, and cmp to a concrete problem; once bound, the schema produces a concrete typed DAG that the executor can run.

A schema, in this sense, is exactly the object that the cognitive-map literature identifies as the entorhinal structural code. It is the abstract pattern that survives transfer; the binding is the conjunctive code that combines structure with content.

2.3 Analogical transfer as structural alignment

Given a new problem P and a library of schemas, transfer proceeds in three steps. First, the structure extractor parses P into a relational graph G_P describing P's entities, types, and dependencies. Second, the analogy engine searches the library for a schema S whose structure aligns with G_P under some type-respecting binding β. Third, the binding β instantiates S into a concrete DAG that the executor runs.

The structural alignment in the second step is precisely Gentner's structure-mapping [7]: a partial isomorphism between schema variables and problem entities that preserves the relational structure. It is not surface similarity. Two problems with no shared surface features can share structure, and the analogy engine finds the schema whose structure matches—regardless of whether the problem's content was ever seen during library construction.

3 Formal Preliminaries

We work in a typed first-order setting. Values have types; operations have input and output types; reasoning is a typed computation organised as a DAG; schemas are typed DAGs with variables.

3.1 Types, values, primitives

Definition 3.1 (Type system)**.** A type system is a pair $(\mathcal{T}, \mathcal{V})$ where $\mathcal{T}$ is a finite, decidable set of types and $\mathcal{V} = \bigsqcup_{\tau \in \mathcal{T}} \mathcal{V}_\tau$ is a disjoint union of value domains. We write $v : \tau$ for $v \in \mathcal{V}_\tau$. The type system may include type constructors: list types T^*, product types $T \times T'$, function types $T \to T'$.

Definition 3.2 (Primitive operation)**.** A primitive operation is a tuple $\tau = (\mathsf{name}_\tau, \mathsf{in}_\tau, \mathsf{out}_\tau, f_\tau)$ with $\mathsf{in}_\tau \in \mathcal{T}^*$ of length k_τ, $\mathsf{out}_\tau \in \mathcal{T}$, and $f_\tau : \mathcal{V}_{\tau_1} \times \cdots \times \mathcal{V}_{\tau_{k_\tau}} \to \mathcal{V}_{\mathsf{out}_\tau}$ a total semantic function. A primitive library $\mathcal{L} = \{\tau_1, \ldots, \tau_K\}$ is a finite set.

3.2 Notions and notion DAGs

A notion is an instance of a primitive operation in a particular computation. A notion DAG is a typed, dependency-ordered graph of notions whose root is the computation's output.

Definition 3.3 (Notion)**.** A notion over a library $\mathcal{L}$ is a tuple $n = (\mathsf{id}_n, \tau_n, \mathsf{spec}_n, \mathsf{pa}_n)$ where id_n is a unique identifier, $\tau_n \in \mathcal{L}$, spec_n encodes auxiliary parameters of the primitive, and $\mathsf{pa}_n = (p_1, \ldots, p_{k_{\tau_n}})$ is the ordered tuple of parent identifiers. A leaf notion has $\mathsf{pa}_n = ()$ and a concrete leaf value supplied externally.

Definition 3.4 (Notion DAG)**.** A notion DAG is a pair $G = (V, r)$ with V a finite set of notions, all parent identifiers referring to elements of V, the induced parent relation acyclic, and $r \in V$ a distinguished output notion. The DAG is well-formed when (i) types match along every edge and (ii) every non-leaf notion has exactly k_{τ_n} parents. All notion DAGs in the remainder are assumed well-formed.

Definition 3.5 (Compositional depth)**.** The depth of notion n is $\mathsf{depth}(n) = 0$ if $\mathsf{pa}_n = ()$ and $1 + \max_{p \in \mathsf{pa}_n} \mathsf{depth}(p)$ otherwise. The depth of the graph is $\mathsf{depth}(G) := \mathsf{depth}(r)$.

4 Schemas

Schemas are the load-bearing object of SFRA: they are what the library stores, what the analogy engine matches, and what transfer transfers.

4.1 Schema variables

Where a notion has a concrete primitive $\tau_n \in \mathcal{L}$ and a concrete spec, a schema node may have a variable in any of these positions. We introduce three classes of variables, corresponding to the three axes of content along which transfer must abstract.

Definition 4.1 (Schema variables)**.** A schema admits three kinds of variables, drawn from disjoint countable sets:

- *Type variables* $X, Y, Z, \ldots$ ranging over $\mathcal{T}$. A schema is parameterised over the types of values it operates on.
- *Value variables* $x, y, z, \ldots$ ranging over $\mathcal{V}$. A schema is parameterised over the specific leaf values supplied at runtime.
- *Operator variables* $\alpha, \beta, \gamma, \ldots$ ranging over $\mathcal{L}$ restricted by type signature. A schema may leave the choice of a particular primitive (e.g., the binary operation passed to a fold) as a parameter.

