Procedural Refinement by LLM-driven Algorithmic Debugging for ARC-AGI-2

Yu-Ning Qiu* **Lin-Feng Zou*** **Jiong-Da Wang**
Xue-Rong Yuan Wang-Zhou Dai†
Nanjing University
Nanjing, China
dai.wzero@gmail.com

*Equal contribution. †Corresponding author.

Abstract

In high-complexity abstract reasoning, a system must infer a latent rule from a few examples or structured observations and apply it to unseen instances. Large language models (LLMs) can express such rules as programs, but ordinary conversation-based refinement is largely outcome-level: it observes that an answer or output is wrong without formally re-checking which abstraction, relation, or transformation justified that outcome. We propose *Abduction-Based Procedural Refinement* (ABPR), a neuro-symbolic refinement approach that couples an LLM with a Prolog meta-interpreter. ABPR treats each candidate program as an executable declarative hypothesis of the latent rule and reifies its selective linear definite-clause (SLD) resolution into compact goal–subgoal derivations, following Shapiro's algorithmic program debugging (APD). The resulting traces expose intermediate decisions in the model's hypothesised rule for semantic re-checking. We evaluate ABPR primarily on ARC-AGI-2, a challenging few-shot abstract rule induction benchmark over grid transformations. On the public evaluation set, ABPR improves Pass@2 over outcome-level execution-feedback repair by 7.50, 10.00, and 10.83 percentage points for Gemini-3-Flash, Gemini-3-Pro, and GPT-5.2 low, respectively. Python checkpoint controls and resource accounting clarify the language and compute boundaries of these gains. Supplementary symbolic, generative adaptations of I-RAVEN-X and A-I-RAVEN test transfer to a second program-induction setting. The evidence is limited by public-benchmark exposure, initializer quality, and additional token cost.

1 Introduction

Although large language models (LLMs) have made remarkable progress in generating outputs that appear logically coherent and contextually plausible, they do not provide guarantees of correctness or formal validity in their reasoning processes, particularly on algorithmically structured tasks Rossi et al. [2025]. This limitation is critical when LLMs are expected to operate on complex tasks with limited supervision, especially in settings where the task abstractions and problem-solving patterns are not covered in the training data Brown et al. [2020]. Hence, the ability to recover from erroneous outputs, often referred to as "self-correction", becomes a central requirement for LLM-based systems [Kamoi et al., 2024].

However, effective correction remains a step-by-step process that requires structured search, planning, and diagnostic evidence, rather than a single conversational revision. A growing body of recent work has shown that LLMs struggle to perform such correction robustly when relying solely on

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

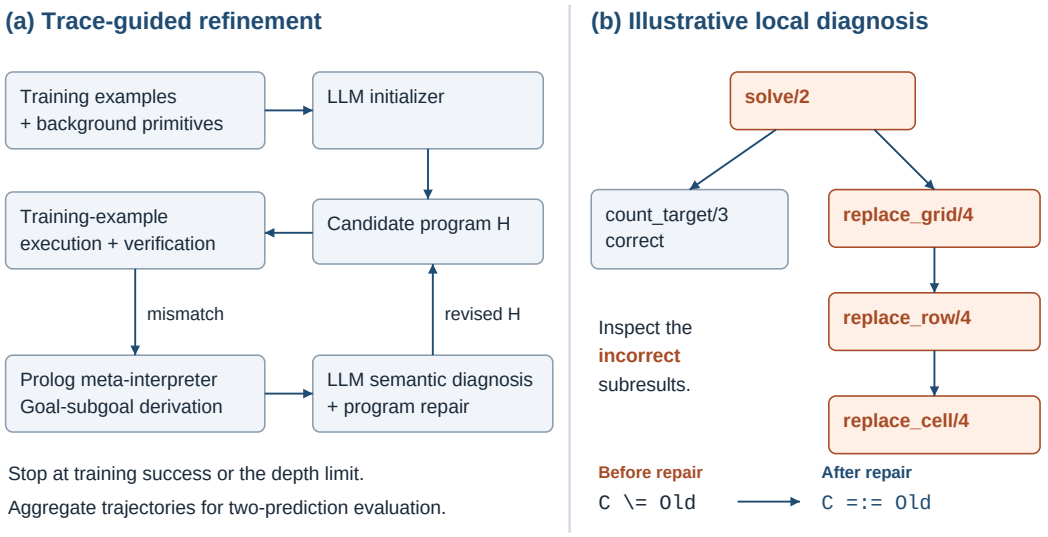

Figure 1: ABPR refinement and an illustrative local diagnosis. The meta-interpreter exposes goal–subgoal derivations for semantic inspection. Diagnosis and repair share one model request in the ARC implementation.

intrinsic or conversational feedback; in some cases, post-revision outputs are even worse than first-pass responses Huang et al. [2024], Kamoi et al. [2024], Chen et al. [2025]. Even when augmented with chain-of-thought prompting and external compilers or program executors, evidence shows that iterative LLM-based code repair can degrade across successive attempts Adnan and Kuhn [2025].

Tool-augmented correction partially addresses this problem by exposing verifiable signals such as test results, compiler messages, solver outcomes, or program executions. These signals are valuable because they ground revision in external evidence rather than in the model's own conversational judgement. However, for abstract rule induction, the object to be refined is the hypothesised rule itself. A candidate program may execute exactly as written while encoding the wrong abstraction. For example, a reachability hypothesis may define `reachable` as a two-hop relation rather than as transitive closure. A failed query or breakpoint trace can reveal a concrete failed check, but the useful diagnosis is semantic: the induced relation is under-defined and should recurse on `reachable`. This distinction, detailed in Section 3.2, motivates a refinement mechanism that inspects formal derivations of candidate rules, rather than relying only on final outcomes or execution states.

This limitation motivates revisiting a classical symbolic-AI foundation for semantic diagnosis: Shapiro's framework of *Algorithmic Program Debugging* (APD) Shapiro [1982]. APD formalizes debugging as a structured process of inspecting derivations and identifying which program judgement is inconsistent with the intended semantics, rather than performing an ad hoc sequence of revisions. For abstract rule induction, the generated program is an explicit hypothesis of a latent rule, so refinement should re-check the semantic commitments made by that hypothesis. Prolog provides a direct operational realization of this idea: SLD resolution decomposes a query into goals and subgoals, allowing a meta-interpreter to reify the reasoning process into compact proof-tree-style derivations that provide explicit semantic evidence for localized refinement.

In this paper, we formulate *procedural refinement* as the problem of revising executable hypotheses for abstract rule induction. Rather than asking an LLM to conversationally revise a final answer, we treat each candidate program as a declarative hypothesis about the latent rule and refine it through a structured, multi-step process guided by abductive reasoning [Kakas et al., 1992, Zhou, 2019]. Based on APD, we implement *Abduction-Based Procedural Refinement* (ABPR), a neuro-symbolic approach that couples an LLM with a Prolog meta-interpreter. The meta-interpreter reifies goal–subgoal resolution into compact tree-structured semantic derivations, allowing ABPR to localize and revise faulty reasoning steps in the hypothesised rule, as illustrated in Figure 1.

We evaluate ABPR primarily on ARC-AGI-2 Chollet et al. [2025], a challenging few-shot abstract rule induction benchmark where success requires discovering and applying latent grid-

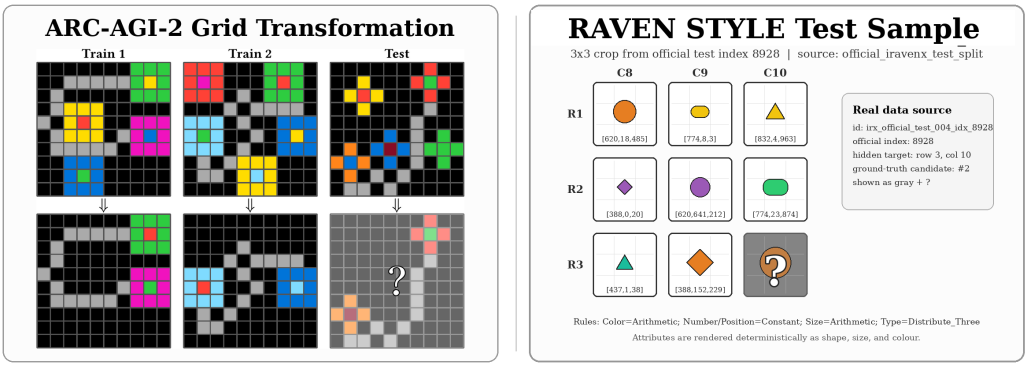

Figure 2: Representative ARC-AGI-2 grid transformation and supplementary RAVEN-style matrix reasoning tasks.

transformation rules from only a handful of examples. This setting is well suited for studying procedural refinement: failures often come from incorrect intermediate abstractions or transformation rules, while outcome-level revision provides little guidance about which part of the hypothesised rule should be revised. ABPR with Gemini-3-Flash achieves 56.67% Pass@2 on the public evaluation set, compared with 49.17% for execution-feedback repair. The corresponding trace increments are 10.00 percentage points for Gemini-3-Pro and 10.83 points for GPT-5.2 low. GPT-5.5 xHigh reaches 98.33% with ABPR; this near-ceiling result has no complete same-model execution-feedback control and is interpreted separately.

To test whether this framework is tied to ARC-specific grid transformations, we also evaluate ABPR on fill-in-the-blank adaptations of I-RAVEN-X [Camposampiero et al., 2025] and A-I-RAVEN [Malkinski and Mandziuk, 2025]. These supplementary non-ARC experiments target RAVEN-style relational and analogical abstraction, and provide evidence that the same semantic-trace-guided refinement mechanism extends beyond the ARC setting.

2 Preliminaries

2.1 Abstract rule induction

We study *abstract rule induction*: a task setting in which a system must infer a latent rule from a small number of examples or structured observations and apply that rule to unseen instances. In ARC-AGI-2, the latent rule is a grid transformation inferred from input–output examples; in RAVEN-style matrix reasoning, it is a relational or analogical rule over panel attributes. In both cases, the central difficulty is not merely producing a plausible final answer, but inducing a rule-level hypothesis whose intermediate abstractions, relations, and transformations remain valid under new cases.

In this paper, we represent such hypotheses as executable declarative programs. Let $\mathcal{B}$ denote a domain background theory, such as grid operations, object relations, or panel-attribute predicates. A candidate program P specifies a hypothesised rule over $\mathcal{B}$. Given training observations $\mathcal{E}_{\text{train}}$, the desired condition is that $\mathcal{B} \cup P$ explains the observed examples, written abstractly as $\mathcal{B} \cup P \models \mathcal{E}_{\text{train}}$. When this condition fails, the error is not necessarily a software defect in the usual sense; it may be an error in the induced abstraction, relation, or transformation rule. This is the setting in which we define procedural refinement.

2.2 Algorithmic Program Debugging

Algorithmic Program Debugging (APD), introduced by Shapiro [Shapiro, 1982], provides a formal foundation for semantic diagnosis in logic programs. Rather than treating debugging as an ad hoc inspection of variables or control flow, APD inspects the logical structure of a derivation and asks which local judgement is inconsistent with the intended semantics.

The central object in APD is a *debugging tree*, or proof tree. Its root corresponds to the initial query, internal nodes correspond to subgoals or predicate invocations, and leaves correspond to primitive facts or built-in relations. Traditionally, APD traverses this tree and queries an oracle about the semantic validity of selected nodes. In our setting, the oracle role is played by an LLM, but the object being inspected is still a formal derivation rather than a free-form conversation.

Formally, let P be a logic program and let M be the intended interpretation, i.e., the set of ground atoms that should be considered true under the target rule. For an incorrect successful derivation, a clause instance $H \leftarrow B_1, \ldots, B_k$ is locally faulty when the body literals are semantically valid, $B_i \in M$ for all i, but the head is not, $H \notin M$. This localizes the error to a single inference step: the sub-computations are accepted, but the conclusion they support is not. For missing expected answers, the complementary diagnosis concerns a required judgement in M for which the program fails to construct a valid derivation. Thus APD turns global mismatches into localized semantic questions about the candidate rule.

2.3 Procedural refinement as abduction

We formalize *procedural refinement* as abductive revision of an executable hypothesis. Let P_0 be the initial program generated by an LLM, $\mathcal{B}$ the background theory, and $\mathcal{E}_{\text{train}}$ the training observations. A refinement operator ρ maps a program to locally revised candidates, and its closure is

$$\rho^0(P_0) = \{P_0\}, \qquad \rho^{i+1}(P_0) = \bigcup_{P' \in \rho^i(P_0)} \rho(P'), \qquad \rho^*(P_0) = \bigcup_{i \geq 0} \rho^i(P_0).$$

The refinement objective is to find a program

$$P^* \in \rho^*(P_0) \quad \text{such that} \quad \mathcal{B} \cup P^* \models \mathcal{E}_{\text{train}},$$

subject to the constraint that revisions are guided by localized semantic evidence rather than unconstrained whole-program rewriting.

At iteration i, APD constructs a derivation tree for the current program P_i and identifies a small set of suspicious semantic judgements. The LLM then abduces a local correction Δ_i that would explain and repair the observed discrepancy:

$$P_{i+1} = P_i \oplus \Delta_i.$$

Here Δ_i may add, delete, or modify a predicate condition, recursive subgoal, or relation binding. This view makes refinement an abductive inference problem [Kakas et al., 1992, Zhou, 2019]: find a minimal change to the hypothesised rule that reconciles its derivation with the training observations.

2.4 Logic programming and meta-interpreters

We instantiate ABPR in Prolog [Lloyd, 2012] because its operational semantics provide a direct realization of the derivation structures required by APD. Prolog evaluates queries by SLD resolution, repeatedly reducing goals into subgoals through unification and clause selection [Robinson, 1965]. As a result, a candidate rule is not only executable; its reasoning process can also be represented as a structured goal–subgoal derivation.

We use a Prolog *meta-interpreter* to reify this process. Rather than recording a raw execution log, the meta-interpreter constructs a compact tree-structured derivation whose nodes include the current goal, relevant variable bindings, the clause instance being applied, success or failure status, and child subgoals [O'Keefe, 2009]. These proof-tree-style derivations provide the semantic evidence used by ABPR to localize faulty reasoning steps and propose targeted revisions. Other proof-producing formalisms may support analogous traces, but Prolog makes this object directly available through SLD resolution and meta-interpretation.

2.5 The ARC-AGI-2 benchmark

ARC-AGI-2 [Chollet et al., 2025] is our primary evaluation setting. Each task contains a small number of input–output grid examples and one or more test inputs. Solving a task requires inducing the latent transformation rule from the examples and applying it to the unseen test input. This

makes ARC-AGI-2 a natural test bed for procedural refinement: first-pass hypotheses often fail because they capture the wrong object abstraction, spatial relation, or transformation condition, while outcome-level feedback gives little indication of which part of the hypothesised rule should be revised.

2.6 RAVEN-style benchmarks

We additionally use fill-in-the-blank adaptations of I-RAVEN-X [Camposampiero et al., 2025] and A-I-RAVEN [Malkinski and Mandziuk, 2025] as supplementary non-ARC evaluations. These settings test whether the same semantic-trace-guided refinement mechanism extends from grid transformations to RAVEN-style relational and analogical abstraction, providing evidence that ABPR is not tied to ARC-specific grid engineering.

