Compositional Abstractions for ARC-Style Tasks

1 Abstract

We study how **compositional abstractions** shrink program-search for ARC-style grid puzzles by removing symmetries *before* search. We represent an abstraction as **(G + invariant)**: a concrete grid transformation G together with an **invariant** that constrains which concrete worlds we consider equivalent. A task is solvable in an abstracted space if: (i) its invariants hold; and (ii) a **mapping exists** in the abstract space that transfers back to the concrete space via a simple gauge (bookkeeping) map.

Two tiny abstractions suffice to make otherwise messy tasks trivial:

- **A1: Palette canonicalization.** Relabel non-zero colors by decreasing frequency. This quotients out palette symmetry so rules like "least-frequent color" have a canonical id.
- **A2: Canonical object order.** Sort connected components by $(area, top, left, color)$. This quotients out object-enumeration symmetry so rules like "component index 0" are stable—even in ties.

Empirically, composing **A1→A2** collapses the search space from thousands of programs to **two**; both are valid for the studied case, giving **near-zero search cost**.

2 Introduction

ARC tasks operate on small integer grids. Many puzzles involve selecting an object (a connected component), choosing a (target) color, and recoloring. Naively, a solver faces huge **spurious multiplicity**: many grids differ only by a permutation of color ids or by the order we enumerate objects, yet a human treats them as "the same."

Our goal is to **quotient symmetries away** with minimal machinery, leaving a tiny set of candidate programs that are all semantically distinct. We make three design choices:

- **Abstractions as $G+$ invariant.** Each abstraction applies a concrete normalization G (e.g., renaming colors; sorting objects) and comes with an invariant that must hold after normalization.
- **Mapping existence before search.** We first check that a simple family of mappings exists in the abstract space; only then do we enumerate candidate programs.
- **Microscopic DSLs.** We intentionally use tiny search spaces to show how much the **symmetry quotient** alone reduces the burden.

3 Running Examples (3 pairs)

We use small grids where the target action can be stated compactly: **select a component** $\times$ **choose a color** $\times$ **recolor the component** (e.g., "recolor the smallest component to the least-frequent non-zero color"). Examples include ties (equal-area components) and multi-color scenes so that without a canonical order, phrases like "the smallest" are ambiguous.

4 Abstractions as G + Invariant

We spell out the two building blocks and how they eliminate symmetry.

4.1 A1: Palette canonicalization

α_1: relabel non-zero colors by **decreasing frequency**, breaking ties by the **smaller original color id**. Background `0` is preserved. Returns `(x_hat, meta)` with `meta = {"orig_for_can": ..., "can_for_orig": ...}`.

- **Invariant.** No forbidding constraints beyond keeping background `0` intact.
- **Gauge.** A bidirectional palette map (original$\leftrightarrow$canonical) to transfer predictions back to the original ids.
- **Why.** Quotients out **palette symmetry**, making statements like "least-frequent color" well-defined in every instance.

4.2 A2: Canonical object order (on top of A1)

α_2: sort connected components by the tuple $(area, top, left, color)$ after A1.

- **Invariant.** Metadata-only; does not constrain geometry beyond the ordering.
- **Gauge.** The canonical order list to transfer "index-based" references back to concrete component ids.
- **Why.** Quotients out **object-enumeration symmetry** so that "component index 0" has a stable meaning even in tie situations.

4.3 Mapping existence (search-free)

Before enumerating programs, we check that an appropriate **mapping exists** in the abstract space. For the family we study (select-component $\times$ choose-color $\times$ recolor), existence reduces to:

1. **Selectors** can name at least one component under α_2 (e.g., `index0`, `smallest`).
2. **Color rules** can name a palette element under α_1 (e.g., `least_frequent`, `most_frequent`, or fixed ids after canon).
3. **Gauge transfer** (palette map + order list) can un-canonize the abstract action back to the original grid.

If these checks pass on all training pairs, we proceed to program enumeration; otherwise, we abstain.