4.2 Schemas as parameterised templates

Definition 4.2 (Schema)**.** A schema S is a tuple $(V_S, r_S, \mathsf{params}_S)$ where:

- V_S is a finite set of *schema notions*, each of the form $(\mathsf{id}, \alpha, \mathsf{spec}, \mathsf{pa})$ where α may be either a concrete primitive in $\mathcal{L}$ or an operator variable, and spec may contain value variables.
- $r_S \in V_S$ is a distinguished root.
- params_S is a list of typed parameters: each type variable's kind, each value variable's type (which may itself reference a type variable), each operator variable's input/output type signature (likewise).

The induced parent relation must be acyclic and well-typed under any consistent variable assignment.

Example 4.3 (The fold schema)**.** The schema $\mathsf{FOLD}(\alpha, x_0, \ell)$ has parameters: a type variable X (element type), a type variable Y (accumulator type), an operator variable $\alpha : Y \times X \to Y$, a value variable $x_0 : Y$ (initial accumulator), and a value variable $\ell : X^*$ (the sequence). The schema body is a length-dependent chain: for a sequence of length k, the schema expands into a depth-k chain of α invocations starting from x_0. We treat the schema as a function from inputs to a concrete notion-DAG via the expansion rule.

Example 4.4 (The binary-search schema)**.** The schema $\mathsf{BSEARCH}(\mathsf{cmp}, \ell, y)$ has a type variable X, an operator variable $\mathsf{cmp} : X \times X \to \mathsf{Ordering}$, a value variable $\ell : X^*$ (a sorted sequence), and a value variable $y : X$ (the target). The body iteratively halves the candidate range using cmp until the target is located or the range is empty.

4.3 Instantiation

Definition 4.5 (Schema instantiation)**.** A binding β for a schema S is a map from S's parameters to concrete types, values, and primitives respecting the parameter signatures. The instantiation $\mathsf{Inst}_\beta(S)$ of S under β is the concrete notion DAG obtained by substituting β's assignments into the schema body, applying any length-dependent expansion rules, and resolving the resulting DAG's types.

Proposition 4.6 (Well-typed instantiation produces a well-formed DAG)**.** *For any schema S and any binding β that respects S's parameter signatures, $\mathsf{Inst}_\beta(S)$ is a well-formed notion DAG.*

Proof. Direct from the schema's typing discipline and the well-formedness of the substitution. Type variables are bound to concrete types; value variables to values of the bound type; operator variables to primitives with the bound signature. Edge typing in the instantiated DAG follows. □

4.4 Schema composition

Schemas compose by allowing a schema notion to invoke another schema rather than a primitive. The composition mechanism is identical to function calling in a typed lambda calculus: a schema S that uses S' as a sub-procedure simply contains a schema notion whose operator is S' (with its parameters bound either to concrete values or to outer variables of S). Composition is therefore the mechanism by which higher-level reasoning patterns (e.g., divide-and-conquer) are built from lower-level ones (e.g., a sort, a merge).

5 The Schema-Factored Reasoning Architecture

Sfra is a five-component system. Each component has a specific responsibility and a specific interface to the others.

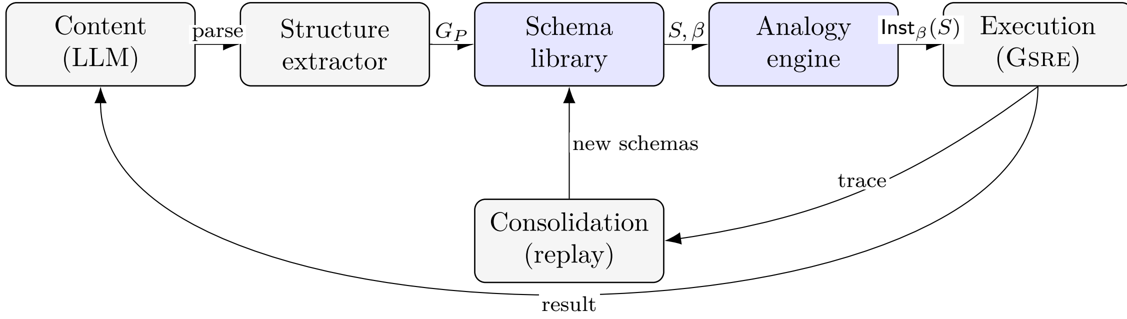

Figure 1: The Schema-Factored Reasoning Architecture. The pipeline is linear: the content layer parses the problem, the structure extractor produces a problem graph G_P, the schema library is queried by the analogy engine which returns a matched schema S with a binding β, and the instantiation $\mathsf{Inst}_\beta(S)$ runs on the execution substrate. Two feedback loops grow and apply the system: traces from successful executions feed the consolidation process, which abstracts new schemas back into the library; the executor's final value returns to the content layer for natural-language output.

5.1 Content layer

The content layer is a frontier LLM. Its responsibility is restricted to what LLMs do well: parsing natural-language problems into entities and types, supplying concrete values, recognising domain-specific patterns at the surface level, and generating natural-language output from concrete answers. It does not perform multi-step reasoning. The architectural assumption is that an LLM trained on web-scale text is a competent semantic parser and a competent content generator, even when it cannot reliably execute multi-step logical procedures.

