Compression Enables Generalization: Wake-Sleep Cycles for Logic Programming with LLM Integration

Alexander Towell
Southern Illinois University Edwardsville
Edwardsville, IL, USA
lex@metafunctor.com
ORCID: 0000-0001-6443-9897

***Abstract*—Knowledge bases in logic programming grow through fact accumulation but do not learn: adding facts does not create understanding. We present DreamLog, a system that compresses its knowledge base through wake-sleep cycles, discovering rules that generalize to unseen entities. During wake phases, DreamLog answers queries via SLD resolution, optionally generating rules through LLM invocation for undefined predicates. During sleep phases, eight compression operations guided by the Minimum Description Length (MDL) principle transform accumulated facts into general rules: subsumption elimination, redundant fact pruning, guard-based generalization with negation-as-failure exceptions, predicate invention via skeleton fingerprinting, body pattern extraction, dead clause pruning, LLM-assisted cross-predicate rule discovery, and lemma caching. All operations are verified against a test suite of positive and negative queries with atomic rollback on failure. On a synthetic crafting domain with invented terms unknown to the LLM, compression enables $64 \pm 2\%$ recall on unseen entities (up from 0%), with symbolic compression alone achieving 53%. A raw LLM baseline (direct prompting without compression) achieves 0% recall on the same domain, confirming that the compression pipeline is genuinely necessary. On a canonical family tree domain, the full pipeline achieves 80% recall with 100% precision. In a 25-session research lab simulation, the system exhibits invest-then-compress-then-saturate dynamics matching DreamCoder's convergence behavior. These results are consistent with the Solomonoff induction thesis: shorter programs generalize better.**

***Index Terms*—compression-based learning, logic programming, wake-sleep cycles, knowledge base compression, inductive logic programming, Solomonoff induction, LLM integration**

I. Introduction

Knowledge bases in logic programming accumulate facts over time, yet this accumulation does not constitute learning. A knowledge base with 300 ground facts about a family tree cannot determine whether a newly introduced person is someone's father, even though the concepts `father`, `mother`, and `grandparent` are implicit in the data. The facts grow, but understanding does not emerge.

This paper presents an empirical test of a classical theoretical claim: *compression is learning*. Solomonoff's theory of inductive inference [1] and the closely related Minimum Description Length (MDL) principle [2], [3] assert that the shortest program consistent with observations provides the best predictions. We test this claim in a logic programming setting, where "shorter program" means fewer clauses in a knowledge base (KB), and "better predictions" means the ability to derive facts about entities never seen during training.

We introduce DreamLog, a logic programming system that implements *wake-sleep cycles* inspired by DreamCoder [4]. During wake phases, DreamLog answers queries via SLD resolution, optionally invoking a large language model (LLM) to generate rules for undefined predicates. During sleep phases, eight compression operations transform accumulated facts into general rules. The operations are guided by MDL and verified against a test suite of positive and negative queries, with atomic rollback ensuring behavioral preservation.

The central contribution is an empirical demonstration that KB compression produces rules that generalize to unseen data. We test this on two domains:

1) **A novel synthetic domain.** We construct a crafting system with 15 invented materials ("lumite," "vexal," "drossite") that no LLM has encountered in training. Compression discovers concepts such as `master_artisan` and `safe_recipe` purely from structural patterns, achieving 53% recall on unseen entities through symbolic compression alone, and $64 \pm 2\%$ with LLM-assisted operations (a raw LLM baseline achieves 0%).
2) **A canonical family tree.** On a 29-person, 4-generation family tree, the full pipeline achieves 80% recall on newly introduced individuals with 100% precision, compressing the KB from 300 to 184 clauses.

The novel domain result is the stronger validation. Because the LLM has never seen the invented terms, it cannot retrieve memorized rules from training data. The symbolic operations discover generalizations through MDL-driven compression of structural regularities in the data, without any labeled training examples or domain-specific inductive bias.

Our contributions are:

- A wake-sleep architecture for logic programming that combines eight symbolic compression operations with

LLM-assisted rule discovery, all verified against behavioral test suites.
- Empirical evidence that KB compression enables generalization to unseen entities, including on novel domains with invented vocabulary.
- An ablation study demonstrating complementary contributions: symbolic operations discover entity-level generalizations while the LLM discovers cross-predicate concepts.
- A long-lived accumulation study showing invest-then-compress-then-saturate dynamics consistent with DreamCoder's convergence.

II. Background and Related Work

A. Compression and Induction

Solomonoff's theory of inductive inference [1] formalizes the intuition that simpler explanations generalize better. The Kolmogorov complexity of a string is the length of the shortest program that produces it; Solomonoff induction predicts future data by weighting hypotheses inversely by their description length. While Kolmogorov complexity is uncomputable, the MDL principle [2], [3] provides a practical approximation: among models consistent with data, prefer the one with the shortest total description (model plus data given model). In our setting, the "model" is the set of rules in a KB, and the "data given model" is the set of remaining ground facts not derivable from those rules.