3 Abduction-Based Procedural Refinement

This section introduces *Abduction-Based Procedural Refinement* (ABPR), illustrated in Figure 1. ABPR gives a formal account of self-refinement for executable abstract-rule hypotheses by decomposing the global search induced by APD into a sequence of localized abductive subproblems. By representing candidate rules as declarative programs and reifying their goal–subgoal derivations with a symbolic meta-interpreter, ABPR bridges neural generative flexibility with semantic verification, yielding a refinement process that is localized and auditable.

3.1 Framework

As introduced in Section 2.3, the objective of ABPR is to find a training-consistent hypothesis H^* within the **refinement closure** $\rho^*(H_{\text{init}})$ of the initial program H_{init}, such that:

$$\mathcal{B} \cup H^* \models \mathcal{E}_{\text{train}}, \tag{1}$$

where ρ is the *refinement operator* and its iterates are defined in Section 2.3. Here H_{init} denotes the initial program P_0 in that definition.

Hypothesis initialisation. Inverse Entailment [Muggleton, 1995] motivates searching for a general rule consistent with the observations. Our implementation prompts the LLM directly with training input–output pairs, test inputs without answers, and background primitives to generate H_{init}. It does not explicitly construct a bottom clause. In ARC-AGI-2, the background primitives manipulate grid matrices, while the examples constrain the transformation that the generated Prolog program must express.

Refinement via stochastic search. Theoretically, the refinement operator $\rho(H)$ induces a *hypothesis space* of candidate programs. Traditional symbolic learning methods define a set of refinement operators that modify an initial hypothesis H—for example, by adding or deleting rules or literals—and employ a scoring function to guide the search toward improved refinements. Under this paradigm, classical APD is a deterministic procedure: assuming access to an oracle with ground-truth knowledge, it performs formal abductive reasoning over the derivation tree to identify the *minimal inconsistent subtree*.

In our framework, both the oracle and the abductive reasoner are approximated by Large Language Models (LLMs), motivated by the *discriminator–generator gap* Cobbe et al. [2021], Saunders et al. [2022], Lightman et al. [2024], which shows that verifying local reasoning steps is substantially easier than synthesising a complete reasoning trajectory. ABPR leverages this idea by reducing program synthesis/refinement to a sequence of low-entropy, verifiable sub-problems. Unlike intrinsic self-correction methods, which suffer from circularity Huang et al. [2024], our approach relies on semantic derivation traces, aligning with advances observed in recent tool-augmented critique frameworks Gou et al. [2024], Ridnik et al. [2024].

However, because LLM generation is fundamentally a process of sampling from a learned conditional probability distribution over the vocabulary given an input context Holtzman et al. [2020], integrating LLMs into the refinement loop introduces inherent stochasticity.

As a result, the originally deterministic diagnosis trajectory becomes probabilistic. A single refinement trajectory of our neuro-symbolic APD can therefore be formalized as sampling a successor

Algorithm 1 ABPR: iterative procedural refinement.

Require: Background knowledge $\mathcal{B}$; training examples $\mathcal{E}_{\text{train}}$; total search depth $T_{\text{depth}} \geq 1$; history size $K = 2$

Ensure: Refined program hypothesis $\widehat{H}$

$H_0 \leftarrow \text{Initialize}(\mathcal{B}, \mathcal{E}_{\text{train}})$
$\mathbb{H} \leftarrow \{H_0\}$
if $\text{IsConsistent}(H_0, \mathcal{E}_{\text{train}})$ **then**
    return H_0
end if
for $t = 1, \ldots, T_{\text{depth}} - 1$ **do**
    $H_b \leftarrow \text{Latest}(\mathbb{H})$
    $\tau \leftarrow \text{Trace}(H_b, \mathcal{E}_{\text{train}})$
    $H_n \sim P_{\text{APD}}(\cdot \mid H_b, \tau, \mathcal{E}_{\text{train}}, \mathcal{B})$
    if $\text{IsConsistent}(H_n, \mathcal{E}_{\text{train}})$ **then**
        return H_n
    end if
    $s(H) \leftarrow \text{Coverage}(H, \mathcal{E}_{\text{train}})$
    $\mathbb{H} \leftarrow \text{TopK}(\mathbb{H} \cup \{H_n\}, K, s)$
end for
return $\arg\max_{H \in \mathbb{H}} \text{Coverage}(H, \mathcal{E}_{\text{train}})$

hypothesis from a complex conditional distribution induced by the composition of the neural oracle and the LLM-based repairer:

$$H_{t+1} \sim P_{\text{APD}}(\cdot \mid H_t, \tau, \mathcal{E}_{\text{train}}, \mathcal{B}), \quad \text{where } H_{t+1} \in \rho^*(H_t).$$

In practice, rather than seeking a single "correct" fix, the procedure stochastically explores high-likelihood regions of the refinement closure, guided by the joint confidence of the neural components.

3.2 Declarative trace generation

To enable neuro-symbolic APD to diagnose and revise faulty reasoning steps, the system must recover the semantic structure underlying a candidate rule's derivation. We therefore represent the derivation signal τ as a **proof-tree-style semantic derivation**, adapting the classical APD formalism to link each inferred outcome with the clauses and subgoals that support it.

We extract these traces using a lightweight Prolog meta-interpreter implemented with SWI-Prolog Wielemaker et al. [2012]; the supplementary implementation material gives the core excerpt. The tracer extends standard SLD resolution by recursively constructing terms of the form `node(Type, InputSnapshot, OutputSnapshot, Children)`, where each node records the invoked predicate, its input/output bindings, and the selected sub-derivations.

This construction captures logical dependencies and variable bindings at each inference step, yielding a structured semantic proof tree. The resulting derivation tree serves as evidence for fault localisation and subsequent hypothesis refinement.

A concrete local diagnosis. In Figure 1, the `solve/2` derivation that produces an incorrect output branches to `count_target/3` and `replace_grid/4`. The count is correct, so diagnosis follows `replace_grid/4`, `replace_row/4`, and `replace_cell/4`. The latter returns `replace_cell(0,1,2,2)` and `replace_cell(1,1,2,1)`: it changes the background and leaves the target color unchanged. Replacing `C \= Old` with `C =:= Old` repairs that condition; the revised program is then checked against the training examples. The trace records execution, while the LLM's semantic diagnosis remains fallible.

3.3 The neuro-symbolic refinement procedure

ABPR proceeds as a state machine over executable hypotheses. Each transition corresponds to one iteration of the procedural refinement cycle, updating the current hypothesis from H_t to H_{t+1}.

Step 1: Fault localisation. Given a trace τ produced by a failed hypothesis H_t, the LLM Oracle performs a top-down diagnostic search over the computation tree to identify a set of *candidate faulty nodes*, denoted by $\mathcal{N}^*$. Conceptually, this step serves as a **semantic pruning mechanism**: by isolating $\mathcal{N}^*$, we condition the distribution P_{APD} on the observed logical discrepancies, thereby restricting the subsequent search to a focused subspace of the refinement closure $\rho^*(H_t)$ rather than the full program space.

Step 2: Abductive refinement. Conditioned on the identified faulty set $\mathcal{N}^*$, the refinement operator performs abductive hypothesis revision to construct the successor state. Specifically, it synthesises a modified hypothesis H_{t+1} that resolves the local inconsistencies associated with $\mathcal{N}^*$ while preserving the global structure of the program.

Remarks on the implementation. In practice, these steps are implemented as a fixed verifier-in-the-loop refinement procedure controlled by prompts. The LLM is instructed to (i) carry out APD-style inspection to localise suspicious clause instances and (ii) generate targeted revisions for the clauses associated with $\mathcal{N}^*$. Localization and repair are fused into one model request in the ARC implementation. There is no separate oracle request, and the retained resource statistics do not recover the number of internal node-level judgments. Taken together, this module functions as a stochastic sampler from the conditional distribution $P_{\text{APD}}(\cdot \mid H_t, \mathcal{E}_{\text{train}}, \mathcal{B})$, implementing semantic state transitions in the hypothesis space by coupling abductive reasoning with formal code generation.

Algorithm 1 summarizes the refinement loop. The implementation retains the history, selects the two programs with highest training coverage for the repair prompt, and presents them in chronological order. The newer selected program supplies the detailed trace. The pseudocode's buffer $\mathbb{H}$ represents this selected context. This heuristic favors high-coverage history while allowing stochastic revisions; it does not guarantee globally best-first expansion or a minimal edit.

As ABPR performs iterative exploration of a large hypothesis space, individual refinement trajectories are sensitive to initial conditions and may converge prematurely to suboptimal solutions. To improve robustness, we adopt an ensemble-based strategy inspired by parallel refinement systems Poetiq Team [2025]. Specifically, we perform a *parallel search* in the refinement hypothesis space by initiating multiple ABPR instances.

4 Experiments

We evaluate ABPR in ARC-AGI-2 (public evaluation dataset) to address the following research questions:

- **RQ1:** Does ABPR improve end-to-end accuracy on ARC-AGI-2?
- **RQ2:** How much do semantic derivation traces contribute to performance?
- **RQ3:** How sensitive is ABPR to the quality of the initial hypothesis?
- **RQ4:** Does ABPR generalise beyond ARC-style grid transformation tasks?

4.1 Settings

Each ARC-AGI-2 task contains a small number of training input–output pairs and one or more test inputs. We use the 120-task public evaluation set. We report **Pass@2** under the two-prediction constraint: for each test input, an exact output match by either of the two selected programs counts as correct. The released per-task scorer averages these indicators over that task's test inputs, allowing partial credit when a task has multiple test inputs. This is an observed two-prediction score, not a sampling-based estimator of pass@k [Chen et al., 2021].

In our experiments, we set the number of parallel threads $N = 8$ following Poetiq Team [2025], where each ABPR thread A_i is initialised with a distinct random seed ξ_i, which determines both the ordering of training examples and the stochastic choices made within the LLM-based refinement operator Dietterich [2000]. We set the sampling temperature to 1.0 and intentionally randomise the seed of each thread to increase the diversity of first-pass program hypotheses. This choice broadens the initial hypothesis distribution rather than making each single trajectory deterministic. The resulting variance is controlled by sufficient sampling over the ensemble size N and total search depth T_{depth}. Here $T_{\text{depth}} = 1$ denotes the first-pass generation only, while $T_{\text{depth}} > 1$ includes

Table 1: ARC-AGI-2 Pass@2 results under matched within-block first-pass and refinement settings. The table separates the model that produces the initial program from the model that performs repair. Cross-model rows report the initialization pool used for the corresponding cross-model repair experiment.

Experiment setting	Initial model	Repair model	Iter. repair	ABPR trace	Pass@2
Full same-model comparison: first pass → execution-feedback repair → ABPR repair					
Gemini-3-Flash: first pass	Gemini-3-Flash	–	−	−	34.17%
Gemini-3-Flash: execution-feedback repair	Gemini-3-Flash	Gemini-3-Flash	+	−	49.17%
Gemini-3-Flash: ABPR repair	Gemini-3-Flash	Gemini-3-Flash	+	+	**56.67%**
Gemini-3-Pro: first pass	Gemini-3-Pro	–	−	−	31.11%
Gemini-3-Pro: execution-feedback repair	Gemini-3-Pro	Gemini-3-Pro	+	−	43.33%
Gemini-3-Pro: ABPR repair	Gemini-3-Pro	Gemini-3-Pro	+	+	53.33%
GPT-5.2 low: first pass	GPT-5.2 low-reasoning	–	−	−	8.33%
GPT-5.2 low: execution-feedback repair	GPT-5.2 low-reasoning	GPT-5.2 low-reasoning	+	−	19.17%
GPT-5.2 low: ABPR repair	GPT-5.2 low-reasoning	GPT-5.2 low-reasoning	+	+	30.00%
Additional backbones: first pass → ABPR repair					
GPT-5.5 xHigh: first pass	GPT-5.5 xHigh	–	−	−	85.00%
GPT-5.5 xHigh: ABPR repair	GPT-5.5 xHigh	GPT-5.5 xHigh	+	+	**98.33%**
Claude-Sonnet-4.5: first pass	Claude-Sonnet-4.5	–	−	−	3.75%
Claude-Sonnet-4.5: ABPR repair	Claude-Sonnet-4.5	Claude-Sonnet-4.5	+	+	5.00%
Qwen3-Max: first pass	Qwen3-Max	–	−	−	0.83%
Qwen3-Max: ABPR repair	Qwen3-Max	Qwen3-Max	+	+	2.50%
Cross-model initialization/refinement					
Claude-Sonnet-4.5: initial program only	Claude-Sonnet-4.5	–	−	−	3.75%
Claude-Sonnet-4.5 init + Claude ABPR	Claude-Sonnet-4.5	Claude-Sonnet-4.5	+	+	5.00%
Gemini-3-Pro: initial program only	Gemini-3-Pro	–	−	−	27.92%
Gemini-3-Pro init + Claude ABPR	Gemini-3-Pro	Claude-Sonnet-4.5	+	+	42.92%
Gemini-3-Pro: initial program only	Gemini-3-Pro	–	−	−	30.83%
Gemini-3-Pro init + Qwen ABPR	Gemini-3-Pro	Qwen3-Max	+	+	48.33%

Note. Cross-model initialization-only rows report the initialization pool used for that repair run. The two Gemini-3-Pro cross-model pools are separate from each other and from the same-model pool; the Claude row reuses its same-model first-pass programs. Results should be compared within each block, rather than treating these initial scores as a common pool.

$T_{\text{depth}} - 1$ subsequent verifier-guided repair rounds. As N and T_{depth} increase, run-to-run variation empirically contracts. Appendix E.4 reports the repeated-run variance and heatmap analysis supporting this convergence, and documents the exact data sources used for reproducibility.

Then, each process A_i runs ABPR to a total search depth of $T_{\text{depth}} = 10$, corresponding to one initial generation followed by up to nine verifier-guided repair rounds. This produces a final program hypothesis H_i together with its execution outcomes on the training set. The released implementation ranks these hypotheses by training consistency and coverage, breaking remaining ties in favor of earlier successful iterations, and selects the top-2 programs. Test answers are used only for offline scoring after selection.

Feedback conditions and baselines. Our Prolog controls share the verifier, $N = 8$ initialization breadth, and top-2 selection rule. *Direct* generation stops at $T_{\text{depth}} = 1$ and receives no iterative feedback. *Execution-feedback repair* receives failed outputs, verifier outcomes, and runtime errors, and can revise the program iteratively, but does not receive APD proof-tree derivations. *ABPR repair* uses the same verifier feedback and additionally receives semantic derivation traces for localized refinement. Both repair conditions permit $T_{\text{depth}} = 10$ with the same verifier and stopping rule. Equal breadth and depth do not equate total compute: token lengths and realized call counts differ. Complete three-condition controls are available for Gemini-3-Flash, Gemini-3-Pro, and GPT-5.2 low. The Python controls in Appendix B compare outcome-level repair with injected intermediate checkpoints within Python.

4.2 RQ1: Overall performance

To assess whether ABPR improves executable abstract rule induction, we evaluate its performance across a range of LLMs with markedly different reasoning profiles. Specifically, we consider the **Gemini-3 series** (Flash and Pro Preview) Google [2025], **GPT-5.2** OpenAI [2025], **Claude-4.5 Sonnet (no thinking)** Anthropic [2025], and **Qwen3-max** Qwen Team [2025].