5.2 Structure extractor

The structure extractor receives a parsed problem from the content layer and produces a *problem graph* G_P: a typed relational graph describing the problem's entities (typed nodes), the relations among them (typed edges or operation nodes), the givens (which entities are known), and the goal (which entity is to be computed). The structure extractor abstracts away from surface vocabulary as far as it can while preserving relational fidelity. A simple word problem about apples and a structurally identical problem about voltages should produce isomorphic problem graphs modulo type relabelling.

5.3 Schema library

The schema library $\mathcal{S}$ is a growing set of schemas. Each schema is stored with: its parameterised structure (Definition 4.2), a precondition predicate over problem graphs that determines applicability, example traces from which the schema was abstracted, and meta-information used by the analogy engine (frequency of use, success rate, average depth of instantiated DAG).

5.4 Analogy engine

The analogy engine receives a problem graph G_P and queries the library to find schemas whose structure aligns with G_P. Alignment is a partial graph isomorphism under a typed binding: a map from the schema's variables to entities, types, and operators in G_P that preserves the relational structure. Where multiple schemas match, the engine ranks candidates by structural coverage, parameter fit, and prior success rate; where partial matches are possible, the engine

may propose a binding that uses a sub-schema and leaves the residual structure to be solved by another (recursive) library lookup.

The analogy engine is the architectural analogue of Gentner's Structure-Mapping Engine [8], lifted to learned representations of types and primitives.

5.5 Execution substrate

Once a schema S is selected and bound, $\mathsf{Inst}_\beta(S)$ is a concrete notion DAG. It runs on the Graph-Structured Recurrent Executor (Section 6, full treatment in Appendix A). The substrate provides the determinism, termination, and soundness guarantees that make the architecture's output predictable from the library's content.

5.6 Consolidation

The consolidation process is offline and replay-driven, in deliberate analogy to hippocampal replay during rest [5, 6]. The consolidation engine periodically examines the corpus of recent successful traces, looks for recurring structural patterns, abstracts these patterns into candidate schemas by replacing constants with typed variables, and (subject to verification on held-out traces) adds them to the library. Section 7 treats this in detail.

6 The Execution Substrate

The execution layer of SFRA is the Graph-Structured Recurrent Executor (GSRE), inherited from the Notion-Based Reasoning System. We summarise its properties here and defer the full operational specification to Appendix A. The role of this section is to establish what the architecture inherits from the executor, not to develop it as a contribution.

6.1 Summary of executor properties

The GSRE is a deterministic state machine over a typed notion DAG. Each node carries a state in {PENDING, READY, EXECUTING, RESOLVED, FAILED}; a node becomes READY only when all of its parents are RESOLVED. A shared neural module $\mathsf{NodeProc}_\theta$ is invoked at each node on its local context $(\tau_n, \mathsf{spec}_n, (v_p)_{p\in\mathsf{pa}_n})$ alone. The same parameters θ are reused at every node, regardless of position or graph size.

The properties of relevance to SFRA are:

- *Determinism.* For any DAG G and leaf assignment, the executor produces a unique value assignment (Theorem A.1).
- *Order independence.* The value assignment is invariant under any choice of topological schedule among ready nodes (Theorem A.2).
- *Termination.* Execution completes in at most $|V|$ steps (Theorem A.3).
- *Soundness against the oracle.* If $\mathsf{NodeProc}_\theta$ is primitive-correct on every node it sees, the executor's output equals the oracle's (Theorem A.4).
- *Per-step bounded input.* The per-invocation input size is bounded by the maximum primitive arity, independent of $\mathsf{depth}(G)$ and $|V|$ (Proposition A.5).
- *Capacity–depth decoupling.* Under an independent-error model, success probability decays at worst geometrically in depth, with the per-step parameter count constant in depth (Theorem A.6).

6.2 Inherited guarantees for instantiated schemas

The point of inheriting the substrate is the following: every schema instantiation $\mathsf{Inst}_\beta(S)$ is, by construction, a well-formed notion DAG. The executor properties above therefore apply directly. The architecture commits to the following inherited guarantee:

Proposition 6.1 (Inherited correctness for instantiated schemas)**.** *If a schema S is sound (its expansion under a valid binding β produces a DAG whose oracle evaluation equals the intended schema semantics) and the node processor $\mathsf{NodeProc}_\theta$ is primitive-correct on the primitives that $\mathsf{Inst}_\beta(S)$ invokes, then* Sfra*'s output on S instantiated by β equals the schema's intended semantics.*

The architectural separation isolates two questions that previous monolithic reasoning systems conflate. *Is the right algorithm applied?* is a question about schema selection and binding—the analogy engine's job. *Is the algorithm executed correctly?* is a question about the substrate's reliability—the executor's job. The two can be analysed and improved independently.

7 Schema Induction

The library grows by inducing schemas from traces of successful problem-solving. This section formalises the induction problem and identifies its key challenges.