B. DreamCoder and Library Learning

DreamCoder [4] introduced wake-sleep library learning for program synthesis. During wake phases, it solves programming tasks using a library of primitives. During sleep phases, it compresses solved programs to extract reusable abstractions (via anti-unification of lambda terms), which become new library entries. Over iterations, the library converges to a small set of powerful primitives. DreamLog adapts this architecture to logic programming: our "library entries" are Horn clause rules, our "programs" are KB clauses, and our compression operations (particularly Operations C, D, and E) play the role of DreamCoder's abstraction phase.

C. Inductive Logic Programming

Inductive Logic Programming (ILP) [5] learns logic programs from labeled positive and negative examples. FOIL [6] uses information gain to grow clause bodies. Progol [7] uses inverse entailment to construct the most specific hypothesis and searches for generalizations. Metagol [8] uses meta-interpretive learning with declarative bias in the form of metarules. ILASP [9] learns Answer Set Programs from partial interpretations. Popper [10] uses hypothesis testing and constraint learning to prune the search space. ∂ILP [11] and Neural Theorem Provers [12] differentiably learn rules from examples. LINC [13] combines LLMs with first-order logic provers for reasoning.

DreamLog differs from ILP in two key respects. First, DreamLog derives its training signal from the KB itself under the closed-world assumption, rather than requiring externally provided labeled examples. All ground facts serve as positive examples, and systematically generated perturbations serve as negative examples. Second, DreamLog's learning signal is compression, not classification accuracy. Rules are accepted only if they reduce description length while preserving the deductive closure of the KB.

D. Anti-Unification

Anti-unification, introduced by Plotkin [14] and Reynolds [15], computes the least general generalization (lgg) of two terms. It is the dual of Robinson's unification [16]: where unification finds the most general term subsuming both inputs, anti-unification finds the most specific term that both inputs are instances of. DreamLog uses anti-unification in Operations C and D to identify shared structure across groups of facts and rules.

E. Predicate Invention

Predicate invention [17], [18] creates new predicates not present in the background knowledge. Metagol [8] invents predicates using metarules. Predicate invention is considered one of the most challenging aspects of ILP [19]. DreamLog's Operation D discovers structurally identical rule sets via skeleton fingerprinting and extracts parameterized predicates using `call/N` dispatch, analogous to lambda abstraction and application.

F. Neural-Symbolic Integration

DeepProbLog [20] integrates neural networks with probabilistic logic programming. Logic Tensor Networks [21] ground logical formulas in real-valued tensors. NeuralLog and related approaches [22], [23] represent relations as learned embeddings. Knowledge graph completion methods (TransE [24], RotatE [25]) learn embedding-based link prediction. DreamLog takes a different approach: the LLM is used as a hypothesis generator during compression (Operation G), but all reasoning remains symbolic. This preserves interpretability: every derived fact has a human-readable proof trace through Horn clause resolution.

G. Never-Ending Learning

NELL [26] continuously learns beliefs from the web, accumulating an ontology over years. DreamLog shares the goal of continuous knowledge accumulation but operates on a different axis: rather than acquiring new facts from external sources, it discovers structure in existing facts through compression. The two approaches are complementary.

III. System Design

DreamLog is a logic programming system with S-expression syntax that operates in alternating wake and sleep phases. This section describes the architecture and the eight compression operations that constitute the sleep phase.

Algorithm 1 Dream Cycle

Require: Knowledge base K, verification flag v
$K_0 \leftarrow \text{copy}(K)$
$S \leftarrow \text{BuildVerificationSuite}(K)$ {Pos/Neg queries}
$K \leftarrow \text{OpF}(K)$ {Dead clause pruning (wake data)}
$K \leftarrow \text{OpA}(K)$ {Subsumption elimination}
$K \leftarrow \text{OpB}(K)$ {Redundant fact pruning}
$K \leftarrow \text{OpC}(K, S)$ {Fact generalization}
$S \leftarrow \text{ExtendSuite}(S, K)$ {Add rule-derived queries}
$K \leftarrow \text{OpD}(K, S)$ {Predicate invention}
$K \leftarrow \text{OpE}(K, S)$ {Body pattern extraction}
$K \leftarrow \text{OpG}(K, S)$ {LLM-assisted compression}
$K \leftarrow \text{OpB}(K)$ {Re-prune (post-LLM)}
$K \leftarrow \text{OpH}(K)$ {Lemma caching}
if v **and** $\neg\text{Verify}(K, S)$ **then**
  $K \leftarrow K_0$ {Atomic rollback}
end if
return K, $|K|/|K_0|$

A. Wake Phase

During the wake phase, DreamLog accepts assertions and queries. Assertions add ground facts (e.g., `(parent alice bob)`) or user-defined rules to the knowledge base. Queries are answered via SLD resolution [27] with negation-as-failure (NAF) [28], supporting:

- **`not/1`**: Negation-as-failure with floundering guard.
- **`call/N`**: Meta-predicate for runtime predicate dispatch.
- **Usage tracking**: Each clause records how often it fires during resolution, providing frequency data for sleep-phase operations.

