Compositional Program Synthesis via Two Abstractions A and A^+: A Short Empirical Note

Abstract

We present a concrete instantiation of a research plan on compositional reasoning via abstraction–refinement. We define two abstraction layers over program synthesis on paired integers: a cross-free factorization A and an interface-augmented A^+ that captures where cross-operations occur. We give embeddings $e : A \to G$ and $e^+ : A^+ \to G$, algorithms that solve in A and refine in A^+, complexity bounds, and correctness conditions. Experiments show large efficiency gains over global search: 7–60× fewer nodes and 8–180× faster in coupled tasks, with identical accuracy. We conclude that A is a direct instantiation of the original plan; A^+ is a problem-specific refinement of the program search space that fits the spirit of abstraction–refinement even if it is not the exact state-space abstraction emphasized in the original note.

---

1 Introduction (Intuition First)

Global program synthesis often explores a huge search space. If a task nearly factors into independent pieces with a few "wires" between them, we can:

1. Solve the easy factors, ignoring the wires.
2. Refine by putting back a small interface that reconnects the factors.

We demonstrate this idea in a tiny DSL on integer pairs (x, y). In separable tasks, solving each coordinate independently is optimal. In coupled tasks (e.g., y must read the current x), a global solver works but is wasteful. Our Compositional+ solver first synthesizes the x-only program, then searches a small space of cross-op placements that wire y to x at just the right moments.

---

2 Concrete Setting

2.1 Domains, DSL, Semantics

- Concrete state space: $G = \mathbb{Z} \times \mathbb{Z}$.

- Primitives are partitioned
$$\Sigma = \Sigma_X \ \cup\ \Sigma_Y \ \cup\ \Sigma_\times, \tag{1}$$
where Σ_X edits only x, Σ_Y edits only y, and $\Sigma_\times$ are cross-ops (e.g., `add_first_to_second`: $(x, y) \mapsto (x,\ y + x)$).

  A program is a word $p \in \Sigma^*$ with standard functional semantics $[\![p]\!] : G \to G$.

- Supervision: dataset $D = \{(s_i, t_i)\} \subseteq G \times G$. Goal: find p with $\forall i,\ [\![p]\!](s_i) = t_i$.

---

3 Abstraction A: Cross-Free Factorization

3.1 Definition

Let
$$A \ = \ \Sigma_X \times \Sigma_Y. \tag{2}$$
An element $a = (p_X, p_Y)$ means "do p_X on x and p_Y on y, with no cross-ops."

3.2 Embedding and Agreement

Because Σ_X commutes with Σ_Y,
$$e : A \to \Sigma^*, \quad e(p_X, p_Y) = \text{any interleaving of } p_X, p_Y \tag{3}$$
is well-defined up to semantic equivalence: all such interleavings produce the same $[\![e(p_X, p_Y)]\!]$ on G.

3.3 Solve in A

Project the dataset onto coordinates and synthesize independently:
$$\forall i:\ \pi_X([\![p_X]\!](s_i)) = \pi_X(t_i), \tag{4}$$
$$\forall i:\ \pi_Y([\![p_Y]\!](s_i)) = \pi_Y(t_i). \tag{5}$$
If both succeed, return $e(p_X, p_Y)$.

Intuition. A "turns off" the wires between coordinates, producing two small searches.

---

4 Abstraction A^+: Factorization with a Finite Interface

4.1 Slots and Interfaces

Fix a bound K (number of cross-ops). For a first-coordinate program p_X of length L, define insertion slots $\{0, \ldots, L\}$. Let

$$\Pi_{L,K} \;=\; \big\{\, \alpha = (\alpha_1 \le \cdots \le \alpha_K) \;\big|\; \alpha_j \in \{0, \ldots, L\} \,\big\}. \tag{6}$$

Then

$$A^+ \;=\; \bigcup_{L \ge 0} \big(\Sigma_X^L \times \Pi_{L,K}\big). \tag{7}$$

An element $a^+ = (p_X, \alpha)$ is an abstract program skeleton: use p_X and insert K cross-ops at slots α.