7.1 The induction problem

Definition 7.1 (Trace corpus)**.** A trace corpus is a finite collection $\mathcal{T} = \{(G_i, \beta_i, v_i)\}_{i=1}^N$ where each G_i is a notion DAG, β_i is the binding under which it was produced (if known), and v_i is its root output value (verified correct).

Definition 7.2 (Schema induction)**.** The schema-induction problem is: given a trace corpus $\mathcal{T}$ and the current library $\mathcal{S}$, identify candidate new schemas S^* such that some non-trivial subset of $\mathcal{T}$ is expressible as instantiations of S^* under appropriate bindings, and admit S^* to $\mathcal{S}$ when the resulting compression of $\mathcal{T}$ exceeds the description-length cost of S^*.

7.2 Compositional compression

Definition 7.3 (Compression value of a candidate schema)**.** For a candidate schema S^*, let $\mathsf{matches}(S^*, \mathcal{T})$ be the set of traces in $\mathcal{T}$ that can be re-expressed as $\mathsf{Inst}_\beta(S^*)$ for some valid binding β. Let $|G_i|$ be the description length of the explicit trace, $|\beta|$ the description length of a binding, and $|S^*|$ the description length of the schema itself. The compression value is

$$\Delta(S^*; \mathcal{T}) \;=\; \sum_{i \in \mathsf{matches}(S^*, \mathcal{T})} \big(|G_i| - |\beta_i|\big) \;-\; |S^*|.$$

A candidate with $\Delta(S^*; \mathcal{T}) > 0$ is admitted; one with $\Delta \leq 0$ is rejected as unproductive. The criterion is the standard minimum-description-length criterion used in inductive program synthesis, adapted to the schema setting [14].

7.3 Type generalisation

The hardest sub-problem of schema induction is type generalisation. A trace corpus may contain traces over Int, Float, and String that all share structural pattern P. The induced schema should be parameterised over a type variable X, with the structural pattern P preserved. Detecting that the corpus is consistent with a single type-parameterised schema—rather than three separate type-specific schemas—is the operation that turns library extension into genuine abstraction.

We do not give an induction algorithm here; the formal target is a procedure that, given $\mathcal{T}$, produces candidate type-parameterised schemas with $\Delta > 0$. Approaches consistent with the framework include: unification-based generalisation over typed graph patterns, neural sequence-to-sequence schema synthesis, and wake–sleep procedures in the style of DreamCoder.

7.4 Verification of induced schemas

A candidate S^* admitted to the library must be sound: applying S^* to a binding β must produce a DAG whose oracle evaluation matches the schema's intended semantics. Two verification strategies are operationally distinct:

- *Compression-based.* S^* is admitted if it compresses a held-out portion of $\mathcal{T}$ that was not used in its induction. This guards against overfitting to the training traces.
- *Executable-test-based.* The system generates synthetic bindings for S^*, executes the resulting DAGs against an oracle (where available) or against a corroborating schema or LLM-supplied ground truth, and admits S^* only when the success rate on synthetic tests exceeds a threshold.

The verification step is what separates schema induction from blind pattern-mining. A schema admitted without verification is at best a hypothesis; admitted with verification, it is a tool the architecture can rely on.

8 Analogical Transfer

The transfer claim that distinguishes SFRA from contemporary monolithic reasoning systems is the following: a schema learned from problems in one content domain can be applied to a problem in a different content domain whose structure aligns with the schema, without retraining and without exposure to bindings drawn from the new domain.

8.1 The transfer property, stated

Definition 8.1 (Cross-domain transfer)**.** Let $\mathcal{S}$ be the schema library after training on a problem distribution over content domain $\mathcal{C}_{\text{train}}$. Let P^* be a problem from content domain $\mathcal{C}_{\text{test}} \neq \mathcal{C}_{\text{train}}$ whose problem graph G_{P^*} admits a valid binding β^* to some schema $S \in \mathcal{S}$ such that $\mathsf{Inst}_{\beta^*}(S)$ correctly solves P^*. SFRA exhibits cross-domain transfer on P^* if the analogy engine, given G_{P^*} and $\mathcal{S}$, returns (S, β^*) (or any other binding that yields a correct DAG).

The property has three preconditions: (i) a schema with the required structure exists in the library (the induction layer has produced it from earlier training); (ii) the structure extractor produces a problem graph for P^* that is amenable to alignment with that schema; (iii) the analogy engine identifies and binds the schema correctly. Failure to transfer can be diagnosed to one of these three preconditions.

8.2 Structure-mapping over learned representations

The analogy engine implements a form of structure-mapping [7]: it searches for a partial isomorphism between the schema's graph and the problem's graph that preserves relations and respects types, prioritising mappings that align deeper relational structure over those that align mere surface features. The implementation challenge—which the literature on classical structure-mapping engines did not face—is that the schema's variables and the problem's entities live in learned representational spaces. The alignment is not over hand-written symbols but over typed embeddings.