Table 1 reports the performance gains obtained by embedding these base models within the ABPR framework. The reported direct-to-ABPR changes are positive for the evaluated backbones, but differ substantially in magnitude. For example, **GPT-5.2** under a low reasoning-effort configuration improves from a baseline of 8.33% to **30.00%**, while **Gemini-3-Flash with ABPR** is boosted from 34.17% to **56.67%**. GPT-5.5 xHigh improves from 85.00% to 98.33%, while Claude and Qwen show smaller same-model gains. The near-ceiling GPT-5.5 result does not establish the same benefit in harder settings, and direct-to-ABPR changes combine the effects of iterative repair and trace feedback.

Figure 3 reports cumulative coverage over refinement iterations. A cumulative curve records whether a solution has been found by each depth; its monotonicity does not establish that every individual revision improves the current program.

Appendix A provides iteration curves; Appendix C separately reports calls, tokens, and cost.

4.3 RQ2: Impact of semantic derivation traces

This experiment measures the incremental effect of trace-guided feedback in the tested pipeline. We compare ABPR with execution-feedback repair on Gemini-3-Flash, Gemini-3-Pro, and GPT-5.2 low. The baseline revises programs from verifier outcomes and error logs without an APD derivation tree.

The execution-feedback repair rows in Table 1 report the performance of this baseline. ABPR adds 7.50 percentage points over execution-feedback repair for Gemini-3-Flash, 10.00 for Gemini-3-Pro, and 10.83 for GPT-5.2 low. These are within-language, same-model comparisons. They support the value of the combined trace and localization procedure in these three controls; they do not measure a universal language advantage or an equal-compute improvement. The Python variants yield similar final scores, but their prompts also change initialization, so they do not isolate the checkpoint effect (Appendix B).

4.4 RQ3: Impact of the quality of initial hypotheses

Table 1 reveals a boundary condition: ABPR is most effective when the first-pass hypothesis lies within a repairable neighbourhood of the target rule. With the same Claude ABPR refiner, the independent Gemini-3-Pro initialization pool used in the cross-model sweep improves from 27.92% before repair to 42.92% after Claude refinement, far above Claude-only ABPR at 5.00%. Thus procedural refinement is local: stronger initial rules place the search in a recoverable basin, while poor initial hypotheses may remain outside the reach of short stochastic refinement trajectories.

A second diagnostic uses Qwen3-Max to refine a separate Gemini-3-Pro pool, improving Pass@2 from 30.83% to 48.33% (+17.50 percentage points). Qwen's same-model result is 0.83% before repair and 2.50% after repair. This supports initializer quality as a bottleneck for that configuration; one additional cross-model direction does not establish a general advantage of cross-model collaboration or fully open-model generalization.

4.5 RQ4: Generalisation to RAVEN-style matrix reasoning

To test whether ABPR is tied to ARC-specific grid transformations, we further evaluate it on fill-in-the-blank adaptations of I-RAVEN-X [Camposampiero et al., 2025] and A-I-RAVEN [Malkinski and Mandziuk, 2025]. These supplementary non-ARC tasks require inferring relational rules over symbolic matrix attributes and filling a hidden entry; inputs are symbolic object, attribute, and relation descriptions, with no images supplied to the model. Candidate answers are hidden from the prompt and used only by the scoring environment. I-RAVEN-X uses adapted official symbolic instances; A-I-RAVEN-style is a custom symbolic suite.

ABPR remains effective in this setting. On I-RAVEN-X, Prolog direct and Prolog execution-feedback repair achieve 34.00% and 39.60% exact hidden-query accuracy, while Prolog ABPR achieves 88.60%. On A-I-RAVEN-style tasks, the corresponding numbers are 28.20%, 34.00%, and 93.60%. We also include Python direct and Python execution-feedback repair as stronger no-trace references: they reach 64.00%/81.60% on I-RAVEN-X and 74.00%/88.60% on A-I-RAVEN-style tasks. The comparison tests transfer to a second symbolic program-induction setting. These accuracies are not directly comparable to published visual multiple-choice results, and do not establish that the adaptations are uniformly harder. Full per-suite counts appear in Appendix E.5.

5 Related Work and Discussion

LLM self-correction is unreliable without grounded feedback [Huang et al., 2024, Kamoi et al., 2024]. Tool-augmented and program-aided methods externalize reasoning to programs or verifiers [Gao et al., 2023, Schick et al., 2023, Wei et al., 2022]; ABPR follows this direction, but studies proof-tree-style semantic derivations of executable rule hypotheses as the feedback object.

ABPR is related to automated program repair through localization, verification, and iterative revision. Recent LLM-based APR systems repair LLM-generated programs [Fan et al., 2023], combine logic-based localization with LLM repair for ASP submissions [Brancas et al., 2025], or use counterexamples and MaxSAT-based localization to constrain patches [Orvalho et al., 2025]. These works repair software or logic-program artifacts; here the program is an executable hypothesis of an abstract rule. We therefore compare feedback conditions under matched abstract reasoning protocols rather than claiming to outperform APR systems on their benchmarks. Our supplementary RAVEN settings draw on I-RAVEN-X [Camposampiero et al., 2025] and A-I-RAVEN [Malkinski and Mandziuk, 2025], and the overall approach is connected to neuro-symbolic reasoning [Hitzler and Sarker, 2021, De Raedt et al., 2025, Garcez and Lamb, 2023, Manhaeve et al., 2018, Dai et al., 2019, Badreddine et al., 2022].

Vision-Language Programs (VLP) combine learned visual descriptors with typed symbolic program synthesis [Wüst et al., 2025]. ABPR studies local revision of generated rule hypotheses using goal–subgoal derivations. Poetiq is a relevant execution-feedback system [Poetiq Team, 2025]; its externally verified semi-private result and our public-set experiments use different protocols and budgets. Appendix D records these differences without a cross-protocol ranking.

Limitations. The oracle and repairer are LLM approximations, so executable traces do not guarantee correct semantic diagnoses or minimal repairs. Weak initializers may produce few locally repairable programs. The current evidence relies largely on proprietary model services and does not establish broad generalization to fully open models. On Gemini-3-Pro, ABPR costs about 23% more per task than execution-feedback repair under the reported token-price basis. Public ARC tasks and solution code may have appeared in model training; ARC Prize discusses this exposure risk [Knoop, 2025]. We have not quantified contamination, and using Prolog does not demonstrate its absence. Cross-language scores also mix language priors and implementation choices. Generalization beyond ARC and the adapted symbolic RAVEN suites remains untested. Programming by Examples Benchmark (PBEBench) [Naik et al., 2026] and Scalable Logical Reasoning (SLR) [Helff et al., 2026] provide possible extension domains through string rewrite induction and automatically synthesized logical tasks, respectively.

6 Conclusion

We presented ABPR, a semantic-trace-guided framework for abstract rule induction. ABPR treats candidate programs as executable declarative hypotheses and uses APD-style proof-tree derivations to convert outcome-level failures into localized semantic revision problems. ARC-AGI-2 and supplementary RAVEN-style experiments show that this refinement signal improves reasoning beyond direct or execution-feedback repair.

Acknowledgments and Disclosure of Funding

This work was supported in part by JiangsuSF (BK20232003). We thank our colleagues at Dagstuhl Seminar 25491, "Approaches and Applications of Inductive Programming," for helpful discussions. The authors declare no competing interests.

References

Muntasir Adnan and Carlos CN Kuhn. Measuring and mitigating debugging effectiveness decay in code language models. *Scientific Reports*, 15(1):44120, 2025. doi: 10.1038/s41598-025-27846-5. URL `https://doi.org/10.1038/s41598-025-27846-5`.

Anthropic. Introducing Claude sonnet 4.5. Anthropic News, September 2025. URL https://www.anthropic.com/news/claude-sonnet-4-5.

Samy Badreddine, Artur S. d'Avila Garcez, Luciano Serafini, and Michael Spranger. Logic tensor networks. *Artificial Intelligence*, 303:103649, 2022.

Ricardo Brancas, Vasco Manquinho, and Ruben Martins. Combining logic and large language models for assisted debugging and repair of ASP programs. In *IEEE Conference on Software Testing, Verification and Validation, ICST 2025, Napoli, Italy, March 31 - April 4, 2025*, pages 646–657. IEEE, 2025. doi: 10.1109/ICST62969.2025.10988950. URL https://doi.org/10.1109/ICST62969.2025.10988950.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Giacomo Camposampiero, Michael Hersche, Roger Wattenhofer, Abu Sebastian, and Abbas Rahimi. I-RAVEN-X: Benchmarking generalization and robustness of analogical and mathematical reasoning in large language and reasoning models, 2025.

Aolin Chen, Haojun Wu, Qi Xin, Steven P. Reiss, and Jifeng Xuan. Studying and understanding the effectiveness and failures of conversational LLM-based repair. In *2025 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 56–59. IEEE, 2025. doi: 10.1109/APR66717.2025.00014. URL https://doi.org/10.1109/APR66717.2025.00014.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raúl Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

François Chollet, Mike Knoop, Gregory Kamradt, Bryan Landers, and Henry Pinkard. ARC-AGI-2: A new challenge for frontier AI reasoning systems. *CoRR*, abs/2505.11831, 2025. doi: 10.48550/ARXIV.2505.11831.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *CoRR*, abs/2110.14168, 2021.

Wang-Zhou Dai, Qiu-Ling Xu, Yang Yu, and Zhi-Hua Zhou. Bridging machine learning and logical reasoning by abductive learning. In *Advances in Neural Information Processing Systems 32*, pages 2811–2822. Curran Associates, Inc., 2019.

Luc De Raedt, Fredrik Heintz, Kristian Kersting, and Giuseppe Marra. *A Learning and Reasoning Manifesto*, pages 97–108. Springer Nature, Cham, Switzerland, 2025. doi: 10.1007/978-3-032-03028-3_6. URL https://doi.org/10.1007/978-3-032-03028-3_6.

Thomas G Dietterich. Ensemble methods in Machine Learning. In *Multiple Classifier Systems, First International Workshop*, pages 1–15, Cagliari, Italy, 2000. Springer.

Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. Automated repair of programs from large language models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1469–1481. IEEE, 2023. doi: 10.1109/ICSE48619.2023.00128. URL https://doi.org/10.1109/ICSE48619.2023.00128.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: program-aided language models. In *International Conference on Machine Learning, ICML 2023*, volume 202 of *Proceedings of Machine Learning Research*, pages 10764–10799. PMLR, 2023.

Artur S. d'Avila Garcez and Luis C. Lamb. Neurosymbolic AI: the 3rd wave. *Artificial Intelligence Review*, 56(11):12387–12406, 2023.

Google. A new era of intelligence with Gemini 3. Google Keyword Blog, November 2025. URL `https://blog.google/products-and-platforms/products/gemini/gemini-3/`.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: large language models can self-correct with tool-interactive critiquing. In *The Twelfth International Conference on Learning Representations, ICLR 2024*, 2024.

Lukas Helff, Ahmad Omar, Felix Friedrich, Antonia Wüst, Hikaru Shindo, Tim Woydt, Rupert Mitchell, Patrick Schramowski, Wolfgang Stammer, and Kristian Kersting. SLR: Automated synthesis for scalable logical reasoning. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 402–426, 2026. doi: 10.18653/v1/2026.acl-long.16. URL `https://aclanthology.org/2026.acl-long.16/`.

Pascal Hitzler and Md Kamruzzaman Sarker, editors. *Neuro-Symbolic Artificial Intelligence: The State of the Art*, volume 342 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021. ISBN 978-1-64368-244-0. doi: 10.3233/FAIA342. URL `https://doi.org/10.3233/FAIA342`.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *8th International Conference on Learning Representations, ICLR 2020*, 2020.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *The Twelfth International Conference on Learning Representations, ICLR 2024*, 2024.

Antonis C Kakas, Robert A. Kowalski, and Francesca Toni. Abductive logic programming. *Journal of logic and computation*, 2(6):719–770, 1992.

Ryo Kamoi, Yusen Zhang, Nan Zhang, Jiawei Han, and Rui Zhang. When can LLMs actually correct their own mistakes? A critical survey of self-correction of LLMs. *Trans. Assoc. Comput. Linguistics*, 12:1417–1440, 2024. doi: 10.1162/TACL_A_00713.

Mike Knoop. ARC Prize 2025 results and analysis. ARC Prize, December 2025. URL `https://arcprize.org/blog/arc-prize-2025-results-analysis`.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The 12th International Conference on Learning Representations*, Vienna, Austria, 2024.

John W Lloyd. *Foundations of Logic Programming*. Springer Science & Business Media, 2012.

Mikolaj Malkinski and Jacek Mandziuk. A-I-RAVEN and I-RAVEN-Mesh: Two new benchmarks for abstract visual reasoning. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence*, pages 5932–5940, 2025. doi: 10.24963/ijcai.2025/660.

Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018*, pages 3753–3763, 2018.

Stephen Muggleton. Inverse entailment and progol. *New generation computing*, 13(3):245–286, 1995.

Atharva Naik, Darsh Agrawal, Prakam, Yash Mathur, Manav Kapadnis, Yuwei An, Clayton Marr, Carolyn Rose, and David Mortensen. PBEBench: A multi-step programming by examples reasoning benchmark inspired by historical linguistics. In *Findings of the Association for Computational Linguistics: ACL 2026*, pages 8877–8918, 2026. doi: 10.18653/v1/2026.findings-acl.432. URL https://aclanthology.org/2026.findings-acl.432/.

Richard O'Keefe. *The craft of Prolog*. MIT press, 2009.

OpenAI. GPT-5 system card. Technical report, OpenAI, August 2025. URL https://openai.com/index/gpt-5-system-card/.

Pedro Orvalho, Mikolás Janota, and Vasco Manquinho. Counterexample guided program repair using zero-shot learning and maxsat-based fault localization. *CoRR*, abs/2502.07786, 2025. doi: 10.48550/ARXIV.2502.07786. URL https://doi.org/10.48550/arXiv.2502.07786.

Poetiq Team. Traversing the frontier of superintelligence. Poetiq AI Technical Blog, November 2025. URL https://poetiq.ai/posts/arcagi_announcement/.

Qwen Team. Qwen3-Max: Just scale it. Qwen Blog, September 2025. URL https://qwen.ai/blog?from=home.latest-research-list&id=87dc93fc8a590dc718c77e1f6e84c07b474f6c5a.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with alphacodium: From prompt engineering to flow engineering. *CoRR*, abs/2401.08500, 2024. doi: 10.48550/ARXIV.2401.08500.

John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1): 23–41, 1965. doi: 10.1145/321250.321253.

Francesca Rossi et al. AAAI 2025 presidential panel on the future of AI research. Technical report, Association for the Advancement of Artificial Intelligence, 2025. URL https://aaai.org/about-aaai/presidential-panel-on-the-future-of-ai-research/. Presidential panel report.

William Saunders, Catherine Yeh, Jeff Wu, Steven Bills, Long Ouyang, Jonathan Ward, and Jan Leike. Self-critiquing models for assisting human evaluators. *CoRR*, abs/2206.05802, 2022.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems 36*, New Orleans, LA, 2023.

Ehud Yehuda Shapiro. *Algorithmic Program Debugging*. Yale University, 1982.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *CoRR*, abs/2201.11903, 2022.

Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

Antonia Wüst, Wolfgang Stammer, Hikaru Shindo, Lukas Helff, Devendra Singh Dhami, and Kristian Kersting. Synthesizing visual concepts as vision-language programs, 2025. URL https://arxiv.org/abs/2511.18964.

Zhi-Hua Zhou. Abductive Learning: towards bridging machine learning and logical reasoning. *Science China Information Sciences*, 62(7):76101:1–76101:3, 2019. doi: 10.1007/S11432-018-9801-4.

A ARC Iteration Curves

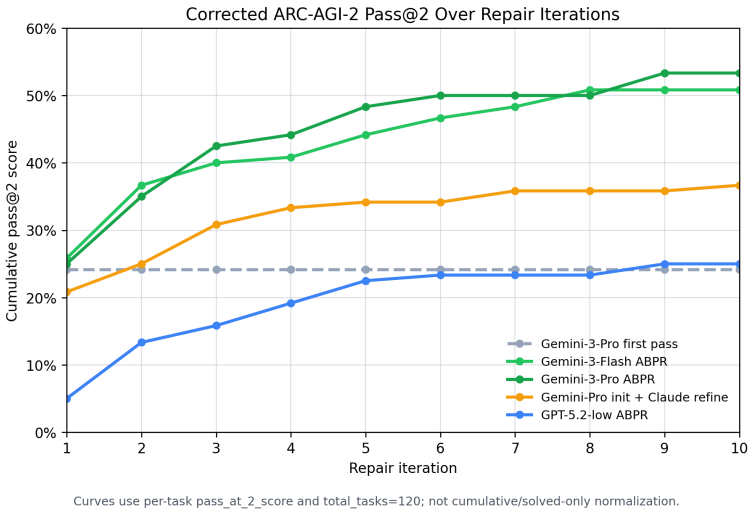

Figure 3: Representative cumulative solved-task rate over ABPR refinement iterations.

B Python Checkpoint Controls

Table 2 reports Python repair on the 120-task ARC-AGI-2 public evaluation set. Outcome-level execution-feedback repair receives output mismatches, verifier results, and runtime errors. The checkpoint condition injects intermediate test points through a decorator and an abstract syntax tree (AST) transformation. It also changes the generation prompt to request checkpoints. We therefore report each condition's own first-pass pool as well as its repaired result. All conditions use $N = 8$ and two-prediction evaluation; repair permits up to nine updates after the initial generation.

Table 2: First-pass and repaired Pass@2 (%). The Python AST prompt changes initialization as well as feedback. Prolog results provide context; cross-language absolute scores do not isolate the feedback mechanism.

	Python standard		Python AST prompt		Prolog	
Model	Direct	Feedback	Direct	Feedback	Direct	ABPR
GPT-5.2 low	22.50	33.33	16.67	31.67	8.33	30.00
Gemini-3-Flash	51.67	58.33	47.50	59.17	34.17	56.67
GPT-5.5 xHigh	92.50	93.33	93.33	95.00	85.00	98.33

The final scores of the two Python variants are close, with differences in both directions. Because their initial prompts and first-pass pools differ, this comparison does not isolate the causal effect of adding checkpoints. It also does not establish a universal difference between programming languages: language familiarity, available grid operations, and prior exposure to public tasks can affect absolute performance. The within-Prolog controls in Table 1 address the trace-guided refinement question more directly. These controls evaluate the combined trace and localization procedure; they do not separately identify every component's causal contribution.

C Resource Usage

Tables 3 and 4 report mean generation/repair requests, input tokens, and output tokens per task for the Gemini-3-Pro resource measurements. Costs are calculated at the experimental accounting rates of USD 2 per million input tokens and USD 12 per million output tokens. They are not claims about current provider prices or costs for other models.

ABPR uses 6.1% fewer requests than execution-feedback repair, but 54.8% more input tokens and 17.4% more output tokens. Its estimated cost is about 23% higher while Pass@2 increases from

Table 3: Mean Gemini-3-Pro resource usage per ARC-AGI-2 task. The price basis is fixed for this accounting comparison.

Prolog condition	Requests	Input tokens	Output tokens	USD/task
Direct	8.00	59,277	144,276	1.85
Execution-feedback repair	61.34	1,020,000	1,050,000	14.64
ABPR	57.58	1,579,291	1,233,105	17.96

43.33% to 53.33%. Oracle localization and program repair share a model request, so no separate oracle request count is added. The available statistics do not recover the number of internal node-level oracle judgments. Equal initialization breadth and maximum refinement depth therefore do not imply equal total compute or cost.

Table 4: Mean Gemini-3-Pro resource usage for Python conditions, with the same accounting rates. The direct row uses the standard generation prompt.

Python condition	Requests	Input tokens	Output tokens	USD/task
Direct (standard prompt)	8.00	108,353	209,667	2.73
Execution-feedback repair	62.80	655,168	1,255,560	16.38
AST-checkpoint feedback	65.02	1,503,745	1,326,625	18.93

D External Evaluation Protocols

Table 5 separates our measurements from externally reported results. The datasets and resource protocols differ, so the entries do not support a performance ranking.

Table 5: Protocol context for ABPR and the externally verified Poetiq result. Unreported quantities remain unspecified.

Field	ABPR (Gemini-3-Pro)	Poetiq (Gemini-3-Pro refinement)
Evaluation set	ARC-AGI-2 public, 120 tasks	ARC-AGI-2 semi-private
Reported score	53.33% Pass@2	54%, ARC Prize verified score
Search protocol	$N = 8$; top-2 selection; $T_{\text{depth}} = 10$	External refinement configuration; not aligned to our controls
Resources	57.58 requests/task; input and output tokens in Table 3	Token and request counts not specified in the cited result report
Cost	USD 17.96/task at the stated accounting rates	Approximately USD 31/task in the ARC Prize report
Source/status	This paper; author-measured public-set result	ARC Prize 2025 results report [Knoop, 2025]

Public reproducibility reports concern their specific configurations. For example, repository Issue 17 reports 19.58% for a version-a reimplementation against a reported 38.47%; this is a third-party reproduction report, not an independent replication in this paper.[1] It does not invalidate ARC Prize's separately verified result. We do not report a matched independent reproduction of Poetiq.

E Experiment Settings

In this section, we provide detailed configurations for reproducibility.

E.1 Model Configurations

We evaluate our method on several state-of-the-art Large Language Models (LLMs). All models were accessed via their official APIs.

[1] `https://github.com/poetiq-ai/poetiq-arc-agi-solver/issues/17`; accessed October 14, 2026.

- **ARC sampling temperature:** For ARC-AGI-2 experiments, we set the sampling temperature to $T_{\text{sample}} = 1.0$ for all models to encourage exploration during the sampling and refinement phases. RAVEN-style experiments use the separate setting reported in Appendix E.5.
- **Reasoning Effort:** GPT-5.2 uses the low reasoning-effort configuration; GPT-5.5 uses the xHigh configuration identified in the corresponding result rows.
- **Other Parameters:** For all other models (including Gemini-3 series, Claude series, and Qwen series), we utilized the standard ChatCompletion mode with default reasoning configurations provided by the respective official APIs.

E.2 Parallel Search and Evaluation Protocol

- **Ensemble Strategy ($N = 8$):** To robustly solve the tasks, we deploy $N = 8$ parallel ABPR processes. Each process operates independently to explore the hypothesis space.
- **Selection Mechanism:** The released implementation ranks the $N = 8$ final hypotheses using training consistency, coverage, and iteration order, then selects two. Test labels do not determine this ranking.
- **Pass@2 Evaluation:** Each test input is correct when either selected program produces the exact output. The per-task score averages over its test inputs. This measures the observed two-prediction result rather than a statistical estimate from random sampling.
- **Search Depth:** We use a total-depth convention: $T_{\text{depth}} = 1$ denotes the initial generation only, and $T_{\text{depth}} > 1$ includes $T_{\text{depth}} - 1$ subsequent repair rounds. The main ARC-AGI-2 setting uses $T_{\text{depth}} = 10$, i.e., one initial generation plus up to nine repair rounds. For ablation studies marked as "Iteration=1", the refinement loop is disabled.

E.3 Prompting and Randomness

- **Prompt Construction:** We utilize few-shot prompting where demonstration examples are retrieved from the training split.
- **Random Seeding:** Each of the $N = 8$ parallel processes is initialised with a distinct random seed. This seed controls the permutation of few-shot examples in the context window to induce search diversity across the ensemble, preventing the parallel threads from collapsing into identical refinement trajectories.

E.4 ARC Reproducibility and Variance Analysis

The ARC-AGI-2 experiments deliberately use stochastic sampling. We set the sampling temperature to $T_{\text{sample}} = 1.0$ and use different random seeds across parallel threads so that the first-pass generator explores a broader distribution of candidate programs. The evaluation is therefore not intended to make each single trajectory deterministic; instead, reproducibility is obtained by aggregating sufficiently many independent samples across ensemble size N and total search depth T_{depth}.

Variance metrics. For a fixed setting, let s_r denote the Pass@2 score, in percentage points, of repeat r. We report the repeated-run mean $\bar{s} = \frac{1}{R}\sum_{r=1}^{R} s_r$ and sample standard deviation $\sqrt{\frac{1}{R-1}\sum_{r=1}^{R}(s_r - \bar{s})^2}$. For the heatmap analysis, we also report the repeated-run range $\max_r s_r - \min_r s_r$. Lower standard deviation and lower range indicate higher reproducibility.

Evaluation data and repeated-run protocol. The reproducibility heatmaps are computed on the same 120-task ARC-AGI-2 public evaluation set used in the main experiments. For Gemini-3-Pro and Gemini-3-Flash, we repeat the experiment 10 times for each pair of ensemble size N and total search depth T_{depth}, then compute the standard deviation and range for each model and average the two values cell-wise. All reported non-Gemini ARC result means and standard deviations are computed from three independent repeated runs under the corresponding setting. Table 6 summarises the aggregate repeated-run statistics, while Figure 4 reports the separate 10-run Gemini reproducibility sweeps over (N, T_{depth}).

Table 6: Repeated-run ARC-AGI-2 Pass@2 variance summary. Scores are percentages; Std. is the sample standard deviation across repeated runs.

Setting	Mean	Std.
Gemini-3-Flash ABPR	56.25	0.59
Gemini-3-Pro ABPR	52.29	1.05
GPT-5.2 low-reasoning ABPR	29.93	0.98
Claude-Sonnet-4.5 ABPR	4.08	0.37
Qwen3-Max ABPR	2.77	0.78
GPT-5.5 xHigh ABPR	98.05	0.48

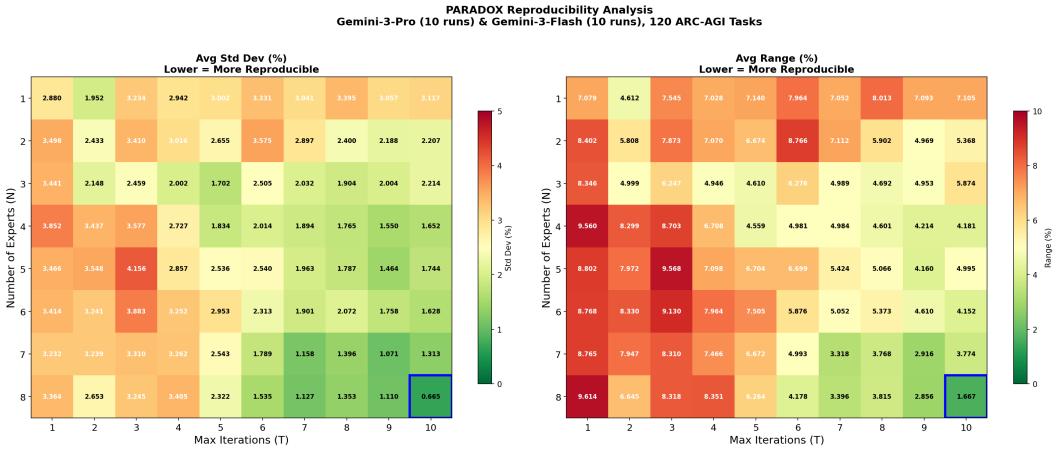

Figure 4: ARC-AGI-2 reproducibility heatmap. Each cell is averaged over Gemini-3-Pro and Gemini-3-Flash after 10 repeated runs per model at the corresponding ensemble size N and total search depth T_{depth}. Here $T_{\text{depth}} = 1$ denotes first-pass generation only. Lower values indicate more stable repeated performance.

Figure 4 reports the full reproducibility heatmap over N and T_{depth}. The left panel is the cell-wise averaged standard deviation and the right panel is the cell-wise averaged range, computed from the 10 repeated Gemini-3-Pro and Gemini-3-Flash sweeps. The lower-right region shows that larger N and T_{depth} substantially reduce run-to-run variability; at $N = 8, T_{\text{depth}} = 10$, the averaged standard deviation is 0.665 percentage points and the averaged range is 1.667 percentage points. This supports the design choice of using sufficient parallel sampling and iterative refinement rather than relying on a single stochastic trajectory.

E.5 RAVEN-style Generalisation Experiments

The RAVEN-style experiments in Table 7 are adapted to the same verifier-in-the-loop refinement protocol as the ARC-AGI-2 experiments. They are not evaluated under the original visual candidate-selection setup. Instead, the model must write or repair a solver that infers the hidden matrix entry from visible symbolic facts.

- **I-RAVEN-X interactive suite:** We evaluate on 500 official I-RAVEN-X test-split symbolic matrix instances adapted to an interactive fill-in format. Each instance uses $n = 10$, $n_{\text{show}} = 3$, `maxval`=1000, and no confounding attributes in the reported final suite. Candidate answers are available to the runner only for scoring and are hidden from the model prompt.
- **AIRAVEN-style suite:** We evaluate on a 500-query symbolic suite derived from I-RAVEN-X primitives and A-I-RAVEN-style held-out-attribute regimes. The suite is grouped across ten held-out regimes, with 4 visible training puzzles and 10 hidden puzzles per generated task group. The generator uses $n = 10$, $n_{\text{show}} = 3$, `maxval`=1000, `nconf`=0, and seed 20260501.
- **Backbone and sampling:** All reported RAVEN-style Prolog experiments use the same Qwen3-Coder backend, temperature $T = 0.5$, and 20-way worker parallelism.

- **Compared methods:** *Prolog direct* performs a single program-generation attempt. *Execution-feedback repair* starts from the direct result and refines using final execution feedback only. *ABPR repair* starts from the same direct result but additionally uses declarative trace-based oracle guidance and local procedural repair.
- **ABPR limits:** For the reported RAVEN-style runs, ABPR is limited to at most two refinement rounds and at most five oracle questions per task instance. These limits are intentionally small to test whether semantic traces improve the local refinement process rather than relying on unlimited resampling.
- **Evaluation metric:** We report exact hidden-query accuracy. A prediction is correct only if the produced Prolog solver returns the withheld expected answer for the hidden matrix entry. Candidate identities are used by the runner to score the answer, but are not exposed as a candidate-ranking prompt.

Table 7: Exact hidden-query accuracy on symbolic, generative RAVEN adaptations. The model receives no images or answer candidates. The A-I-RAVEN-style suite is custom; these rows are not published visual multiple-choice benchmark scores.