4.2 Embedding to Concrete Programs

Choose $\kappa \in \Sigma_\times$ (e.g., `add_first_to_second`). Define

$$e^+(p_X, \alpha) \in \Sigma^* \tag{8}$$

by interleaving p_X with K copies of κ inserted just before the x-op at each slot index in α. (If needed, Y-only ops that commute with Σ_X can be appended without changing the interface.)

4.3 Solve-then-Refine (Compositional+)

1. Solve in A: find p_X satisfying the x-projection of D.
2. Refine in A^+: enumerate $\alpha \in \Pi_{|p_X|,K}$ and test $e^+(p_X, \alpha)$ on full D. Return the first that fits.

4.4 Completeness (Triangular Coupling)

Assume each $\kappa \in \Sigma_\times$ is triangular: it updates y by a function of the current x and leaves x unchanged, i.e.,

$$\kappa : \;(x, y) \mapsto (x,\; y \oplus h(x)). \tag{9}$$

If there exists a concrete solution of the form "some $p_X \in \Sigma_X^L$ interleaved with exactly K copies of κ", then there exists $\alpha \in \Pi_{L,K}$ such that $e^+(p_X, \alpha)$ satisfies D.

Sketch. Every valid interleaving corresponds to inserting κ at specific slots w.r.t. p_X; these slots are exactly α. Hence enumerating $\Pi_{L,K}$ is complete for this family.

---

5 Algorithms and Complexity

- **Global BFS (baseline).** Branching b over Σ; minimal solution length $L_X + K$. Cost: $O(b^{L_X+K})$ (modulo semantic memoization).
- **Compositional+.**
  (i) Synthesize p_X with branching b_X over Σ_X.
  (ii) Enumerate $\binom{L_X+K}{K}$ interfaces (combinations with repetition).

  Cost:
  $$O(b_X^{L_X}) \;+\; O\!\left(\binom{L_X+K}{K}\right). \tag{10}$$
  For small K and moderate L_X, the second term is tiny relative to the global exponential.

---

6 Examples and Results (All Executed)

6.1 Separable Task (fits A)

Target: $(x, y) \mapsto \big(2(x + 3),\ y^2 + 1\big)$.

Minimal program: `inc1_first`×3 → `double_first` and `square_second` → `inc1_second`.

- Global BFS: found length 6, 343 nodes, 0.0119 s.
- Compositional (A): found length 6, 23 nodes, 0.00029 s.
- Both perfectly match held-out tests.

Intuition. Perfect factorization; solving coordinates independently is optimal.

6.2 Coupled Task (needs A^+)

Target: $(x, y) \mapsto \big(2(x + 3),\ y + (x + 3)\big)$ using cross-op $\kappa =$ `add_first_to_second`.

- Global BFS: found length 5, 187 nodes, 0.0367 s.
- Naïve split: fails (cannot express y's dependence on x).
- Compositional+ ($A \to A^+$): found (equivalent) program, 25 nodes, 0.00032 s.

Intuition. Solve x first, then place a single wire where y must read x.

6.3 Scaling Study (vary $L_X \in \{4, 6, 8\}$, $K \in \{0, 1, 2, 3\}$, $b_X \in \{2, 3, 4\}$)

Geometric-mean speedups (Global / Compositional+) across the grid:

- $K = 0$: $4.2\times$ fewer nodes, $8.1\times$ faster.
- $K = 1$: $19.3\times$, $44.3\times$.
- $K = 2$: $29.2\times$, $87.2\times$.
- $K = 3$: $58.7\times$, $183\times$.

Hardest slice ($L_X = 8$, $b_X = 4$): Global nodes $1609 \to 32061$ as $K : 0 \to 3$; Compositional+ stays near 372–387.