We treat the alignment procedure as itself a learned component: a neural matcher that takes a schema graph and a problem graph and outputs a candidate binding, trained on (synthetic and real) pairs of (problem, applicable schema) examples. Training such a matcher is the central engineering problem of the analogy engine.

8.3 Soundness of transfer

Proposition 8.2 (Soundness of bound transfer)**.** *Suppose $S \in \mathcal{S}$ is a sound schema and $\mathsf{NodeProc}_\theta$ is primitive-correct on the primitives that $\mathsf{Inst}_\beta(S)$ invokes. If the analogy engine returns a binding β such that the parameter-signature constraints of S are satisfied and the typing of $\mathsf{Inst}_\beta(S)$ matches the problem P^*'s required input–output types, then the value v_r returned by the executor is correct for P^*.*

Proof. Schema soundness gives correctness of $\mathsf{Inst}_\beta(S)$ against the schema's intended semantics. Primitive correctness gives executor correctness against the schema's semantics. Type compatibility gives that the schema's semantics is what P^* requires. □

The implication: transfer failure is always attributable to one of (i) wrong schema selected, (ii) wrong binding constructed, (iii) executor error on the primitives, (iv) type-compatibility mismatch undetected by the analogy engine. The architecture localises blame, which is what makes failure analysis tractable.

9 Generalisation Properties

The structure–content factorisation has direct consequences for generalisation. Because schemas are content-independent, applying a learned schema to new content does not require learning new schema parameters. Because the executor sees only local contexts, executing a schema on values drawn from a new distribution does not require depth-dependent capacity. The architecture's generalisation properties are therefore inherited from two separable sources: schema-level generalisation (the analogy engine's ability to recognise structural matches across domains) and execution-level generalisation (the executor's ability to handle the bindings the analogy engine produces).

9.1 Schema-level generalisation

Proposition 9.1 (Structural transfer dominates surface transfer)**.** *Let $\mathcal{S}$ be a library trained on $\mathcal{C}_{\text{train}}$. For a test problem P^* from $\mathcal{C}_{\text{test}}$, the existence of a successful binding to some $S \in \mathcal{S}$ depends only on the relational structure of P^*, not on whether $\mathcal{C}_{\text{test}}$ overlaps $\mathcal{C}_{\text{train}}$.*

Proof. Schema variables abstract over types, values, and operators. Two problems with isomorphic relational structure (under a type relabelling that respects operator signatures) admit the same set of schema bindings. Domain overlap affects only what types and values appear; it does not affect what relational structures appear. Therefore the existence of a transfer-supporting schema is invariant to domain overlap, conditional on the structure of P^* being one for which an appropriate schema exists in $\mathcal{S}$. □

This is the formal counterpart of the cognitive-map result: a structural representation that abstracts over content predicts transfer wherever structure repeats.

9.2 Execution-level generalisation

The execution substrate inherits the local-context invariance property of the GSRE: the per-node processor sees only the local context $(\tau_n, \mathsf{spec}_n, (v_p)_{p \in \mathsf{pa}_n})$, so its error rate on a node

depends only on the marginal distribution of local contexts, not on the depth or shape of the surrounding graph (Theorem A.7). For instantiated schemas, this means that depth—which can be arbitrary if a schema like FOLD is bound to a long sequence—does not introduce new failure modes at the executor level.

10 Empirical Programme

The architecture's claims are empirical. The minimal programme that would test them comprises four tiered demonstrations, each calibrated to a specific architectural prediction.

10.1 Tier 1: structural transfer within a domain

The minimum viable test: train the schema library on traces of a structural pattern (e.g., the binary-search procedure) instantiated on integer inputs. Without further training, evaluate on the same procedure applied to inputs of types not seen during training (strings, floating-point numbers, structured records). Success requires that the schema's type variables be correctly bound to the new type and that the executor handle the new primitives. Failure modes localise to the analogy engine (type binding) or to the executor (primitive accuracy on new types).

10.2 Tier 2: cross-content transfer within a structural pattern

A stronger test: train on traces from one content domain whose problem graphs share structure with problems in a different content domain. Examples include: train on optimisation problems framed as iterative refinement (e.g., Newton's method on polynomial roots), then test on problems whose underlying procedure is structurally identical but whose content is entirely different (e.g., bisection on binary games, fixed-point iteration on convergent series). Success requires the analogy engine to recognise the structural match across surface content.

10.3 Tier 3: cross-domain analogical reasoning benchmarks

The architecture's claim should be testable against contemporary benchmarks designed to expose LLM analogical failures. Two candidates:

- *Letter-string analogies in novel alphabets* [9, 10]. LLMs are known to fail on these in proportion to the unfamiliarity of the alphabet. SFRA, with schemas abstracting over the alphabet's specific symbols, should not exhibit this degradation.
- *ARC-AGI tasks* [12, 13]. Each task supplies a few input–output examples of an abstract transformation and requires applying it to a held-out input. The benchmark is designed to require few-shot abstraction beyond pattern matching. SFRA's schema-induction layer is the architectural counterpart to what these tasks measure.