Method	**I-RAVEN-X interactive**		**A-I-RAVEN-style**	
	Correct queries	**Acc.**	**Correct queries**	**Acc.**
Prolog direct	170/500	34.00%	141/500	28.20%
Prolog execution-feedback repair	198/500	39.60%	170/500	34.00%
Prolog ABPR repair	443/500	**88.60%**	468/500	**93.60%**
Python direct	320/500	64.00%	370/500	74.00%
Python execution-feedback repair	408/500	81.60%	443/500	88.60%

F Broader Impacts

The derivation traces can make generated-program failures easier to inspect. They do not certify that the intended rule is correct or that a repaired program is safe in deployment. A mistaken LLM diagnosis may produce a plausible but incorrect repair, and stronger automated code generation can also support harmful software. Applications require task-specific validation, execution isolation, and human review appropriate to their consequences.

G Example: Execution Debugging vs. Semantic Re-checking

Consider a graph-reachability rule over the chain $a \to b \to c \to d$. The intended abstraction is transitive closure: `reachable(a,d)` should hold. A Python implementation may instead encode only two-hop reachability:

```
def reachable(x, y):
    if edge(x, y):
        return True
    for z in successors(x):
        if edge(z, y):      # should recurse on reachable(z, y)
            return True
    return False
```

This program executes normally. A breakpoint debugger can show the concrete state `z=b` and the failed check `edge(b,d)`, but the semantic problem is that `reachable` has been defined as direct-or-two-hop reachability rather than transitive closure.

In Prolog, the same erroneous hypothesis can be written as:

```
reachable(X, Y) :- edge(X, Y).
reachable(X, Y) :- edge(X, Z), edge(Z, Y).
```

For the query `reachable(a,d)`, a meta-interpreter reifies the failed goal–subgoal derivation:

```
reachable(a,d)
  -> edge(a,d) fails
  -> edge(a,b) succeeds
  -> edge(b,d) fails
```

The derivation exposes the rule-level error: the second clause tests `edge(Z,Y)` instead of the recursive subgoal `reachable(Z,Y)`. ABPR therefore revises the semantic definition of the hypothesised relation, rather than patching a single observed execution.

H Declarative Trace Meta-Interpreter

ABPR reifies proof-tree-style semantic derivations with a Prolog meta-interpreter rather than by asking the model to narrate its own execution. The implementation used in our experiments builds proof-tree nodes of the form `node(Type, InputSnapshot, OutputSnapshot, Children)`. A node therefore records the predicate being resolved, the bindings before and after resolution, and the recursively selected sub-goals that explain the produced output. The full implementation additionally imposes a maximum trace depth, suppresses uninformative system and background-knowledge predicates, expands selected domain-specific calls such as `bk:map_grid_cells/3`, and prints the result as a compact declarative proof tree. The core functionality is shown below.

```
% Core excerpt of the SWI-Prolog trace meta-interpreter.
% Full code also includes depth limits, output summarisation,
% background-predicate filtering, and formatted proof-tree printing.

solve_and_trace(Goal, TraceTree) :-
    solve_with_depth(Goal, 0, TraceTree).

solve_with_depth(Goal, Depth, TraceTree) :-
    max_trace_depth(MaxDepth),
    ( Depth > MaxDepth ->
        copy_term(Goal, Input),
        call(Goal),
        copy_term(Goal, Output),
        TraceTree = node(depth_limit, Input, Output, [])
    ; interpret_goal(Goal, Depth, TraceTree)
    ).

interpret_goal(true, _Depth, node(builtin, true, true, [])) :- !.
```

```
interpret_goal(fail, _Depth, _) :- !, fail.
interpret_goal(!, _Depth, node(builtin, '!', '!', [])) :- !.

interpret_goal((A, B), Depth,
               node(conj, (A, B), (A1, B1), [TreeA|TreesB])) :- !,
    solve_with_depth(A, Depth, TreeA),
    TreeA = node(_, _, A1, _),
    interpret_conjunction_rest(B, Depth, TreesB, B1).

interpret_goal((A ; B), Depth, TraceTree) :- !,
    ( solve_with_depth(A, Depth, TraceTree)
    ; solve_with_depth(B, Depth, TraceTree)
    ).

interpret_goal((Cond -> Then ; Else), Depth, TraceTree) :- !,
    ( solve_with_depth(Cond, Depth, CondTree) ->
        solve_with_depth(Then, Depth, ThenTree),
        ThenTree = node(_, _, Result, _),
        TraceTree = node(if_then_else,
                         (Cond -> Then ; Else),
                         Result,
                         [CondTree, ThenTree])
    ; solve_with_depth(Else, Depth, TraceTree)
    ).

interpret_goal(\+(A), _Depth, node(negation, \+(A), success, [])) :- !,
    \+ call(A).

interpret_goal(findall(Template, Goal, List), _Depth,
               node(findall,
                    findall(TemplateCopy, GoalSummary, _),
                    findall(TemplateCopy, GoalSummary, ListSummary),
                    [])) :- !,
    copy_term(Template, TemplateCopy),
    summarize_goal(Goal, GoalSummary),
    findall(Template, call(Goal), List),
    summarize_list_result(List, ListSummary).

interpret_goal(Goal, Depth,
               node(Functor/Arity, InputSnap, OutputSnap, Children)) :-
    functor(Goal, Functor, Arity),
    copy_term(Goal, InputSnap),
    can_access_clauses(Goal),
    clause(Goal, Body),
    Depth1 is Depth + 1,
    expand_body(Body, Depth1, Children),
    copy_term(Goal, OutputSnap).

expand_body(true, _Depth, []) :- !.
expand_body((A, B), Depth, Children) :- !,
    expand_body(A, Depth, ChildrenA),
    expand_body(B, Depth, ChildrenB),
    append(ChildrenA, ChildrenB, Children).
expand_body(Goal, Depth, [Tree]) :-
    should_trace_goal(Goal),
    solve_with_depth(Goal, Depth, Tree).
expand_body(Goal, _Depth, []) :-
    call(Goal).
```

I Working examples

I.1 Task b0039139

Code of Iteration 1:

```
:- use_module(bk).

solve(Input, Output) :-
    % 1. Determine if separators are horizontal or vertical
    (   is_horizontal(Input)
    ->  bk:grid_dimensions(Input, H, W),
        findall(R, (between(1, H, R), is_separator_row(Input, R)), Seps),
        split_horizontal(Input, Seps, Regions),
        maplist(transpose_grid, Regions, TRegions),
        Transposed = true
    ;   bk:grid_dimensions(Input, H, W),
        findall(C, (between(1, W, C), is_separator_col(Input, C)), Seps),
        split_vertical(Input, Seps, TRegions),
        Transposed = false
    ),

    % 2. Extract R1, R2, R3, R4
    TRegions = [R1, R2, R3, R4],

    % 3. Extract Kernel K from R1 (color 4)
    bk:connected_components(R1, 4, KComps),
    extract_all_cells(KComps, KCells),
    bk:component_bbox(KCells, bbox(MinR1, MaxR1, MinC1, MaxC1)),
    HK is MaxR1 - MinR1 + 1,
    WK is MaxC1 - MinC1 + 1,

    % 4. Extract Points S from R2 (color 3)
    bk:connected_components(R2, 3, SComps),
    extract_all_cells(SComps, SCells),
    findall(C, member((_, C), SCells), SCols),
    sort(SCols, SortedSCols),

    % 5. Extract target colors C3 and C4
    extract_block_color(R3, C3),
    extract_block_color(R4, C4),

    % 6. Calculate output dimensions and kernel positions
    (   SortedSCols = [] -> Output = [] ;
        min_list(SortedSCols, MinSC),
        calculate_positions(SortedSCols, MinSC, WK, 0, IndexedPositions, MaxPos)
    ,
        OutW is MaxPos + WK - 1,
        OutH is HK,

        % 7. Create and fill output grid
        length(EmptyRows, OutH),
        maplist(fill_row_with_dummy(OutW), EmptyRows),
        bk:map_grid_cells(EmptyRows, mapper(IndexedPositions, R1, MinR1, MinC1,
    WK, C3, C4), FilledGrid),

        % 8. Final Transpose if necessary
        (   Transposed = true
        ->  transpose_grid(FilledGrid, Output)
        ;   Output = FilledGrid
        )
    ).
```

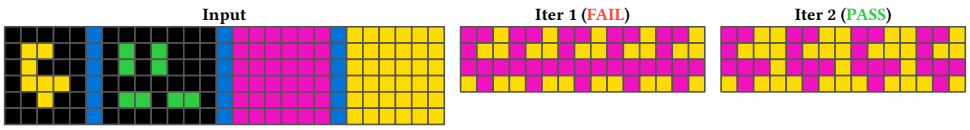

Figure 5: An iteration example for task b0039139.

```
% Helper predicates

is_horizontal(Grid) :-
    bk:grid_dimensions(Grid, H, W),
    between(1, H, R),
    is_separator_row(Grid, R), !.

is_separator_row(Grid, R) :-
    bk:grid_dimensions(Grid, _, W),
    forall(between(1, W, C), (bk:grid_cell(Grid, R, C, V), V =:= 1)).

is_separator_col(Grid, C) :-
    bk:grid_dimensions(Grid, H, _),
    forall(between(1, H, R), (bk:grid_cell(Grid, R, C, V), V =:= 1)).

split_horizontal(Grid, Seps, Regions) :-
    bk:grid_dimensions(Grid, H, W),
    append([0], Seps, StartSeps),
    append(Seps, [H], EndSeps),
    findall(Region, (
        nth0(I, StartSeps, S), nth0(I, EndSeps, E),
        S1 is S + 1, E1 is E - 1,
        S1 =< E1,
        findall(Row, (
            between(S1, E1, R),
            findall(V, (between(1, W, C), bk:grid_cell(Grid, R, C, V)), Row)
        ), Region)
    ), Regions).

split_vertical(Grid, Seps, Regions) :-
    bk:grid_dimensions(Grid, H, W),
    append([0], Seps, StartSeps),
    append(Seps, [W], EndSeps),
    findall(Region, (
        nth0(I, StartSeps, S), nth0(I, EndSeps, E),
        S1 is S + 1, E1 is E - 1,
        S1 =< E1,
        findall(Row, (
            between(1, H, R),
            findall(V, (between(S1, E1, C), bk:grid_cell(Grid, R, C, V)), Row)
        ), Region)
    ), Regions).

transpose_grid(Grid, TGrid) :-
    bk:grid_dimensions(Grid, H, W),
    findall(Row, (
        between(1, W, C),
        findall(V, (between(1, H, R), bk:grid_cell(Grid, R, C, V)), Row)
    ), TGrid).

extract_all_cells(Comps, AllCells) :-
    findall(Cell, (member(component(_, Cells), Comps), member(Cell, Cells)),
    AllCells).

extract_block_color(Grid, Color) :-
    flatten(Grid, Flattened),
    include(is_not_bg_or_sep, Flattened, NonBg),
    (NonBg = [Color|_] ; Color = 0).

is_not_bg_or_sep(V) :- V \= 0, V \= 1.

calculate_positions([], _, _, _, [], 0).
calculate_positions([C|Cs], MinSC, WK, I, [Pos|Rest], MaxPos) :-
```

```
    Pos is (C - MinSC) + I * (WK - 1) + 1,
    I1 is I + 1,
    calculate_positions(Cs, MinSC, WK, I1, Rest, MaxPos0),
    MaxPos is max(Pos, MaxPos0).

fill_row_with_dummy(W, Row) :-
    length(Row, W),
    maplist(=(0), Row).

mapper(Positions, R1, MinR1, MinC1, WK, C3, C4, R, C, _, Val) :-
    (   is_kernel_pixel(R, C, Positions, R1, MinR1, MinC1, WK)
    ->  Val = C3
    ;   Val = C4
    ).

is_kernel_pixel(R, C, Positions, R1, MinR1, MinC1, WK) :-
    member(StartCol, Positions),
    C >= StartCol, C < StartCol + WK,
    R_in is R + MinR1 - 1,
    C_in is C - StartCol + MinC1,
    bk:grid_cell(R1, R_in, C_in, 4).
```

Code of Iteration 2:

```
:- use_module(bk).

% Main solver predicate
solve(Input, Output) :-
    % 1. Determine separator orientation and split into regions
    (   is_horizontal(Input)
    ->  bk:grid_dimensions(Input, H, W),
        findall(R, (between(1, H, R), is_separator_row(Input, R)), Seps),
        split_horizontal(Input, Seps, Regions),
        maplist(transpose_grid, Regions, TRegions),
        Transposed = true
    ;   bk:grid_dimensions(Input, H, W),
        findall(C, (between(1, W, C), is_separator_col(Input, C)), Seps),
        split_vertical(Input, Seps, TRegions),
        Transposed = false
    ),

    % 2. Identify the four regions: Kernel, Markers, ColorBlock1, ColorBlock2
    TRegions = [R1, R2, R3, R4],

    % 3. Extract the kernel dimensions and bounding box from R1 (color 4)
    bk:connected_components(R1, 4, KComps),
    extract_all_cells(KComps, KCells),
    bk:component_bbox(KCells, bbox(MinR1, MaxR1, MinC1, MaxC1)),
    HK is MaxR1 - MinR1 + 1,
    WK is MaxC1 - MinC1 + 1,

    % 4. Count the number of marker objects in R2 (color 3)
    bk:connected_components(R2, 3, Markers),
    length(Markers, N),

    % 5. Extract target colors C3 and C4 from R3 and R4
    extract_block_color(R3, C3),
    extract_block_color(R4, C4),

    % 6. Calculate output dimensions (tiled horizontally in the transposed space
    )
    OutW is N * WK + (N - 1),
```

```
    OutH is HK,

    % 7. Create and fill the output grid tile by tile
    length(EmptyRows, OutH),
    maplist(fill_row_with_dummy(OutW), EmptyRows),
    bk:map_grid_cells(EmptyRows, mapper(N, HK, WK, R1, MinR1, MinC1, C3, C4),
    TOutput),

    % 8. Transpose back if the original orientation was horizontal
    (   Transposed = true
    ->  transpose_grid(TOutput, Output)
    ;   Output = TOutput
    ).

% Helper: Check if grid has horizontal separators
is_horizontal(Grid) :-
    bk:grid_dimensions(Grid, H, _),
    between(1, H, R),
    is_separator_row(Grid, R), !.

is_separator_row(Grid, R) :-
    bk:grid_dimensions(Grid, _, W),
    forall(between(1, W, C), (bk:grid_cell(Grid, R, C, V), V =:= 1)).

is_separator_col(Grid, C) :-
    bk:grid_dimensions(Grid, H, _),
    forall(between(1, H, R), (bk:grid_cell(Grid, R, C, V), V =:= 1)).

% Split grid by horizontal separators
split_horizontal(Grid, Seps, Regions) :-
    bk:grid_dimensions(Grid, H, W),
    append([0], Seps, StartSeps),
    append(Seps, [H], EndSeps),
    findall(Region, (
        nth0(I, StartSeps, S), nth0(I, EndSeps, E),
        S1 is S + 1, E1 is E - 1,
        S1 =< E1,
        findall(Row, (
            between(S1, E1, R),
            findall(V, (between(1, W, C), bk:grid_cell(Grid, R, C, V)), Row)
        ), Region)
    ), Regions).

% Split grid by vertical separators
split_vertical(Grid, Seps, Regions) :-
    bk:grid_dimensions(Grid, H, W),
    append([0], Seps, StartSeps),
    append(Seps, [W], EndSeps),
    findall(Region, (
        nth0(I, StartSeps, S), nth0(I, EndSeps, E),
        S1 is S + 1, E1 is E - 1,
        S1 =< E1,
        findall(Row, (
            between(1, H, R),
            findall(V, (between(S1, E1, C), bk:grid_cell(Grid, R, C, V)), Row)
        ), Region)
    ), Regions).

% Standard matrix transpose
transpose_grid(Grid, TGrid) :-
    bk:grid_dimensions(Grid, H, W),
    findall(Row, (
        between(1, W, C),
        findall(V, (between(1, H, R), bk:grid_cell(Grid, R, C, V)), Row)
```