5 Program Search Spaces (DSLs)

Code-coherent identifiers (from `arc_abstraction_dsl_2.py`):

- A1: `alpha1_palette(x) -> (x_hat, meta)` where `meta = {"orig_for_can": ..., "can_for_orig": ...}`.
- A2: `alpha2_objorder(x_hat) -> (x_hat, {"order": comps_sorted})` with sort key $(area, top, left, colo$
- Color rules: `COLOR_RULES = [("max_id", sel_color_max_id), ("argmin_hist", sel_color_argmin_his`
  - Under A1, **both** pick a least-frequent color; when multiple colors tie, both choose the **largest canonical id**.
- Selectors:
  - Raw/G: `sel_comp_smallest_unstable` (hash-shuffled tie-break) **and** `sel_comp_smallest_canonical`.
  - A1: many variants from `build_A1_selectors()` (different tie-break keys and seeded shuffles) **plus** `sel_comp_smallest_canonical`.
  - A1→A2: fixed `("index0", lambda a: sel_comp_smallest_canonical(a))`.
- G pre-ops: `build_preops_for_dataset(...)` produces `("identity", ...)` and hundreds of `"perm_*"` random palette permutations to explode the search space.

These names are what appear in the results and JSON artifacts the script writes.

We use three nested spaces, each reusing the same **action family** (select component × choose color × recolor) but differing in *how many spurious variants* they contain.

- **G (raw).** Enumerates many selector variants (by geometry and enumeration) and many color rules that simulate palette symmetry. This intentionally inflates the candidate count.
- **A1.** After palette canonicalization, we still allow numerous selector tie-break variants to show that removing palette symmetry alone is incomplete.
- **A1→A2.** After canonical object ordering, the DSL collapses to **two** programs: {color rule} × {index0} (implemented as "canonical smallest").

Across all spaces, the mapping family is identical; only the **symmetry multiplicity** changes.

5.1 Concrete worked examples (aligned with `arc_abstraction_dsl_2.py`)

5.1.1 Concrete example: A1 palette canonicalization

Original grid (colors: 0=background, 2,3,5):

```
0 0 2 2 0 0
0 3 3 0 0 0
0 3 0 0 5 5
0 0 0 0 5 0
```

Non-zero histogram: 3→3, 5→3, 2→2. Canonical relabel (descending frequency, ties by **smaller original id**): 3→1, 5→2, 2→3.

Canonicalized grid `x_hat` and meta:

```
1 1 1 0 0 0
1 1 0 0 2 2
0 0 0 0 0 0
0 0 0 2 2 2
0 0 0 0 0 1
```

`meta.orig_for_can = {1: 3, 2: 5, 3: 2}` `meta.can_for_orig = {3: 1, 5: 2, 2: 3}`

5.1.2 Concrete example: A2 canonical object order

Connected components in `x_hat` above, summarized as $(area, (top, left), color)$:

- (2, (0, 2), 3)
- (3, (1, 1), 1)
- (3, (2, 4), 2)

A2 orders them by $(area, top, left, color)$, so **index 0** is the $(2, (0, 2), 3)$ component.

5.1.3 End-to-end example: A1→A2 + recolor + gauge back

Input x (original palette):

```
7 7 7 0 0 0
7 7 0 0 2 2
0 0 0 0 0 0
0 0 0 3 3 3
0 0 0 0 0 7
```

After A1 canonicalization (`x_hat`):

```
1 1 1 0 0 0
1 1 0 0 3 3
0 0 0 0 0 0
0 0 0 2 2 2
0 0 0 0 0 1
```

Selector: `sel_comp_smallest_canonical(x_hat)` → the single-pixel component (area=1). Color rule: `sel_color_argmin_hist(x_hat)` → choose the **least-frequent** canonical color = 3. Apply `recolor_component(x_hat, comp, 3)`:

```
1 1 1 0 0 0
1 1 0 0 3 3
0 0 0 0 0 0
0 0 0 2 2 2
0 0 0 0 0 3
```

Gauge back to original palette using `meta.orig_for_can` (1→7, 2→3, 3→2):

```
7 7 7 0 0 0
7 7 0 0 2 2
0 0 0 0 0 0
0 0 0 3 3 3
0 0 0 0 0 2
```

6 Algorithm (sketch)

1. **Normalize inputs** with α_1 then α_2; record palette map and order list.
2. **Validate invariants** on all train pairs.
3. **Check mapping existence** in the abstract space (selectors × color rules × recolor).
4. **Enumerate candidate programs** in the chosen DSL (G, A1, or A1→A2).
5. **Evaluate** on train; keep programs consistent with all pairs.
6. **Transfer** the chosen abstract action back to each concrete grid via the gauges.

This ordering—**validation** → **existence** → **search**—prevents wasted search when symmetry removal already makes the solution unique.