10.4 Tier 4: cross-discipline conceptual transfer

The architecture's most ambitious target: train the library on procedures from one scientific or mathematical discipline, and demonstrate that schemas induced from training transfer to a structurally analogous problem in an unrelated discipline. Candidate transfer pairs include: equilibrium reasoning in mechanics $\to$ market-clearing in economics; iterative refinement in numerical analysis $\to$ contraction-mapping arguments in functional analysis; recursive case analysis in combinatorics $\to$ inductive proofs in graph theory. Success at this tier would directly demonstrate the analogical capability that humans display and contemporary LLMs do not.

10.5 Baselines and diagnostics

Each tier is to be evaluated against three baselines: a frontier LLM with chain-of-thought prompting, a frontier LLM with retrieval augmentation over the same trace corpus from which SFRA's library was induced, and an ablation of SFRA with the schema library disabled (forcing the architecture to solve each problem from scratch via the executor). Diagnostics include schema-induction efficiency (how many traces are required to induce each schema), transfer success rate as a function of structural divergence between training and test problems, and the marginal-invariance diagnostic from prior work on the executor.

11 Connections to Prior Work

Cognitive maps and entorhinal–hippocampal computation. The architectural commitment to structure–content factorisation is taken from the Tolman–Eichenbaum Machine [1] and the cognitive-map literature it formalises [2, 3, 4]. SFRA operationalises the same factorisation in a procedural-reasoning setting: schemas are structural codes, bindings are conjunctive codes, and library growth is analogous to entorhinal consolidation.

Structure-mapping theory. The analogy engine is the operational counterpart of Gentner's Structure-Mapping Theory [7] and its computational implementation, the Structure-Mapping Engine [8]. The novel commitment of SFRA is to perform structure-mapping over learned (rather than hand-symbolic) representations of types and operators.

Library learning and inductive program synthesis. Schema induction (Section 7) is closest to library learning as developed in DreamCoder [14] and the broader inductive-program-synthesis literature [15]. SFRA extends this to typed parameterisation over operators and to integration with an LLM content layer, both of which the existing library-learning systems do not address.

Neural module networks. Andreas et al. [16] compose computation graphs from a fixed module library at inference time. SFRA generalises this in two directions: the library is grown (not fixed), and the modules carry typed parameterisation that supports cross-domain transfer.

Consolidation in neural models. The replay-driven consolidation process is informed by recent computational models of hippocampal replay and memory consolidation [5, 6].

LLM-augmented neurosymbolic reasoning. Recent work on neurosymbolic LLM augmentation [9] typically translates problems into a fixed logical form for an external solver. SFRA differs in maintaining a learned, growing library of typed schemas; the content layer's role is parsing, not full reduction to logic.

The executor substrate. The Graph-Structured Recurrent Executor and its operational guarantees are inherited from prior work on the Notion-Based Reasoning System, summarised here in Section 6 and detailed in Appendix A.

12 Discussion

The architectural bet of SFRA is that the transfer gap separating contemporary LLMs from human reasoning is fundamentally about representation: the absence of an explicit, content-independent layer that stores transferable procedures. The neuroscience evidence for such a

layer in biological reasoners is strong and recent. The classical symbolic literature on analogical reasoning has long argued that structure-mapping is the right computational model. What has been missing is an integration that combines the abstraction power of cognitive-map-style factorisation, the procedural depth of program-induction-style library learning, and the perceptual breadth of LLM-style content processing.

Two design commitments of Sfra are non-obvious and warrant emphasis. First, the schema library is the architectural locus of learning at the structural level: the LLM is not asked to learn or transfer procedures, only to parse and generate. This separation is what gives the system the prospect of sample-efficient transfer. Second, the execution substrate is deliberately conservative: the Gsre guarantees only what a deterministic typed-DAG evaluator can guarantee. The interesting failure modes of the system are therefore localised to schema induction and analogical alignment, not to execution—which is the right place for them to be.

The empirical risk is real. Schema induction in non-trivial domains is open; structure-mapping in learned representations is open; integration with LLMs at the content layer is open. The architecture is not a guaranteed solution. It is a coherent place to spend effort, with each architectural commitment tied to a specific scientific hypothesis and a specific failure-diagnosis pathway.

If the empirical programme of Section 10 succeeds at Tiers 1 and 2 with reasonable sample complexity, the architecture has demonstrated the structure–content factorisation hypothesis at small scale. If it succeeds at Tier 3, it has demonstrated competitive performance with frontier LLMs on benchmarks designed to expose their analogical failures. If it succeeds at Tier 4, it has demonstrated cross-discipline analogical reasoning—the capability that, more than any other, distinguishes human general intelligence from current artificial systems.

A The Execution Substrate: Full Specification

This appendix specifies the Gsre in detail and proves its operational properties. The text of this appendix is inherited from the earlier formal development of the Notion-Based Reasoning System and is retained here without substantive modification.