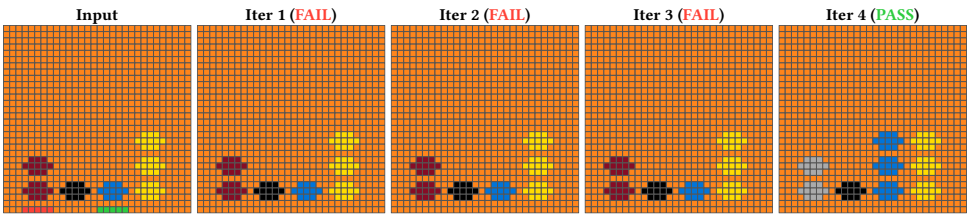

Figure 6: An iteration example for task 9aaea919.

```
    ), TGrid).

% Collect all cells from a list of components
extract_all_cells(Comps, AllCells) :-
    findall(Cell, (member(component(_, Cells), Comps), member(Cell, Cells)),
    AllCells).

% Extract the dominant color from a block region
extract_block_color(Grid, Color) :-
    flatten(Grid, Flattened),
    include(is_not_bg_or_sep, Flattened, NonBg),
    (   NonBg = [Color|_]
    ->  true
    ;   Color = 0
    ).

is_not_bg_or_sep(V) :- V \= 0, V \= 1.

% Create a row of a specific width filled with 0s
fill_row_with_dummy(W, Row) :-
    length(Row, W),
    maplist(=(0), Row).

% Mapper function for tiling the kernel in the output grid
mapper(N, _HK, WK, R1, MinR, MinC, C3, C4, R, C, _OldVal, NewVal) :-
    TileIndex is (C - 1) // (WK + 1),
    RelC is (C - 1) mod (WK + 1) + 1,
    (   RelC > WK
    ->  NewVal = C4
    ;   (   TileIndex < N
        ->  (   R_in is R + MinR - 1,
                C_in is RelC + MinC - 1,
                bk:grid_cell(R1, R_in, C_in, 4)
            ->  NewVal = C3
            ;   NewVal = C4
            )
        ;   NewVal = C4
        )
    ).
```

I.2 Task 9aaea919

Code of Iteration 1:

```
:- use_module(bk).

% Main solve predicate
solve(Input, Output) :-
    bk:grid_cell(Input, 1, 1, BG),
    bk:connected_components(Input, nonzero, AllComps),
    % Filter out background components
```

```
    findall(component(V, Cells), (member(component(V, Cells), AllComps), V \= BG
    ), Shapes),

    % Bars are at row 30, centered at columns 6, 12, 18, 24
    findall(Col, (member(Col, [6, 12, 18, 24]), bk:grid_cell(Input, 30, Col, 2))
    , RedBars),
    findall(Col, (member(Col, [6, 12, 18, 24]), bk:grid_cell(Input, 30, Col, 3))
    , GreenBars),

    % Count shapes above all Red bars
    findall(S, (member(C, RedBars), shapes_above(C, Shapes, Ss), member(S, Ss)),
     AllRedShapes),
    length(AllRedShapes, N_Red),

    % Pre-calculate highest shapes for each Green bar
    findall(green_info(Col, V, Cells),
            (member(Col, GreenBars),
             shapes_above(Col, Shapes, Ss),
             Ss \= [],
             highest_shape(Ss, component(V, Cells))),
            GreenInfos),

    % Transform the grid cell by cell
    bk:map_grid_cells(Input, transform(BG, Shapes, RedBars, GreenInfos, N_Red),
    Output).

% Find all shapes that have at least one pixel in a specific column
shapes_above(C, Shapes, Ss) :-
    findall(component(V, Cells),
            (member(component(V, Cells), Shapes), member((_, C), Cells)),
            Ss).

% Find the component with the minimum row coordinate (the highest one)
highest_shape([S|Ss], Highest) :-
    highest_shape_helper(Ss, S, Highest).

highest_shape_helper([], H, H).
highest_shape_helper([S|Ss], CurrentH, Highest) :-
    S = component(_, Cells),
    CurrentH = component(_, HCells),
    min_row(Cells, R1),
    min_row(HCells, R2),
    (R1 < R2 -> NewH = S ; NewH = CurrentH),
    highest_shape_helper(Ss, NewH, Highest).

% Helper to get the minimum row in a list of cells
min_row(Cells, MinR) :-
    findall(R, member((R, _), Cells), Rs),
    min_list(Rs, MinR).

% Transformation logic for each cell
transform(BG, Shapes, RedBars, GreenInfos, N_Red, R, C, OldV, NewV) :-
    (   R =:= 30
    ->  NewV = BG
    ;   % Rule 1: Change color of shapes above Red bars to 5 (Grey)
        member(RedCol, RedBars),
        member(component(_V, Cells), Shapes),
        member((_, RedCol), Cells),
        member((R, C), Cells)
    ->  NewV = 5
    ;   % Rule 2: Extend highest shape above Green bars by N_Red copies
        member(green_info(_GreenCol, V, Cells), GreenInfos),
        member(I, [1, 2, 3, 4, 5, 6, 7]),
        I =< N_Red,
```

```
        Offset is 4 * I,
        R_orig is R + Offset,
        member((R_orig, C), Cells)
    ->  NewV = V
    ;   % Default: keep original value
        NewV = OldV
    ).
```

Code of Iteration 2:

```
:- use_module(bk).

% Main solve predicate
solve(Input, Output) :-
    bk:grid_dimensions(Input, Rows, Cols),
    bk:grid_cell(Input, 1, 1, BG),
    bk:connected_components(Input, any, AllComps),

    % Shapes are components not of the background color and not part of the
    control row
    findall(component(V, Cells),
            (member(component(V, Cells), AllComps),
             V \= BG,
             forall(member((R, _), Cells), R < Rows)),
            Shapes),

    % Find centers of Red (2) and Green (3) bars at the bottom row
    findall(C, (member(C, [6, 12, 18, 24]), bk:grid_cell(Input, Rows, C, 2)),
    RedBarCenters),
    findall(C, (member(C, [6, 12, 18, 24]), bk:grid_cell(Input, Rows, C, 3)),
    GreenBarCenters),

    % Identify shapes above Red bars and count them
    findall(S, (member(C, RedBarCenters), member(S, Shapes), S = component(_,
    Cells), member((_, C), Cells)), RedShapesList),
    sort(RedShapesList, UniqueRedShapes),
    length(UniqueRedShapes, N_Red),

    % Identify the highest shape above each Green bar
    findall(Highest,
            (member(GC, GreenBarCenters),
             findall(S, (member(S, Shapes), S = component(_, Cells), member((_,
    GC), Cells)), Ss),
             Ss \= [],
             highest_shape(Ss, Highest)),
            GreenHighestShapes),

    % Pre-calculate all new cells created by the Green bar rule
    findall((R_copy, C, V),
            (member(component(V, Cells), GreenHighestShapes),
             member(I, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]),
             I =< N_Red,
             Offset is 4 * I,
             member((R, C), Cells),
             R_copy is R - Offset,
             R_copy >= 1),
            CopyCells),

    % Map the final grid
    bk:map_grid_cells(Input, transform(BG, Rows, UniqueRedShapes, CopyCells),
    Output).
```

```
% Helper to find the highest component (minimum row)
highest_shape([S|Ss], Highest) :-
    highest_shape_helper(Ss, S, Highest).

highest_shape_helper([], H, H).
highest_shape_helper([S|Ss], CurrentH, Highest) :-
    S = component(_, Cells),
    CurrentH = component(_, HCells),
    min_row(Cells, R1),
    min_row(HCells, R2),
    (R1 < R2 -> NewH = S ; NewH = CurrentH),
    highest_shape_helper(Ss, NewH, Highest).

min_row(Cells, MinR) :-
    findall(R, member((R, _), Cells), Rs),
    min_list(Rs, MinR).

% Transformation logic for each cell
transform(BG, LastRow, RedShapes, CopyCells, R, C, OldV, NewV) :-
    (   R =:= LastRow
    ->  NewV = BG
    ;   member((R, C, V), CopyCells)
    ->  NewV = V
    ;   member(component(_, Cells), RedShapes), member((R, C), Cells)
    ->  NewV = 5
    ;   NewV = OldV
    ).
```

Code of Iteration 3:

```
:- use_module(bk).

% Main solve predicate
solve(Input, Output) :-
    bk:grid_cell(Input, 1, 1, BG),
    bk:grid_dimensions(Input, Rows, Cols),
    bk:connected_components(Input, any, AllComps),

    % Shapes are components not of the background color and not part of the
    control row
    findall(component(V, Cells),
            (member(component(V, Cells), AllComps),
             V \= BG,
             \+ (member((R, _), Cells), R =:= Rows)),
            Shapes),

    % Find centers of Red (2) and Green (3) bars at the bottom row
    findall(C, (member(C, [6, 12, 18, 24]), C =< Cols, bk:grid_cell(Input, Rows,
     C, 2)), RedBarCenters),
    findall(C, (member(C, [6, 12, 18, 24]), C =< Cols, bk:grid_cell(Input, Rows,
     C, 3)), GreenBarCenters),

    % Identify shapes above Red bars and count them (N_Red)
    findall(S, (member(C, RedBarCenters), member(S, Shapes), S = component(_,
    Cells), member((_, C), Cells)), RS),
    length(RS, N_Red),

    % Collect all pixels of Red shapes to be recolored to 5
    findall((R, C), (member(component(_, Cells), RS), member((R, C), Cells)),
    RC_raw),
    sort(RC_raw, RC),
```

```
    % Identify the highest shape above each Green bar
    findall(H, (member(GC_Col, GreenBarCenters),
                findall(S, (member(S, Shapes), S = component(_, Cells), member((
    _, GC_Col), Cells)), GS),
                GS \= [],
                highest_shape(GS, H)),
            GHS),

    % Pre-calculate all new cells created by the Green bar replication rule
    findall((Rnew, C, V), (member(component(V, Cells), GHS),
                           member(I, [1,2,3,4,5,6,7,8,9,10]), I =< N_Red,
                           Offset is 4*I, member((R, C), Cells),
                           Rnew is R - Offset, Rnew > 0),
            GC_raw),
    sort(GC_raw, GC),

    % Map the final grid using the pre-calculated transformation data
    bk:map_grid_cells(Input, transform(BG, Rows, GC, RC), Output).

% Helper to find the highest component (minimum row) in a list of components
highest_shape([S|Ss], Highest) :-
    min_row(S, MinR),
    highest_shape_helper(Ss, S, MinR, Highest).

highest_shape_helper([], S, _, S).
highest_shape_helper([S|Ss], BestS, BestMinR, Highest) :-
    min_row(S, MinR),
    (MinR < BestMinR ->
        highest_shape_helper(Ss, S, MinR, Highest) ;
        highest_shape_helper(Ss, BestS, BestMinR, Highest)).

% Helper to get the minimum row index from a component's cells
min_row(component(_, Cells), MinR) :-
    findall(R, member((R, _), Cells), Rs),
    min_list(Rs, MinR).

% Fast lookup for Green replication copies
find_in_gc(R, C, GC, V) :- member((R, C, V), GC), !.

% Transformation logic for each cell
transform(BG, Rows, GC, RC, R, C, OldV, NewV) :-
    (   R =:= Rows
    ->  NewV = BG
    ;   find_in_gc(R, C, GC, V_GC)
    ->  NewV = V_GC
    ;   memberchk((R, C), RC)
    ->  NewV = 5
    ;   NewV = OldV
    ).
```

Code of Iteration 4:

```
:- use_module(bk).

/**
 * The transformation pattern follows a consistent logic across all examples:
 * 1.  Identify the background color (BG) from the top-left cell.
 * 2.  Identify "control bars" in the last row (Row 30). Red bars (color 2) and
    Green bars (color 3)
 *     are contiguous segments of their respective colors.
 * 3.  Identify all "shapes" in the grid (connected components that are not the
    background
```

```
 *     and are not part of the last row).
 * 4.  A shape is "above" a bar if they share any column.
 * 5.  Calculate N_Red: the total number of distinct shapes that are above any
    Red bar.
 * 6.  Rule 1 (Greying): Any shape that is above any Red bar has its color
    changed to Grey (color 5).
 * 7.  Rule 2 (Copying): For every shape that is above any Green bar, create
    N_Red copies of that
 *     shape. Each copy is shifted vertically upwards by a period of 4 rows (4,
    8, 12, ...).
 * 8.  Rule 3 (Cleanup): The entire control row (the last row) is set to the
    background color.
 * 9.  All other shapes that are not above a Red or Green bar remain unchanged.
 */

% Main solve predicate
solve(Input, Output) :-
    bk:grid_dimensions(Input, Rows, Cols),
    bk:grid_cell(Input, 1, 1, BG),

    % Find all shapes: connected components of each color (0-9) excluding BG
    % and components that are entirely on the last row (control bars).
    findall(component(V, Cells), (
        between(0, 9, V),
        V \= BG,
        bk:connected_components(Input, V, Comps),
        member(component(V, Cells), Comps),
        % A shape must have at least one pixel NOT on the last row
        \+ forall(member((R, _), Cells), R =:= Rows)
    ), Shapes),

    % Find all Red (2) and Green (3) bars in the last row
    findall(bar(2, Cells), (
        bk:connected_components(Input, 2, Comps),
        member(component(2, Cells), Comps),
        forall(member((R, _), Cells), R =:= Rows)
    ), RedBars),

    findall(bar(3, Cells), (
        bk:connected_components(Input, 3, Comps),
        member(component(3, Cells), Comps),
        forall(member((R, _), Cells), R =:= Rows)
    ), GreenBars),

    % N_Red: the count of distinct shapes above any Red bar
    findall(S, (member(B, RedBars), member(S, Shapes), is_above(S, B)),
    RedAffected),
    sort(RedAffected, UniqueRedAffected),
    length(UniqueRedAffected, N_Red),

    % Identify pixels belonging to shapes that turn Grey
    findall((R, C), (
        member(component(_, Cells), UniqueRedAffected),
        member((R, C), Cells)
    ), GreyPixels),

    % Identify pixels and colors created by the copying rule
    findall((RNew, C, V), (
        member(B, GreenBars),
        member(S, Shapes),
        is_above(S, B),
        S = component(V, Cells),
        between(1, N_Red, I),
        Offset is 4 * I,
```

```
        member((R, C), Cells),
        RNew is R - Offset,
        RNew >= 1
    ), CopyPixels),

    % Transform the grid cell by cell
    bk:map_grid_cells(Input, transform(BG, Rows, GreyPixels, CopyPixels), Output
    ).

% A shape is "above" a bar if they share at least one column
is_above(component(_, Cells), bar(_, BarCells)) :-
    member((_, C), Cells),
    member((_, C), BarCells).

% Transformation logic for each cell (R, C)
transform(BG, LastRow, GreyPixels, CopyPixels, R, C, OldV, NewV) :-
    (   R =:= LastRow
    ->  % Rule 3: Cleanup the control row
        NewV = BG
    ;   % Rule 2: Place copies first (highest priority for newly added objects)
        member((R, C, V), CopyPixels)
    ->  NewV = V
    ;   % Rule 1: Apply Grey (5) to shapes above Red bars
        member((R, C), GreyPixels)
    ->  NewV = 5
    ;   % Default: Keep existing cell value
        NewV = OldV
    ).
```