When a query references an undefined predicate, the system optionally invokes an LLM to propose rules defining that predicate. Generated rules undergo structural validation, optional LLM-as-judge verification, and are added to the KB only if they pass all checks. This mechanism transforms undefined predicates from errors into opportunities for knowledge acquisition.

B. Sleep Phase: The Dream Cycle

The sleep phase compresses the KB through eight operations, executed in sequence. Algorithm 1 outlines the overall procedure. Operations that modify the KB (C, D, E, G) perform per-candidate verification against the test suite, accepting each candidate only if it preserves behavioral equivalence. A final pipeline-level verification with atomic rollback provides an additional safety net.

Definition 1 (Verification Suite): Given a KB K, the verification suite $S = (S^+, S^-)$ consists of:

- $S^+ = \{t \mid \text{Fact}(t) \in K\}$: all ground facts.
- S^-: synthetic negative queries formed by substituting atoms into existing fact templates at positions where those atoms do not appear in K.

A compression passes verification if every $q \in S^+$ remains derivable and no $q \in S^-$ becomes derivable.

C. Operation A: Subsumption Elimination

Removes clauses subsumed by more general clauses already in the KB. A rule R_1 subsumes R_2 if there exists a substitution θ such that $R_1\theta \sqsubseteq R_2$ and both have the same body length. Additionally, bodyless rules (rules with empty bodies) subsume matching facts. This removes redundancy introduced by prior operations or user input.

D. Operation B: Redundant Fact Pruning

Removes facts that are derivable from the remaining KB (rules plus other facts). Each fact f is tentatively removed; if f remains derivable via SLD resolution, the removal is permanent. Batch removal is attempted first, with one-at-a-time fallback for mutually dependent facts. This is the primary mechanism by which rule discovery translates into fact removal.

E. Operation C: Fact Generalization with Exceptions

This is the central symbolic compression operation. Given a group of $n \geq 3$ facts sharing a functor and arity, the operation:

1) **Partitions** facts by argument position: for each position p, groups facts that agree on all arguments except position p.
2) **Finds a guard predicate**: a unary predicate g whose extension covers all values appearing at position p in the group (and possibly more).
3) **Computes exceptions**: values in the guard's extension not appearing in the fact group.
4) **Applies MDL**: the generalization is accepted only if $1 + |\text{exceptions}| < n$ (one rule plus exception facts is shorter than the original n facts).

Example 2: Given facts `(master_artisan kael)`, `(master_artisan sera)`, ..., `(master_artisan zara)` covering 7 of 9 artisans, and the guard predicate `artisan/1` covering all 9, Operation C produces:

```
(master_artisan X)
:- (artisan X), (not
(exception_master_artisan_artisan
X))
(exception_master_artisan_artisan
thom)
(exception_master_artisan_artisan
fen)
```

This replaces 7 facts with 1 rule + 2 exception facts = 3 clauses, a net reduction of 4.

F. Operation D: Predicate Invention

Discovers structurally identical rule sets across different predicates and extracts a parameterized "invented" predicate. The procedure:

1) **Extract skeletons**: For each predicate's rule set, compute a *skeleton* that abstracts functor names into roles (SELF for recursive calls, PARAM_i for distinct body functors) while preserving rule structure, arity, and variable connectivity.

2) **Group by skeleton**: Predicates with identical skeletons are structurally equivalent.
3) **Build parameterized predicate**: Create an invented predicate with an additional `call/N` dispatch parameter, and replace each original predicate with a one-line wrapper delegating to the invented predicate.
4) **Apply MDL**: Accept only if $k + m < m \cdot k$, where k is the number of rules per predicate and m is the number of predicates in the group.

This is the logic programming analog of lambda abstraction: the skeleton captures shared computational structure, and `call/N` dispatch plays the role of function application.

G. Operation E: Body Pattern Extraction

Finds common contiguous sub-goal sequences across rule bodies and extracts them as named predicates. Interface variables (those appearing both inside and outside the subsequence) become the extracted predicate's arguments. This discovers shared computational fragments across rules, analogous to common subexpression elimination.