A.1 Node states

During execution, every node $n \in V$ is in one of five states drawn from $\mathcal{Q} = \{$Pending, Ready, Executing, Res written $\mathsf{state}(n) \in \mathcal{Q}$. Once $\mathsf{state}(n) =$ Resolved, the associated value $v_n \in \mathcal{V}_{\mathsf{out}_{\tau_n}}$ is fixed.

A.2 The traversal engine

A.3 Operational properties

Theorem A.1 (Determinism)**.** *Let* $\mathsf{NodeProc}_\theta$ *be a deterministic function of its input. For any well-formed notion DAG* G *and leaf assignment* σ*, the value assignment returned by Algorithm 1 is uniquely determined by* G*,* σ*, and* θ*.*

Theorem A.2 (Order independence)**.** *Under the hypothesis of Theorem A.1, the value assignment produced is invariant under any nonempty subset* $\mathcal{B} \subseteq R$ *chosen at each iteration.*

Theorem A.3 (Termination)**.** *Algorithm 1 terminates in at most* $|V|$ *iterations, with at most* $|V| - |V_{\text{leaf}}|$ *processor invocations.*

Theorem A.4 (Soundness against the oracle)**.** *If* $\mathsf{NodeProc}_\theta$ *is primitive-correct on every primitive it sees, the value* v_r *returned by Algorithm 1 equals the oracle ground-truth answer for any reasoning task* (G, σ)*.*

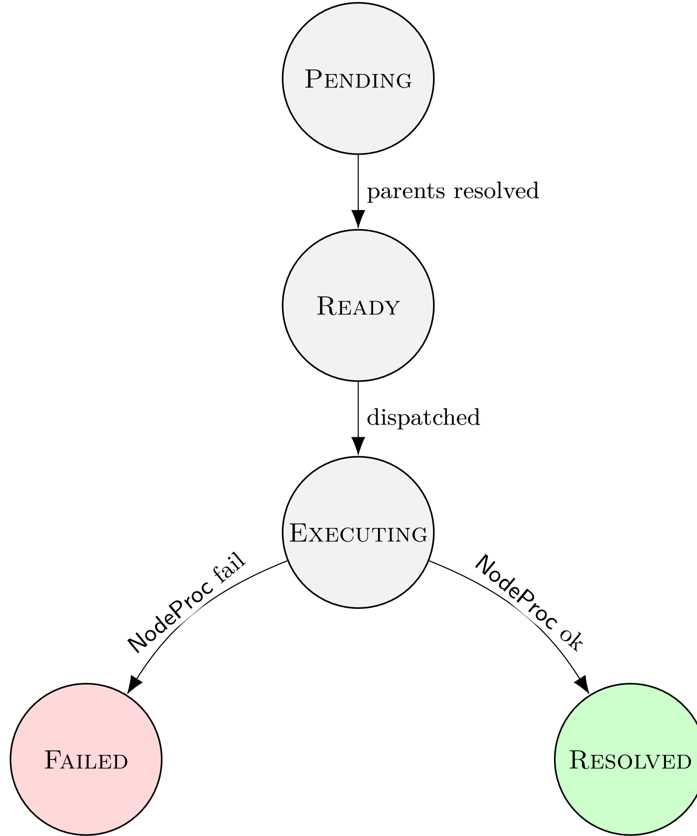

Figure 2: The GSRE node-state machine. PENDING → READY is gated by parent state; READY → EXECUTING by dispatch; EXECUTING → RESOLVED or FAILED by the outcome of the processor call.

Proposition A.5 (Per-step bounded input)**.** *The input length presented to* $\mathsf{NodeProc}_\theta$ *at any node* n *is bounded by* $|\mathsf{Enc}(\tau_n)| + |\mathsf{Enc}(\mathsf{spec}_n)| + k_{\tau_n} \cdot L_{\max}$*, independent of* $|V|$ *and* $\mathsf{depth}(G)$*.*

Theorem A.6 (Capacity–depth decoupling under independent primitive errors)**.** *Suppose* $\mathsf{NodeProc}_\theta$ *has per-invocation error rate* $\leq \varepsilon$ *and errors at distinct invocations are independent. Then* $\mathbb{P}[v_r = \hat{v}_r] \geq (1-\varepsilon)^{|V|-|V_{\mathrm{leaf}}|}$*.*

Theorem A.7 (Local-context invariance)**.** *For any two well-formed DAGs* G, G' *and any nodes* $n \in V_G$*,* $n' \in V_{G'}$ *with the same local context* $(\tau_n, \mathsf{spec}_n, (v_p)_{p\in\mathsf{pa}_n})$*, the executor produces identical values.*

Proofs are direct from the structure of Algorithm 1 and parallel earlier treatments.

Remark A.8 (Independence is the load-bearing assumption in Theorem A.6)*.* The independence assumption is unrealistic in shared-parameter models. The geometric bound is best read as a baseline against which empirical error-correlation structure is measured, not as a tight prediction. The unconditional property of the architecture is the per-step bounded input (Proposition A.5); the conditional property is the geometric decay (Theorem A.6). The empirical programme of Section 10 includes measurement of the actual error-correlation structure.