J Prolog Background Knowledge

```
:- module(bk, [
    grid_dimensions/3,
    grid_cell/4,
    grid_in_bounds/3,
    grid_neighbors4/4,
    collect_points/4,
    connected_components/3,
    components_with_holes/4,
    component_bbox/2,
    component_hole_count/3,
    component_hole_count/4,
    apply_component_labels/4,
    map_grid_cells/3
]).

:- use_module(library(lists)).

% ------------------------------------------------------------------
% Grid primitives
% ------------------------------------------------------------------

% grid_dimensions(+Grid, -Rows, -Cols)
grid_dimensions(Grid, Rows, Cols) :-
    length(Grid, Rows),
    ( Grid = [First|_] -> length(First, Cols) ; Cols = 0 ).

% grid_cell(+Grid, +Row, +Col, -Value)
grid_cell(Grid, Row, Col, Value) :-
    nth1(Row, Grid, R),
```

```
    nth1(Col, R, Value).

% grid_in_bounds(+Grid, +Row, +Col)
grid_in_bounds(Grid, Row, Col) :-
    grid_dimensions(Grid, Rows, Cols),
    Row >= 1, Row =< Rows,
    Col >= 1, Col =< Cols.

% grid_neighbors4(+Grid, +Row, +Col, -Neighbors)
grid_neighbors4(Grid, Row, Col, Neighbors) :-
    grid_dimensions(Grid, Rows, Cols),
    neighbors4_bounds(Row, Col, Rows, Cols, Neighbors).

neighbors4_bounds(Row, Col, Rows, Cols, Neighs) :-
    findall((NR,NC),
        ( (NR is Row+1, NR =< Rows, NC = Col)
        ; (NR is Row-1, NR >= 1, NC = Col)
        ; (NC is Col+1, NC =< Cols, NR = Row)
        ; (NC is Col-1, NC >= 1, NR = Row)
        ),
        Neighs).

% ------------------------------------------------------------------
% Point collection and matching
% ------------------------------------------------------------------

% collect_points(+Grid, +Matcher, -Points, -Values?)
collect_points(Grid, Matcher, Points, Values) :-
    grid_dimensions(Grid, Rows, Cols),
    findall((Coord, Val),
        ( between(1, Rows, R),
          between(1, Cols, C),
          nth1(R, Grid, Row),
          nth1(C, Row, Val),
          match_value(Matcher, Val),
          Coord = (R,C)
        ),
        Raw),
    unzip_points(Raw, Points, Values).

collect_points(Grid, Matcher, Points) :-
    collect_points(Grid, Matcher, Points, _).

unzip_points([], [], []).
unzip_points([((R,C),V)|Rest], [(R,C)|PR], [V|VR]) :-
    unzip_points(Rest, PR, VR).

match_value(any, _) :- !.
match_value(nonzero, Val) :- Val =\= 0, !.
match_value(nonzero, _) :- !, fail.
match_value(color(N), Val) :-
    !,
    number(N),
    Val =:= N.
match_value(Matcher, Val) :-
    callable(Matcher),
    call(Matcher, Val), !.
match_value(Matcher, Val) :-
    is_list(Matcher),
    memberchk(Val, Matcher), !.
match_value(Matcher, Val) :-
    number(Matcher),
    Val =:= Matcher.
```

```
% ------------------------------------------------------------------
% Connected components
% ------------------------------------------------------------------

% connected_components(+Grid, +Matcher, -Components)
% Components are terms component(Value, Cells)
connected_components(Grid, Matcher, Components) :-
    grid_dimensions(Grid, Rows, Cols),
    collect_points(Grid, Matcher, Points),
    components_from_points(Points, Grid, Rows, Cols, Matcher, [], Components).

components_from_points([], _, _, _, _, _, []).
components_from_points([(R,C)|Rest], Grid, Rows, Cols, Matcher, Seen, Components
    ) :-
    ( memberchk((R,C), Seen) ->
        components_from_points(Rest, Grid, Rows, Cols, Matcher, Seen, Components
    )
    ;
        grid_cell(Grid, R, C, Val),
        flood_component(Grid, Rows, Cols, Matcher, [(R,C)], [], Cells),
        append(Cells, Seen, NewSeen),
        Components = [component(Val, Cells)|Tail],
        components_from_points(Rest, Grid, Rows, Cols, Matcher, NewSeen, Tail)
    ).

flood_component(_, _, _, _, [], Visited, Visited).
flood_component(Grid, Rows, Cols, Matcher, [(R,C)|Queue], Visited, Component) :-
    ( memberchk((R,C), Visited) ->
        flood_component(Grid, Rows, Cols, Matcher, Queue, Visited, Component)
    ;
        cell_value(Grid, Rows, Cols, R, C, Val),
        match_value(Matcher, Val) ->
            neighbors4_bounds(R, C, Rows, Cols, Neighs),
            append(Queue, Neighs, NextQueue),
            flood_component(Grid, Rows, Cols, Matcher, NextQueue, [(R,C)|Visited
    ], Component)
    ;
        flood_component(Grid, Rows, Cols, Matcher, Queue, Visited, Component)
    ).

% ------------------------------------------------------------------
% Component utilities
% ------------------------------------------------------------------

component_bbox(Cells, bbox(MinR, MaxR, MinC, MaxC)) :-
    findall(R, member((R,_), Cells), Rs),
    min_list(Rs, MinR),
    max_list(Rs, MaxR),
    findall(C, member((_,C), Cells), Cs),
    min_list(Cs, MinC),
    max_list(Cs, MaxC).

components_with_holes(Grid, Matcher, BackgroundMatcher, Components) :-
    connected_components(Grid, Matcher, Raw),
    annotate_holes(Grid, BackgroundMatcher, Raw, Components).

annotate_holes(_, _, [], []).
annotate_holes(Grid, BackgroundMatcher, [component(Val, Cells)|Rest], [component
    (Val, Cells, holes(H))|Tail]) :-
    component_hole_count(Grid, Cells, BackgroundMatcher, H),
    annotate_holes(Grid, BackgroundMatcher, Rest, Tail).

% Example: Hole counting
```

```
component_hole_count(Grid, Cells, Count) :-
    component_hole_count(Grid, Cells, 0, Count).

component_hole_count(Grid, Cells, BackgroundMatcher, Count) :-
    grid_dimensions(Grid, Rows, Cols),
    component_bbox(Cells, bbox(MinR, MaxR, MinC, MaxC)),
    PMinR is MinR - 1,
    PMaxR is MaxR + 1,
    PMinC is MinC - 1,
    PMaxC is MaxC + 1,
    Limits = limits(PMinR, PMaxR, PMinC, PMaxC),
    list_to_ord_set(Cells, ComponentSet),
    border_background_cells(Grid, Rows, Cols, Limits, ComponentSet,
    BackgroundMatcher, BorderStarts),
    flood_background(Grid, Rows, Cols, Limits, ComponentSet, BackgroundMatcher,
    BorderStarts, ReachableRaw),
    list_to_ord_set(ReachableRaw, Reachable),
    all_background_in_limits(Grid, Rows, Cols, Limits, ComponentSet,
    BackgroundMatcher, BackgroundRaw),
    list_to_ord_set(BackgroundRaw, Background),
    ord_subtract(Background, Reachable, HoleSet),
    count_hole_components(HoleSet, Limits, Count).

border_background_cells(_, _, _, limits(MinR, MaxR, MinC, MaxC), _, _, Starts)
    :-
    MinR > MaxR ; MinC > MaxC, !,
    Starts = [].
border_background_cells(Grid, Rows, Cols, limits(MinR, MaxR, MinC, MaxC),
    ComponentSet, BackgroundMatcher, Starts) :-
    findall((R,C),
        ( between(MinC, MaxC, C), (R = MinR ; R = MaxR)
        ; between(MinR, MaxR, R), (C = MinC ; C = MaxC)
        ),
        RawBorder),
    sort(RawBorder, BorderCoords),
    filter_background_coords(BorderCoords, Grid, Rows, Cols, ComponentSet,
    BackgroundMatcher, Starts).

all_background_in_limits(Grid, Rows, Cols, limits(MinR, MaxR, MinC, MaxC),
    ComponentSet, BackgroundMatcher, Background) :-
    findall((R,C),
        ( between(MinR, MaxR, R),
          between(MinC, MaxC, C),
          background_cell(Grid, Rows, Cols, ComponentSet, BackgroundMatcher, R,
    C)
        ),
        Background).

background_cell(Grid, Rows, Cols, ComponentSet, BackgroundMatcher, R, C) :-
    \+ ord_memberchk((R,C), ComponentSet),
    cell_value(Grid, Rows, Cols, R, C, Val),
    match_value(BackgroundMatcher, Val).

flood_background(_, _, _, _, _, _, [], []).
flood_background(Grid, Rows, Cols, Limits, ComponentSet, BackgroundMatcher,
    Starts, Reachable) :-
    flood_background_queue(Grid, Rows, Cols, Limits, ComponentSet,
    BackgroundMatcher, Starts, [], Visit),
    reverse(Visit, Reachable).

flood_background_queue(_, _, _, _, _, _, [], Vis, Vis).
flood_background_queue(Grid, Rows, Cols, Limits, ComponentSet, BackgroundMatcher
    , [(R,C)|Queue], Vis, Result) :-
    ( memberchk((R,C), Vis) ->
```

```
        flood_background_queue(Grid, Rows, Cols, Limits, ComponentSet,
    BackgroundMatcher, Queue, Vis, Result)
    ; background_cell(Grid, Rows, Cols, ComponentSet, BackgroundMatcher, R, C)
    ->
        neighbors_within_limits((R,C), Limits, Neighs),
        filter_background_coords(Neighs, Grid, Rows, Cols, ComponentSet,
    BackgroundMatcher, Valid),
        append(Queue, Valid, NextQueue),
        flood_background_queue(Grid, Rows, Cols, Limits, ComponentSet,
    BackgroundMatcher, NextQueue, [(R,C)|Vis], Result)
    ;
        flood_background_queue(Grid, Rows, Cols, Limits, ComponentSet,
    BackgroundMatcher, Queue, Vis, Result)
    ).

neighbors_within_limits((R,C), limits(MinR, MaxR, MinC, MaxC), Neighs) :-
    findall((NR,NC),
        ( (NR is R+1, between(MinR, MaxR, NR), NC = C)
        ; (NR is R-1, between(MinR, MaxR, NR), NC = C)
        ; (NC is C+1, between(MinC, MaxC, NC), NR = R)
        ; (NC is C-1, between(MinC, MaxC, NC), NR = R)
        ),
        Neighs).

count_hole_components([], _, 0).
count_hole_components(HoleSet, Limits, Count) :-
    HoleSet \= [],
    count_hole_components_loop(HoleSet, Limits, 0, Count).

count_hole_components_loop([], _, Acc, Acc).
count_hole_components_loop(HoleSet, Limits, Acc, Count) :-
    HoleSet = [Cell|_],
    hole_component(Cell, HoleSet, Limits, Component),
    ord_subtract(HoleSet, Component, Remaining),
    Acc1 is Acc + 1,
    count_hole_components_loop(Remaining, Limits, Acc1, Count).

hole_component(Cell, HoleSet, Limits, ComponentOrd) :-
    flood_subset([Cell], [], HoleSet, Limits, Component),
    list_to_ord_set(Component, ComponentOrd).

flood_subset([], Vis, _, _, Vis).
flood_subset([(R,C)|Queue], Vis, HoleSet, Limits, Component) :-
    ( memberchk((R,C), Vis) ->
        flood_subset(Queue, Vis, HoleSet, Limits, Component)
    ; ord_memberchk((R,C), HoleSet) ->
        neighbors_within_limits((R,C), Limits, Neighs),
        filter_ord_members(Neighs, HoleSet, Valid),
        append(Queue, Valid, NextQueue),
        flood_subset(NextQueue, [(R,C)|Vis], HoleSet, Limits, Component)
    ;
        flood_subset(Queue, Vis, HoleSet, Limits, Component)
    ).

filter_background_coords([], _, _, _, _, _, []).
filter_background_coords([(R,C)|Rest], Grid, Rows, Cols, ComponentSet,
    BackgroundMatcher, Filtered) :-
    ( background_cell(Grid, Rows, Cols, ComponentSet, BackgroundMatcher, R, C)
    ->
        Filtered = [(R,C)|Tail]
    ;
        Filtered = Tail
    ),
```

```
    filter_background_coords(Rest, Grid, Rows, Cols, ComponentSet,
    BackgroundMatcher, Tail).

filter_ord_members([], _, []).
filter_ord_members([(R,C)|Rest], HoleSet, Filtered) :-
    ( ord_memberchk((R,C), HoleSet) ->
        Filtered = [(R,C)|Tail]
    ;
        Filtered = Tail
    ),
    filter_ord_members(Rest, HoleSet, Tail).

cell_value(_, Rows, _, R, _, 0) :-
    (R < 1 ; R > Rows), !.
cell_value(_, _, Cols, _, C, 0) :-
    (C < 1 ; C > Cols), !.
cell_value(Grid, _, _, R, C, Val) :-
    nth1(R, Grid, Row),
    nth1(C, Row, Val), !.
cell_value(_, _, _, _, _, 0).

% ---------------------------------------------------------------------
% Component labeling helpers
% ---------------------------------------------------------------------

apply_component_labels(Grid, Components, Labeler, Result) :-
    findall(((R,C)-Label),
        ( member(Component, Components),
          call(Labeler, Component, Label),
          component_cells(Component, Cells),
          member((R,C), Cells)
        ),
        Pairs),
    map_grid(Grid, Pairs, Result).

component_cells(component(_, Cells), Cells).
component_cells(component(_, Cells, _), Cells).

map_grid(Grid, Pairs, Result) :-
    map_rows(Grid, Pairs, 1, Result).

map_rows([], _, _, []).
map_rows([Row|Rest], Pairs, RowIdx, [Mapped|Tail]) :-
    map_row(Row, Pairs, RowIdx, 1, Mapped),
    NextIdx is RowIdx + 1,
    map_rows(Rest, Pairs, NextIdx, Tail).

map_row([], _, _, _, []).
map_row([Val|Rest], Pairs, R, C, [NewVal|Tail]) :-
    ( memberchk(((R,C)-Label), Pairs) ->
        NewVal = Label
    ;
        NewVal = Val
    ),
    NextC is C + 1,
    map_row(Rest, Pairs, R, NextC, Tail).

% ---------------------------------------------------------------------
% Ordered-set helpers
% ---------------------------------------------------------------------

list_to_ord_set(List, OrdSet) :-
    sort(List, OrdSet).
```

```
ord_subtract([], _, []).
ord_subtract([A|As], Bs, Cs) :-
    ( ord_memberchk(A, Bs) ->
        ord_subtract(As, Bs, Cs)
    ;
        Cs = [A|Rest],
        ord_subtract(As, Bs, Rest)
    ).

ord_memberchk(_, []) :- fail.
ord_memberchk(X, [Y|Ys]) :-
    ( X == Y ->
        true
    ; X @> Y ->
        ord_memberchk(X, Ys)
    ; X @< Y ->
        fail
    ).

% ---------------------------------------------------------------------
% Grid cell mapping
% ---------------------------------------------------------------------

% map_grid_cells(+Grid, +Mapper, -NewGrid)
% Apply a cell-level transformation to every cell in the grid.
% Mapper is called as: call(Mapper, Row, Col, OldValue, NewValue)
% Example usage:
%   map_grid_cells(Input, my_transform, Output).
%   my_transform(R, C, OldVal, NewVal) :-
%       ( in_target_region(R, C), OldVal =:= 0 -> NewVal = 4 ; NewVal = OldVal )
    .

map_grid_cells(Grid, Mapper, NewGrid) :-
    map_grid_rows(Grid, Mapper, 1, NewGrid).

map_grid_rows([], _, _, []).
map_grid_rows([Row|RestRows], Mapper, RowIdx, [NewRow|RestNewRows]) :-
    map_grid_cols(Row, Mapper, RowIdx, 1, NewRow),
    NextRow is RowIdx + 1,
    map_grid_rows(RestRows, Mapper, NextRow, RestNewRows).

map_grid_cols([], _, _, _, []).
map_grid_cols([Val|RestVals], Mapper, RowIdx, ColIdx, [NewVal|RestNewVals]) :-
    call(Mapper, RowIdx, ColIdx, Val, NewVal),
    NextCol is ColIdx + 1,
    map_grid_cols(RestVals, Mapper, RowIdx, NextCol, RestNewVals).
```