H. Operation F: Dead Clause Pruning

Removes clauses with zero usage after sufficient wake-phase queries. Requires both a minimum query count and that at least 50% of predicates have been exercised. User-provided seed facts are protected from pruning; only clauses added by prior dream cycles (lemmas, LLM-generated rules) are eligible for removal.

I. Operation G: LLM-Assisted Compression

The symbolic operations (A–F) cannot discover rules that cross predicate boundaries (e.g., `father(X,Y) :- parent(X,Y), male(X)`) because they operate within a single functor's fact group. Operation G invokes an LLM to propose such rules.

The pipeline:

1) **Prompt construction**: Sample facts ensuring every predicate is represented (round-robin), include predicate fact counts to guide directionality (specific $\leftarrow$ general).
2) **Response parsing**: Parse JSON-formatted rule proposals, handling common LLM formatting errors.
3) **Cycle filtering**: Reject rules creating cross-functor cycles in the dependency graph via DFS. Self-recursive rules (depth-limited by the evaluator) are permitted.
4) **Helper separation**: Separate helper predicates (new predicates supporting `not/1` patterns) from main rules (deriving existing facts).
5) **Evaluation**: Each main rule must derive ≥ 2 existing facts.
6) **False-positive check**: Enumerate solutions and reject rules deriving ground terms absent from the KB.
7) **Combined verification**: Verify the full set of accepted rules against the verification suite with bounded evaluation to prevent combinatorial explosion.

J. Operation H: Lemma Caching

Adds frequently derived intermediate terms as ground facts for faster resolution. Wake-phase derivation tracking identifies terms computed repeatedly during SLD resolution. Caching these as facts converts multi-step derivations into direct lookups, yielding up to 23.6$\times$ query speedup (Section IV).

K. Behavioral Preservation

All operations share a common verification protocol. Before any modification, the system:

1) Takes a snapshot of the KB.
2) Constructs (or extends) the verification suite.
3) Applies the proposed compression.
4) Verifies that all positive queries remain derivable and no negative query becomes derivable.
5) Rolls back to the snapshot if verification fails.

This ensures that the compressed KB is *behaviorally equivalent* to the original on all tested queries. Experiment EX02 confirmed zero semantic drift across 200 held-out queries over multiple dream cycles.

IV. Experiments

We evaluate DreamLog on three axes: generalization to unseen entities (the core claim), long-lived knowledge accumulation, and ablation of the compression pipeline. All experiments use Anthropic Claude Haiku 4.5 for Operation G, with total LLM cost under $0.05 across all experiments.

A. Experimental Setup

All experiments run on DreamLog's Python implementation with SLD resolution, NAF, and `call/N`. The LLM provider for Operation G is Anthropic Claude Haiku 4.5 ($0.25/M input tokens) via API. Each dream cycle invokes the LLM at most twice (once for rule proposal in Operation G, once for predicate naming). Verification suites are constructed automatically from KB state. All compression operations use default parameters: minimum group size 3, shared structure threshold 0.1, maximum 200 prompt facts.

B. EX25b: Generalization on a Novel Domain

1) Motivation: The canonical test for compression-driven generalization (a family tree) is problematic: the LLM has likely seen `father(X,Y) :- parent(X,Y), male(X)` thousands of times during training. We cannot distinguish compression-driven discovery from training data recall. This experiment uses a synthetic domain with invented terms the LLM has never encountered.

2) Domain: An alchemical crafting workshop with 15 materials having invented names (lumite, vexal, drossite, pyreth, quarl, noctite, aerine, sylphex, thalline, morven, zephore, ethylis, fenrik, glacine, bramith) and properties (phase, material class, hazard status). The domain includes 16 recipes combining materials into products, 9 artisans with skills, and 5 skill requirements. Total: 112 base facts and 61 derived facts across 6 predicates (`hazardous_recipe`, `safe_recipe`,

TABLE I
GENERALIZATION ON THE NOVEL CRAFTING DOMAIN (EX25B). 33 CHECKS (19 POSITIVE, 14 NEGATIVE) ON UNSEEN ENTITIES. LLM CONDITIONS: MEAN ± STD OVER 5 RUNS.

Condition	Rules	Acc	Prec	Recall
No dream	0	42%	—	0%
Raw LLM	0	42%	—	0%
Symbolic	5	52%	59%	53%
Full	20	58 ± 1%	64 ± 1%	64 ± 2%

`same_phase_recipe`, `metallic_alloy`, `can_craft`, `master_artisan`).