7 Experimental Protocol

We evaluate a single ARC-style task family with three DSLs (G, A1, A1→A2). Metrics:

- `total_candidates` – size of the enumerated space.
- `num_valid` – programs consistent with all training pairs.
- `avg_tries_to_success` – mean number of attempts until the first valid program is found when sampling uniformly at random.
- `wall_time_s` – runtime for a deterministic sweep on our simple reference implementation.

8 Results

Local-run results (your updated run):

```
[G]       total_candidates=2404  num_valid=441  avg_tries_to_success=5.405
     wall_time_s=3.884
[A1]      total_candidates=172   num_valid=4    avg_tries_to_success=35.573
     wall_time_s=0.197
[A1->A2]  total_candidates=2     num_valid=2    avg_tries_to_success=1.000
     wall_time_s=0.028
```

These match the qualitative trend reported earlier (in a single-dataset summary) and sharpen the magnitude of the win from composing A1 with A2.

8.1 Read-off improvements from the local run

- **Program count shrinks:** G→A1: 2404→172 (−**92.85%**); A1→A1→A2: 172→2 (−**98.84%**). Overall G→A1→A2: −**99.917%** (2404→2).
- **Runtime collapses:** 3.884s→0.197s (×**19.7 faster**), then 0.197s→0.028s (×**7.0 faster**). Overall: ×**138.7 faster** (3.884s→0.028s).

- **Effort to first solution:** `avg_tries_to_success` is **1.0** in A1→A2 (immediate), **5.41** in G (moderate), and **35.57** in A1 (smaller space but **noisier** due to tie-break variants); composition with A2 removes that ambiguity.

- **Validity rates:** A1→A2: **2/2** (100%); A1: **4/172** ≈ **2.3%**; G: **441/2404** ≈ **18.3%**.

9 Interpretation

- **A1 (palette canonicalization)** quotients out color symmetry so rules like "least-frequent color" become stable; however, it leaves **object enumeration** ambiguous, hence many decoy selectors and high try counts.

- **A2 on top of A1** quotients out object-enumeration symmetry so only the semantically distinct index-based programs remain. Because the DSL is tiny after A1→A2, **every remaining program is correct** on the studied task, and time-to-solution is near zero.

10 Why A2 is needed (and helpful)

Ties in geometry (equal-area components, nearby positions) are common in ARC. Without a canonical enumeration, selectors like "smallest," "top-most," or "first component" splinter into many arbitrarily-ordered variants. A2 **chooses one consistent ordering**, eliminating these spurious degrees of freedom and making index-based rules stable across pairs.

11 Limitations & Extensions

- Our DSL is intentionally tiny; we focused on the symmetry quotient effects, not on broad coverage of ARC.

- A1's ordering rule (frequency → first occurrence) is one of many possible canonicalizations; other tasks may require different tie-breaks.

- The mapping-existence step is specialized to **select×color×recolor**. Richer action families (e.g., geometric rewrites) would need their own existence checks and gauges.

- Extensions: add shape-level canonicalizations (e.g., rotation/flip quotient), object-relation canonicalizations, and richer color policies while preserving the **validate→exist→search** pipeline.

12 Reproducibility (runnable artifacts)

- **Script:** `arc_abstraction_dsl_2.py`

- **Outputs:** `challenging_metrics.txt`, `challenging_metrics.json`

Run with `python3 arc_abstraction_dsl_2.py`. The script prints a summary ("Challenging Single-Dataset Metrics") and writes both artifacts in the working directory. The "Local-run results" above are taken from your updated run and reflected here.

13 Takeaways

Abstractions as `G + invariant`, composed as **A1→A2**, turn a symmetry-riddled search into an almost trivial one for the studied ARC family. The composition removes palette and enumeration symmetries, yields a **two-program** search space in which **both** programs are valid, and delivers **×100+** speedups—all with simple, transparent rules.

Appendix: Glossary of terms

- **ARC:** Abstraction and Reasoning Corpus; small integer-grid puzzles.
- **Symmetry quotient:** Identifying states/programs that differ only by a symmetry (palette permutation, object ordering), keeping a single canonical representative.
- **Invariant:** A condition that must hold after normalization (e.g., background `0` preserved).
- **Gauge:** The bookkeeping needed to transfer an abstract action back to the original instance (palette map, canonical order list).
- **Selector:** A rule that names a component (e.g., `index0`, `smallest`).
- **Color rule:** A rule that names a palette color (e.g., `least_frequent`).
- **Recolor:** Action that applies the chosen color to the selected component.