K Prompts

System Prompt

```
You are a world-class expert in solving Abstract Reasoning Corpus (ARC) tasks.
Your approach is methodical, creative, and highly effective.  You are also a
master Prolog programmer, producing elegant, efficient, and well-documented
solutions.

**IMPORTANT: Your Ultimate Goal**

You will be given:
1.  **Training Examples** - Input-output pairs to help you understand the
transformation pattern
```

```
2.  **Challenge(s)** - New input(s) for which you must produce correct output(s)

The Training Examples are for learning the pattern.  Your REAL GOAL is to
produce a `solve/2` predicate that correctly transforms the Challenge input(s).
The training examples help you understand what transformation to apply, but
success is measured by whether your code works on the Challenge(s).

**Part 1:  Initial Analysis and Hypothesis Generation**

1.  **Example Inspection:** Carefully examine the input and output grids for
each training example.  Note their dimensions, color palettes, and any prominent
visual features (shapes, symmetries, patterns).

2.  **Challenge Awareness:** Look at the Challenge input(s).  Consider how they
relate to the training examples - are they similar in structure?  Do they have
edge cases not seen in training?

3.  **Formulate a Hypothesis:**
* Based on your analysis, formulate a transformation rule that works
consistently across all examples AND generalizes to the challenge.
* Express the rule as a sequence of image manipulation operations.
* Prioritize simpler, more general rules that avoid overfitting to training
examples.
* Consider these types of transformations:
* **Object Manipulation:** Moving, rotating, reflecting, or resizing objects.
* **Color Changes:** Changing the color of specific objects or regions.
* **Spatial Arrangements:** Rearranging the objects in a specific pattern.
* **Object Addition/Removal:** Adding or removing objects based on certain
criteria.

**Part 2:  bk.pl Library API**

You have access to a powerful Prolog library called `bk.pl` for grid
manipulation.  Use it!
**Available predicates from bk.pl module:**

You MUST use `:- use_module(bk).` at the beginning of your code to access these
predicates.
Call them with `bk:predicate_name(...)` syntax.

**CRITICAL: All coordinates are 1-indexed!  (Row=1, Col=1) is the top-left
cell.**

------------------------------
GRID PRIMITIVES
------------------------------

- bk:grid_dimensions(+Grid, -Rows, -Cols)
Get grid dimensions.
Example:  bk:grid_dimensions(Input, H, W)

- bk:grid_cell(+Grid, +Row, +Col, -Value)
Get cell value at position (1-indexed!).
Example:  bk:grid_cell(Grid, 1, 1, TopLeftVal)

- bk:grid_in_bounds(+Grid, +Row, +Col)
Check if position is within grid bounds.

- bk:grid_neighbors4(+Grid, +Row, +Col, -Neighbors)
Get 4-connected neighbors as list of (R,C) tuples.

------------------------------
GRID TRANSFORMATION (RECOMMENDED for cell-by-cell operations)
```

````
------------------------------

- bk:map_grid_cells(+Grid, +Mapper, -NewGrid)
Transform every cell in the grid.
Mapper is called as:  call(Mapper, Row, Col, OldValue, NewValue)

**IMPORTANT: Mapper must have 4 parameters (Row, Col, OldVal, NewVal)**

Example:
"‘prolog
solve(Input, Output) :-
bk:map_grid_cells(Input, my_transform, Output).

my_transform(R, C, OldVal, NewVal) :-
( OldVal =:= 0 -> NewVal = 5 ; NewVal = OldVal ).
"‘

------------------------------
COMPONENT DISCOVERY
------------------------------

- bk:connected_components(+Grid, +Matcher, -Components)
Find all connected components matching the criteria.

**Matcher options:**
- Plain number:  2, 8, 0 (matches cells with that exact value)
- color(N): same as plain number
- nonzero:  matches any non-zero cell
- any:  matches all cells
- List [1,2,3]:  matches any value in the list

**NEVER use =:=(N) or =(N) as matcher!**

Returns:  list of component(Value, Cells) where Cells is list of (Row,Col)

Example:
"‘prolog
bk:connected_components(Grid, 8, Comps),
member(component(8, Cells), Comps)
"‘

- bk:components_with_holes(+Grid, +Matcher, +BackgroundMatcher, -Components)
Like connected_components but also counts holes in each component.
Returns:  list of component(Value, Cells, holes(Count))

Example:
"‘prolog
bk:components_with_holes(Input, 2, 0, Comps),
member(component(2, Cells, holes(HoleCount)), Comps)
"‘

------------------------------
COMPONENT UTILITIES
------------------------------

- bk:component_bbox(+Cells, -BBox)
Get bounding box of a component.

**IMPORTANT: BBox is bbox(MinRow, MaxRow, MinCol, MaxCol) - a compound term, NOT
a list!**

Example:
"‘prolog
````

```
bk:component_bbox(Cells, bbox(MinR, MaxR, MinC, MaxC)),
Height is MaxR - MinR + 1,
Width is MaxC - MinC + 1
"‘

- bk:apply_component_labels(+Grid, +Components, +Labeler, -Result)
Relabel components according to a labeling function.

**Labeler signature:  Labeler(Component, NewLabel)**
Component is component(Value, Cells) or component(Value, Cells, holes(H))

Example:
"‘prolog
solve(Input, Output) :-
bk:components_with_holes(Input, 8, 0, Comps),
bk:apply_component_labels(Input, Comps, label_by_holes, Output).

label_by_holes(component(_Val, _Cells, holes(H)), NewLabel) :-
hole_to_color(H, NewLabel).

hole_to_color(1, 5).
hole_to_color(2, 3).
"‘

------------------------------
COMPLETE WORKING EXAMPLES
------------------------------

**Example 1:  Relabel components by hole count**
"‘prolog
:- use_module(bk).

solve(Input, Output) :-
bk:components_with_holes(Input, 8, 0, Comps),
bk:apply_component_labels(Input, Comps, label_by_holes, Output).

label_by_holes(component(_Val, _Cells, holes(H)), NewLabel) :-
hole_to_label(H, NewLabel).

hole_to_label(1, 5).
hole_to_label(2, 3).
hole_to_label(3, 7).
"‘

**Example 2:  Fill regions based on component size**
"‘prolog
:- use_module(bk).

solve(Input, Output) :-
bk:connected_components(Input, 2, Frames),
findall(region(MinR,MaxR,MinC,MaxC,Color),
(member(component(2,Cells), Frames),
bk:component_bbox(Cells, bbox(MinR,MaxR,MinC,MaxC)),
Height is MaxR - MinR + 1,
size_to_color(Height, Color)),
Regions),
bk:map_grid_cells(Input, fill_regions(Regions), Output).

fill_regions(Regions, R, C, OldVal, NewVal) :-
( OldVal =:= 0,
member(region(MinR,MaxR,MinC,MaxC,Color), Regions),
R > MinR, R < MaxR, C > MinC, C < MaxC
-> NewVal = Color
```

````
; NewVal = OldVal
).

size_to_color(5, 8).
size_to_color(7, 4).
"‘

**Example 3:  Simple pixel-wise transformation**
"‘prolog
:- use_module(bk).

solve(Input, Output) :-
bk:map_grid_cells(Input, recolor_cell, Output).

recolor_cell(_R, _C, OldVal, NewVal) :-
( OldVal =:= 8 -> NewVal = 0 ; NewVal = OldVal ).
"‘

------------------------------
COMMON MISTAKES TO AVOID
------------------------------

WRONG: bk:connected_components(Grid, =:=(N), ...)
RIGHT: bk:connected_components(Grid, N, ...)

WRONG: bk:component_bbox(Cells, [MinR,MinC,MaxR,MaxC])
RIGHT: bk:component_bbox(Cells, bbox(MinR,MaxR,MinC,MaxC))

WRONG: my_mapper(OldVal, NewVal) :- ...  (missing Row, Col)
RIGHT: my_mapper(Row, Col, OldVal, NewVal) :- ...

WRONG: Using 0-indexed coordinates
RIGHT: All coordinates are 1-indexed (top-left is Row=1, Col=1)

**Part 3:  Output Requirements**

1.  **Output Format:**
* Begin with a concise paragraph explaining the proposed solution.
* You *must* provide code representing your best attempt.  Do not give up or
refuse to produce code.
* **The code section must be a single, valid Prolog code block in markdown
fenced code block format.**
* **Start with ‘:- use_module(bk).‘**
* The main predicate must have the signature ‘solve(Input, Output)‘.
* Include all helper predicates - the code must be self-contained except for
bk.pl.

* **Ensure your solution generalizes** - avoid hardcoding values specific to
training examples.
````

First Generation Prompt

````
**PROBLEM:**

Below are the Training Examples (with input-output pairs) and Challenge(s)
(input only, you must figure out the output).

{examples}

**Your Task:**
1.  Study the Training Examples to understand the transformation pattern
2.  Write a Prolog ‘solve/2‘ predicate that correctly implements this
````

```
transformation
3.  Your code will be tested on the Training Examples for debugging, and the
FINAL goal is to correctly transform the Challenge input(s)

Remember to use ‘:- use_module(bk).‘ and leverage the bk.pl library predicates
when appropriate.
```

Fix Prompt

```
**CODE REFINEMENT REQUIRED**
Attempt {current_iteration}/{max_iterations}

Your code has gone through {iteration_count} iteration(s) of testing.  Please
carefully analyze:
1.  {history_description}
2.  The detailed execution trace of the most recent attempt
3.  Identify the root cause of failures - is it a logic error, edge case
handling, or incorrect pattern matching?

Based on this analysis, produce an improved solution that addresses all
identified issues.

**REFERENCE SOLUTIONS ({history_type}):**

{attempts_history}

**DECLARATIVE DEBUGGER OUTPUT (Iteration {iteration}):**

The following is the output from the declarative debugger showing:
1.  **INPUT/EXPECTED/ACTUAL grids** - visual comparison of what was expected vs
produced
2.  **DIFFERENCE SUMMARY** - specific cells that differ
3.  **PROOF TREE** - the successful execution path showing how your code
transformed the data

Before you produce your solution, study the proof tree carefully by applying
**algorithmic program debugging** rules to identify the buggy predicates:
1.  Start from the root node and check if its output is correct given its input.
2.  If a node’s output is CORRECT -> skip its entire subtree (no bug there).
3.  If a node’s output is INCORRECT -> examine its children nodes.
4.  A node is the **bug location** when:  its own output is incorrect, BUT all
its children outputs are correct.

"‘
{trace_detail}
"‘

**CHALLENGE INPUT(S) - Your code must correctly transform these:**

{challenge_diagrams}

Remember:  The training examples are for learning the pattern, but the REAL goal
is to produce correct outputs for the Challenge input(s) above.

Now provide a corrected, complete Prolog program.  Make sure to:
1.  Address all issues identified in the trace
2.  Handle edge cases properly
3.  **CRITICAL: Ensure the solution GENERALIZES to the Challenge input(s)** -
don’t overfit to training examples
4.  Use bk.pl library predicates correctly (remember:  1-indexed coordinates,
bbox is a compound term)
5.  Avoid hardcoding specific values that only work for training examples
```

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction describe semantic-trace-guided refinement on ARC-AGI-2 and supplementary symbolic RAVEN adaptations. Cross-language and cross-protocol comparisons are qualified.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The limitations paragraph discusses fallible diagnoses, weak initializers, proprietary model dependence, additional token cost, public-data exposure, and the limited evaluation scope.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: The paper defines a refinement framework and gives an empirical algorithm. It does not present a theorem establishing optimality, diagnosis correctness, or generalization.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The paper and supplement describe the algorithm, benchmarks, model settings, sampling temperature, search breadth and depth, selection and scoring rules, prompts, feedback conditions, and symbolic RAVEN adaptations. The supplement provides the core implementation and examples. Reproduction requires access to the specified model services.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No]

   Justification: The OpenReview supplement contains ARC implementation code and two examples, plus RAVEN runners, generators, sample data, and Prolog result summaries. It does not contain complete runners, configurations, and results for every ARC and Python control.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: The paper describes the public evaluation split, model backbones, sampling temperature, reasoning effort, search breadth and depth, candidate selection, feedback conditions, and symbolic RAVEN protocol. Python checkpoint generation and feedback are reported as separate conditions.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: The appendix defines and reports repeated-run means, sample standard deviations, and ranges for selected ARC settings, with the repeat counts and sources of stochastic variation specified. These describe run-to-run variability; no paired significance claim is made for the added feedback controls.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [No]

   Justification: The appendix reports Gemini-3-Pro request counts, input and output tokens, a fixed cost basis, and search settings. Execution time, local machine resources, and complete compute totals across all experiments are not available in this paper.

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The reported research concerns abstract reasoning benchmarks and model-based program refinement, with no human-subject experiments or private personal data.

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: The Broader Impacts appendix discusses failure inspection, incorrect semantic diagnoses, harmful code-generation uses, execution isolation, and application-specific validation.

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper does not release a new pretrained model or scraped dataset with a high risk for misuse.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [No]

    Justification: The paper credits benchmark and software sources. It does not enumerate the version-specific licenses and service terms for every asset.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The OpenReview supplement documents the implementation and symbolic RAVEN adaptation, including runtime configuration, generators, sample data, example outputs, and Prolog summaries. The paper explains the adaptation and evaluation scope.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A]

Justification: The research does not involve crowdsourcing or human-subject studies.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: No human-subject experiments are conducted.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: Large language models are central to hypothesis generation, semantic diagnosis, and program repair. Their roles, evaluated configurations, and prompts are described.