3) Protocol: Four conditions: (1) *no dream*: baseline KB with no compression; (2) *symbolic*: dream cycle with Operations A–F only (no LLM); (3) *full pipeline*: dream cycle with all operations including LLM-assisted Operation G; (4) *raw LLM*: prompt Haiku directly with all KB facts and new-entity queries (no compression pipeline). After dreaming, 6 new materials, 8 new recipes, and 4 new artisans are introduced with base facts only. 33 checks (19 positive, 14 negative) test whether derived predicates are correctly inferred for the unseen entities. LLM-dependent conditions are run 5 times; we report mean ± standard deviation.

4) Results: Table I shows the results.

The *raw LLM* baseline achieves 0% recall: Haiku answers "NO" to every query about invented terms, confirming that direct LLM prompting contributes nothing on novel domains. The compression pipeline is genuinely necessary.

Symbolic compression alone discovers 5 rules and achieves 53% recall. The key discoveries are:

- `master_artisan(X) :- artisan(X), not(exception...)`: generalizes from the pattern that 7 of 9 artisans have 2+ skills.
- `safe_recipe(X) :- product(X), not(exception...)`: generalizes from the pattern that most recipes use non-hazardous materials.

These are discovered by Operation C through MDL-driven fact generalization. No domain knowledge is used; the system finds that "most artisans are master artisans" and "most recipes are safe" purely from the statistical structure of the data.

The full pipeline (mean of 5 runs) achieves $64 \pm 2\%$ recall with 20 discovered rules. Operation G adds cross-predicate rules including `hazardous_recipe` (via material hazard properties) and concepts like `volatile_material` and `metallic_material`. The low variance ($\pm 2\%$) confirms that results are stable across LLM generations.

Seven false positives occur from `safe_recipe` and `same_phase_recipe` generalizations applying to exception cases (e.g., a liquid-solid mix classified as same-phase). This illustrates the inherent risk of inductive generalization.

The undiscovered predicate is `can_craft(Artisan, Product)`, which requires a 3-hop join (artisan → skill → requires_skill → recipe_type). This exceeds the current operations' reach.

TABLE II
GENERALIZATION ON THE CANONICAL FAMILY TREE (EX25). THE FULL PIPELINE ACHIEVES 80% RECALL WITH 100% PRECISION.

Condition	Rules	Acc	Prec	Recall	KB Size
No dream	0	29%	—	0%	300
Symbolic	1	29%	—	0%	—
Full	7	86%	100%	80%	184

TABLE III
CROSS-DOMAIN COMPARISON OF GENERALIZATION RECALL. RAW LLM SCORES 0% ON BOTH DOMAINS.

Domain	No Dream	Raw LLM	Symbolic	Full
Crafting (novel)	0%	0%	53%	64 ± 2%
Family (canonical)	0%	0%	0%	80%

C. EX25: Generalization on a Canonical Domain

1) Domain: A 4-generation family tree with 29 people, 96 base facts (`person`, `male`/`female`, `parent`), and 204 derived facts across 7 predicates (`father`, `mother`, `grandparent`, `grandfather`, `grandmother`, `great_grandparent`, `ancestor`).

2) Protocol: Same three-condition protocol as EX25b. After dreaming, 6 new people are introduced with base facts only, and 14 checks (10 positive, 4 negative) test derived relationships.

3) Results: Table II shows the results.

The full pipeline discovers 7 rules: `father` ← `parent` + `male`, `mother` ← `parent` + `female`, `grandparent` ← `parent` ∘ `parent`, `grandfather` ← `grandparent` + `male`, `grandmother` ← `grandparent` + `female`, and variants. The KB compresses from 300 to 184 clauses (ratio 0.613). Precision is 100%: no false positives.

The sole unrecovered predicate is `ancestor`, which requires a recursive rule (`ancestor(X,Y) :- parent(X,Y)` and `ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z)`). The current LLM prompt does not propose recursive rules, and symbolic operations cannot discover them from flat fact patterns.

Symbolic compression discovers only one entity-specific rule (e.g., "albert is an ancestor of everyone") rather than the general concept. This highlights the critical role of the LLM: symbolic operations find patterns *within* a predicate, while the LLM discovers relationships *across* predicates.

D. Comparison: Novel vs. Canonical Domain

Table III compares the two domains directly.

The novel domain provides the stronger evidence for compression-driven learning. The symbolic operations achieve 50% recall on a domain with invented terms, demonstrating that MDL-guided generalization works without any prior knowledge. The family domain achieves higher overall recall (80%) but relies entirely on the LLM (Operation G), making it difficult to disentangle compression-driven discovery from LLM training data memorization.

TABLE IV
DREAM CYCLE PROGRESSION IN THE 25-SESSION RESEARCH LAB (EX24).

Cycle	Session	Before	After	Key Discoveries
1	S05	94	91	2 symbolic rules
2	S10	179	195	17 LLM cross-domain rules
3	S15	278	275	2 symbolic rules
4	S20	—	—	Saturation (minor pruning)
5	S25	—	—	No change

E. EX24: Long-Lived Knowledge Accumulation

1) Motivation: Real systems accumulate knowledge incrementally over sessions. We test whether DreamLog exhibits meaningful learning dynamics over an extended interaction.