References

[1] J. C. R. Whittington, T. H. Muller, S. Mark, G. Chen, C. Barry, N. Burgess, and T. E. J. Behrens. The Tolman–Eichenbaum Machine: Unifying space and relational memory through generalization in the hippocampal formation. *Cell*, 183(5):1249–1263, 2020.

Algorithm 1 The Graph-Structured Recurrent Executor

Require: Well-formed notion DAG $G = (V, r)$ with leaf assignment σ; processor $\mathsf{NodeProc}_\theta$.
Ensure: A (possibly partial) value assignment $v : V \rightharpoonup \mathcal{V}$.

1: **for each** $n \in V$: $\mathsf{state}(n) \leftarrow$ PENDING
2: **for each** leaf $n \in V$: $v_n \leftarrow \sigma(n)$; $\mathsf{state}(n) \leftarrow$ RESOLVED
3: **loop**
4: $\quad R \leftarrow \{n \in V : \mathsf{state}(n) =$ PENDING and all $p \in \mathsf{pa}_n$ have $\mathsf{state}(p) =$ RESOLVED$\}$
5: $\quad$ **for each** $n \in R$: $\mathsf{state}(n) \leftarrow$ READY
6: $\quad$ **if** $R = \emptyset$ **then**
7: $\quad\quad$ **if** $\mathsf{state}(r) =$ RESOLVED **then**
8: $\quad\quad\quad$ **return** v
9: $\quad\quad$ **else if** $\mathsf{state}(r) =$ FAILED **or** no PENDING node has all-RESOLVED parents **then**
10: $\quad\quad\quad$ **return** v
11: $\quad\quad$ **end if**
12: $\quad$ **end if**
13: $\quad \mathcal{B} \leftarrow R$
14: $\quad$ **for each** $n \in \mathcal{B}$: $\mathsf{state}(n) \leftarrow$ EXECUTING
15: $\quad \hat{v}_{\mathcal{B}} \leftarrow \mathsf{NodeProc}_\theta\big(\{(\tau_n, \mathsf{spec}_n, (v_p)_{p \in \mathsf{pa}_n}) : n \in \mathcal{B}\}\big)$
16: $\quad$ **for** $n \in \mathcal{B}$ **do**
17: $\quad\quad$ **if** $\hat{v}_{\mathcal{B},n} \neq \perp$ **then**
18: $\quad\quad\quad v_n \leftarrow \hat{v}_{\mathcal{B},n}$; $\mathsf{state}(n) \leftarrow$ RESOLVED
19: $\quad\quad$ **else**
20: $\quad\quad\quad \mathsf{state}(n) \leftarrow$ FAILED
21: $\quad\quad$ **end if**
22: $\quad$ **end for**
23: **end loop**

[2] T. E. J. Behrens et al. What is a cognitive map? Organizing knowledge for flexible behavior. *Neuron*, 100(2):490–509, 2018.

[3] A. O. Constantinescu, J. X. O'Reilly, and T. E. J. Behrens. Organizing conceptual knowledge in humans with a gridlike code. *Science*, 352(6292):1464–1468, 2016.

[4] J. C. R. Whittington, W. Dorrell, S. Ganguli, and T. E. J. Behrens. Disentanglement with biological constraints: A theory of functional cell types. *ICLR*, 2023.

[5] E. Spens and N. Burgess. A generative model of memory construction and consolidation. *Nature Human Behaviour*, 8:526–543, 2024.

[6] D. Kumaran, D. Hassabis, and J. L. McClelland. What learning systems do intelligent agents need? Complementary learning systems theory updated. *Trends in Cognitive Sciences*, 20(7):512–534, 2016.

[7] D. Gentner. Structure-mapping: A theoretical framework for analogy. *Cognitive Science*, 7(2):155–170, 1983.

[8] B. Falkenhainer, K. D. Forbus, and D. Gentner. The structure-mapping engine: Algorithm and examples. *Artificial Intelligence*, 41(1):1–63, 1989.

[9] M. Lewis and M. Mitchell. Evaluating the robustness of analogical reasoning in large language models. *arXiv:2411.14215*, 2024.

[10] M. Mitchell. Can large language models reason? *AI Magazine*, 45(1), 2024.

[11] T. Webb, K. J. Holyoak, and H. Lu. Emergent analogical reasoning in large language models. *Nature Human Behaviour*, 7(9):1526–1541, 2023.

[12] F. Chollet. On the measure of intelligence. *arXiv:1911.01547*, 2019.

[13] F. Chollet et al. ARC Prize: Updates and the path forward. *arXiv:2412.04604*, 2024.

[14] K. Ellis et al. DreamCoder: Bootstrapping inductive program synthesis with wake–sleep library learning. *PLDI*, 2021.

[15] S. Gulwani, O. Polozov, and R. Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1–2), 2017.

[16] J. Andreas, M. Rohrbach, T. Darrell, and D. Klein. Neural module networks. *CVPR*, 2016.