2) Protocol: A simulated research lab scenario: "Dr. Chen" builds institutional knowledge across 25 sessions spanning two simulated years. Seven domains: people, publications, teaching, grants, infrastructure, collaborations, and service. Sessions vary from 5 to 30 assertions. Session 10 includes corrections (role changes). Session 17 includes user-provided rules with NAF helper predicates. Dream cycles occur every 5 sessions.

3) Results: The final KB contains 451 clauses (422 facts, 29 rules) from 443 total assertions. Table IV shows the dream cycle progression.

Three phases are visible:

1) **Investment** (Dream 2): The LLM discovers 17 cross-domain rules including `grant_funded_work`, `cross_institution_collaboration` (a 4-body rule with NAF), and `mentorship_chain`. The KB *expands* from 179 to 195 (ratio 1.09), investing in structural rules.
2) **Compression** (Dream 3): Symbolic operations capitalize on accumulated rules, achieving net compression.
3) **Saturation** (Dreams 4–5): No new patterns remain. The KB is fully compressed.

This invest-then-compress-then-saturate dynamic matches DreamCoder's convergence behavior [4]. The 29 discovered rules come from three sources: 4 symbolic (Operation C), 17 LLM (Operation G), and 8 user-provided. All 13 correctness checks pass, including transitive citation chains, NAF-based active membership, and negative examples. Total LLM cost: $0.023.

F. Ablation Analysis

Table V summarizes the ablation from Experiment EX08 on a 119-clause KB with diverse pattern types.

Operation C is the workhorse, contributing 85% of compression on this KB. Operation E *increases* clause count (it adds a clause for the extracted predicate) but improves structural clarity, which may benefit future compression cycles. Operations A and F contribute on KBs with subsumed or dead clauses, and Operation H contributes to query performance rather than KB size.

TABLE V
LEAVE-ONE-OUT ABLATION ON A 119-CLAUSE KB (EX08). PERCENTAGES ARE NOT ADDITIVE: EACH ROW SHOWS THE COMPRESSION LOST WHEN THAT OPERATION ALONE IS DISABLED.

Operation	Contribution	Role
C: Generalization	85% (11 clauses)	Primary compressor
B: Fact pruning	15% (2 clauses)	Post-rule cleanup
D: Predicate invention	15% (2 clauses)	Structural abstraction
E: Body extraction	−15% (+2 clauses)	Adds structural clarity
A, F, H	0%	Situational

The generalization experiments (Tables I and II) provide a complementary ablation: symbolic operations (C primarily) discover entity-level generalizations ("most artisans are masters"), while Operation G discovers cross-predicate concepts ("father = male parent"). Neither alone achieves the full pipeline's performance.

G. Additional Results

1) Convergence (EX01): The dream cycle converges in exactly 2 iterations across all tested KBs: cycle 1 compresses, cycle 2 confirms no further opportunities exist. The cycle is a monotone operator on KB size that converges quickly in practice.

2) Semantic Drift (EX02): Zero drift across 200 held-out queries over all dream cycles. The verification suite with positive and negative queries prevents any behavioral change. This is a strong safety guarantee for production use.

3) DreamCoder Benchmark (EX03): On list-processing programs (member, append, reverse, length, map, filter), the system discovers `forall/2`: a parameterized predicate equivalent to DreamCoder's "every" combinator. This validates the architectural analogy between the two systems.

4) Scaling (EX09): Compression ratio is stable at 0.88 across KBs from 69 to 1019 clauses (10 to 200 entities). Throughput degrades from 75 clauses/s to 12 clauses/s due to quadratic verification cost.

5) Noise Tolerance (EX11): Robust to 0–20% noise (incorrect facts left as isolated facts, not generalized). At 30%+ noise, Operation C generalizes erroneous patterns that share sufficient structural regularity.

6) Query Speedup (EX10): After dreaming, query time drops from 25.0ms to 1.1ms (23.6× speedup). Operation H caches 15 frequently derived lemma facts, converting multi-step recursive resolutions into direct lookups.

V. DISCUSSION

A. What Symbolic vs. LLM Operations Contribute

The experiments reveal a clean division of labor. Symbolic operations (particularly Operation C) discover generalizations *within* a predicate: "most artisans are master artisans," "most recipes are safe." These are statistical regularities visible from the distribution of ground facts. The LLM (Operation G) discovers relationships *across* predicates: "father = parent who is male," "hazardous recipe = recipe using a hazardous

material." These require understanding how distinct predicates relate semantically.

On the novel crafting domain, symbolic compression achieves 53% recall while the full pipeline reaches 64%. On the canonical family tree, symbolic compression achieves 0% recall (it finds only entity-specific patterns) while the full pipeline achieves 80%. Critically, a raw LLM baseline (prompting Haiku directly with all facts) achieves 0% on both domains, confirming that the compression pipeline provides the generalization capability, not the LLM alone. This asymmetry reflects the nature of the domains: the crafting domain has within-predicate regularities (most artisans are masters) while the family domain's generalization opportunities are entirely cross-predicate.

B. The Novel Domain as Honest Evaluation

The crafting domain result provides stronger evidence for compression-driven generalization than the family tree. When the LLM proposes `father(X,Y) :- parent(X,Y), male(X)` for a family KB, it may be recalling this rule from training data rather than discovering it from structural patterns. The invented terms in the crafting domain (lumite, vexal, drossite) eliminate this confound. That symbolic compression alone achieves 53% recall on invented terms, while the raw LLM achieves 0%, demonstrates genuine MDL-driven concept discovery independent of any LLM prior knowledge.

The family tree remains useful as a benchmark because its higher recall (80%) demonstrates the ceiling achievable when the LLM can draw on domain familiarity. The two domains together bracket the system's capability: novel domains test pure compression, canonical domains test the LLM integration.

C. Limitations

1) Multi-hop rules: The system cannot discover rules requiring 3+ hop joins. `can_craft(Artisan, Product)` requires chaining artisan $\rightarrow$ skill $\rightarrow$ requires_skill $\rightarrow$ recipe_type, which exceeds the body length of rules currently discoverable by either symbolic or LLM operations.

2) Recursive rules: The ancestor predicate (`ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z)`) is not discovered. Symbolic operations work on flat fact patterns; the LLM prompt does not currently encourage recursive proposals. Supporting recursive rule discovery is an important direction for future work.

3) Minimum data threshold: The holdout experiment (Part B of EX25) shows 0% recovery when 20% of derived facts per predicate are removed. Operation C requires a minimum group size (default 3) to trigger, and insufficient facts per pattern prevent generalization. This suggests a minimum data density below which compression-driven learning does not activate.

4) Scaling: Verification cost is quadratic in KB size due to exhaustive query checking. KBs beyond 1000 clauses would benefit from sampled verification or cached resolution.

5) LLM dependence for cross-predicate rules: The family tree ablation shows that symbolic operations alone achieve 0% generalization recall when all opportunities are cross-predicate. The system relies on the LLM for this critical capability, introducing dependence on LLM quality and availability.

D. Relation to Solomonoff Induction

Our results are consistent with the Solomonoff thesis. The compressed KBs generalize better than the uncompressed originals. However, our MDL criterion (clause count) is a coarse proxy for Kolmogorov complexity, and two domains do not establish a predictive relationship between compression ratio and generalization recall. A more faithful implementation would account for clause complexity (number of variables, body length) rather than treating all clauses as unit cost. We leave this refinement to future work.

VI. Conclusion

We have presented DreamLog, a logic programming system that discovers generalizable concepts through knowledge base compression. The central finding is that compression enables generalization: a KB compressed from 300 to 184 clauses derives facts about entities it has never seen, including on novel domains with invented vocabulary. Symbolic compression alone achieves 53% recall on such domains, while a raw LLM baseline achieves 0%, validating MDL-driven concept discovery independent of any LLM prior knowledge. The full pipeline, combining symbolic and LLM-assisted operations, achieves up to 80% recall with 100% precision on canonical domains.

The system exhibits meaningful learning dynamics over extended interactions: an invest-then-compress-then-saturate pattern consistent with DreamCoder's convergence behavior, suggesting a general property of compression-based learning systems.

Several directions remain open. Direct comparison with ILP systems (ILASP, Metagol) on shared benchmarks would position DreamLog precisely in the landscape of rule learners. Support for recursive rule discovery would address the ancestor predicate gap. Transfer learning experiments (training on one domain, compressing a combined KB with a second) would test whether compression discovers cross-domain abstractions. Scaling beyond 1000 clauses requires sampled verification. Finally, a richer MDL metric incorporating clause complexity (variable count, body length, nesting depth) would bring the system closer to Solomonoff's theoretical ideal.

The code, all experiment scripts, and the experiment registry with full provenance are available at https://github.com/queelius/dreamlog.

References

[1] R. J. Solomonoff, "A formal theory of inductive inference. part i," *Information and control*, vol. 7, no. 1, pp. 1–22, 1964.

[2] J. Rissanen, "Modeling by shortest data description," *Automatica*, vol. 14, no. 5, pp. 465–471, 1978.

[3] P. D. Grünwald, *The Minimum Description Length Principle*. MIT Press, 2007.

[4] K. Ellis, C. Wong, M. Nye, M. Sablé-Meyer, L. Morales, L. Hewitt, L. Cary, A. Solar-Lezama, and J. B. Tenenbaum, "Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning," in *International Conference on Machine Learning*. PMLR, 2021, pp. 835–850.

[5] S. Muggleton and L. De Raedt, "Inductive logic programming: Theory and methods," *The Journal of Logic Programming*, vol. 19, pp. 629–679, 1994.

[6] J. R. Quinlan, "Learning logical definitions from relations," *Machine learning*, vol. 5, no. 3, pp. 239–266, 1990.

[7] S. Muggleton, "Inverse entailment and progol," *New generation computing*, vol. 13, no. 3, pp. 245–286, 1995.

[8] S. H. Muggleton, D. Lin, N. Pahlavi, and A. Tamaddoni-Nezhad, "Meta-interpretive learning: application to grammatical inference," *Machine Learning*, vol. 94, no. 1, pp. 25–49, 2014.

[9] M. Law, A. Russo, and K. Broda, "ILASP: Learning answer set programs from examples," in *Proceedings of the 24th International Conference on Logic Programming*, 2014.

[10] A. Cropper and R. Morel, "Learning programs by learning from failures," *Machine Learning*, vol. 110, no. 4, pp. 801–856, 2021.

[11] R. Evans and E. Grefenstette, "Learning explanatory rules from noisy data," *Journal of Artificial Intelligence Research*, vol. 61, pp. 1–64, 2018.

[12] T. Rocktäschel and S. Riedel, "End-to-end differentiable proving," in *Advances in Neural Information Processing Systems*, 2017, pp. 3788–3800.

[13] T. X. Olausson, A. Gu, B. Lipkin, C. E. Zhang, A. Solar-Lezama, J. B. Tenenbaum, and R. Levy, "LINC: A neurosymbolic approach for logical reasoning by combining language models with first-order logic provers," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 5153–5176.

[14] G. D. Plotkin, "A note on inductive generalization," *Machine Intelligence*, vol. 5, pp. 153–163, 1970.

[15] J. C. Reynolds, "Transformational systems and the algebraic structure of atomic formulas," *Machine Intelligence*, vol. 5, pp. 135–151, 1970.

[16] J. A. Robinson, "A machine-oriented logic based on the resolution principle," *Journal of the ACM*, vol. 12, no. 1, pp. 23–41, 1965.

[17] I. Stahl, "Predicate invention in inductive logic programming," *Advances in Inductive Logic Programming*, pp. 34–47, 1995.

[18] S. Muggleton and W. Buntine, "Machine invention of first-order predicates by inverting resolution," in *Proceedings of the 5th International Conference on Machine Learning*, 1988, pp. 339–352.

[19] A. Cropper, S. Dumancic, R. Evans, and S. H. Muggleton, "Inductive logic programming at 30: a new introduction," *Journal of Artificial Intelligence Research*, vol. 74, pp. 765–850, 2022.

[20] R. Manhaeve, S. Dumancic, A. Kimmig, T. Demeester, and L. De Raedt, "DeepProbLog: Neural probabilistic logic programming," in *Advances in Neural Information Processing Systems*, 2018, pp. 3749–3759.

[21] L. Serafini and A. d'Avila Garcez, "Logic tensor networks: Deep learning and logical reasoning from data and knowledge," in *Neural-Symbolic Learning and Reasoning (NeSy)*, 2016.

[22] F. Yang, Z. Yang, and W. W. Cohen, "Differentiable learning of logical rules for knowledge base reasoning," in *Advances in Neural Information Processing Systems*, 2017, pp. 2319–2328.

[23] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner *et al.*, "Relational inductive biases, deep learning, and graph networks," *arXiv preprint arXiv:1806.01261*, 2018.

[24] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko, "Translating embeddings for modeling multi-relational data," in *Advances in Neural Information Processing Systems*, 2013, pp. 2787–2795.

[25] Z. Sun, Z.-H. Deng, J.-Y. Nie, and J. Tang, "RotatE: Knowledge graph embedding by relational rotation in complex space," in *International Conference on Learning Representations*, 2019.

[26] T. Mitchell, W. Cohen, E. Hruschka, P. Talukdar, B. Yang, J. Betteridge, A. Carlson, B. Dalvi, M. Gardner, B. Kisiel *et al.*, "Never-ending learning," *Communications of the ACM*, vol. 61, no. 5, pp. 103–115, 2018.

[27] J. W. Lloyd, *Foundations of Logic Programming*, 2nd ed. Springer, 1987.

[28] K. L. Clark, "Negation as failure," *Logic and Data Bases*, pp. 293–322, 1978.