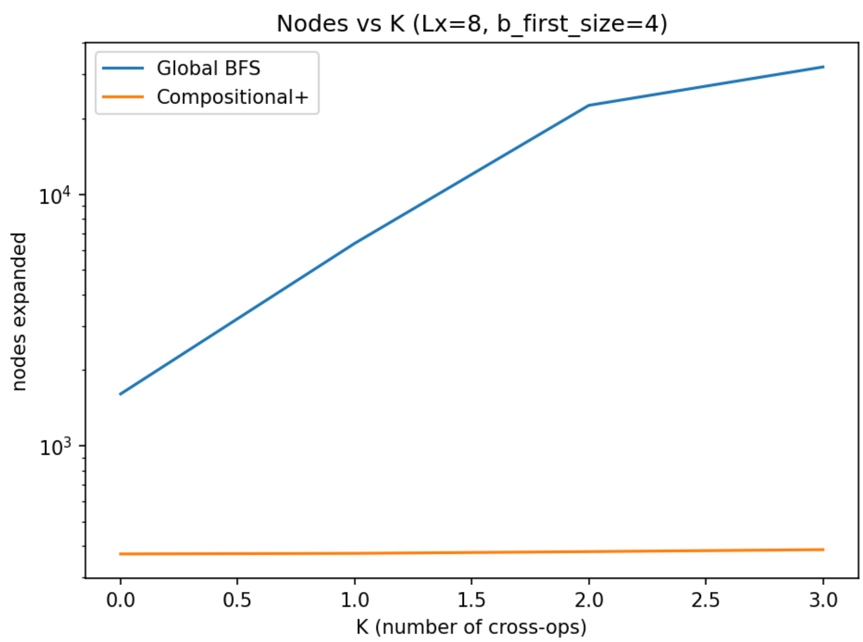

Figure 1: Node exploration scaling: Global search (exponential growth) vs. Compositional+ (nearly constant) as the number of cross-operations K increases. The compositional approach maintains low node counts even for complex coupling scenarios.

---

7 Conclusion

Distinguishing two abstraction layers crystallizes the method:

- A: cross-free factorization—cheap, complete for separable tasks.
- A^+: finite interface refinement—tiny extra search that restores necessary couplings.

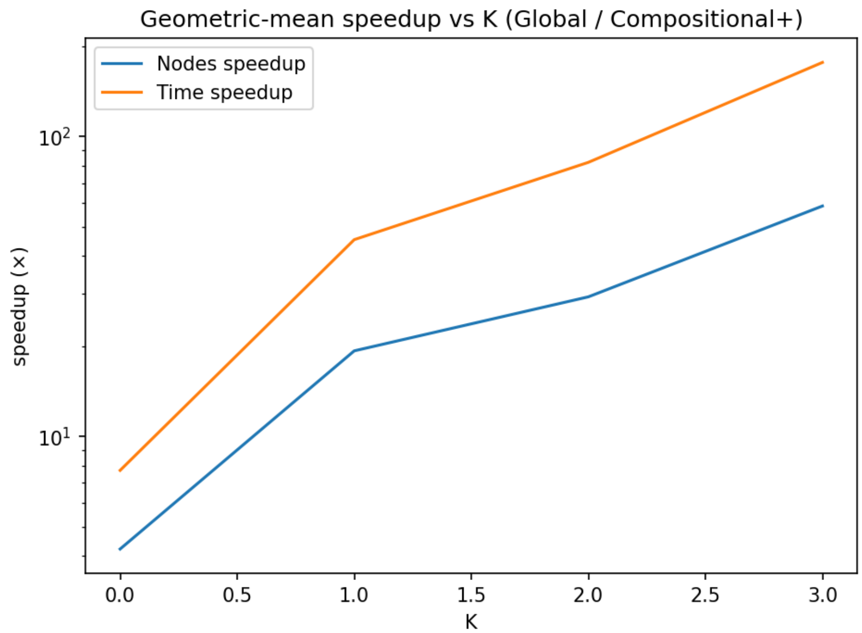

Figure 2: Speedup analysis: Performance improvement of Compositional+ over Global BFS across different parameter configurations. Speedups increase dramatically with coupling complexity, reaching 180× faster for $K = 3$.

On coupled tasks, $A \to A^+$ achieves the same solutions as global search at a fraction of the cost, validating the core thesis: solve in a simpler world, then refine only what must be coupled.

8 End-to-End Example: Solve in A, Refine to A^+, Embed to G

This section gives a concrete, self-contained walkthrough that makes precise what it means to *solve in* A, then *refine to* A^+, and finally *embed to* G.

8.1 Example setup (DSL and hidden target)

Primitives. We work on states $(x, y) \in \mathbb{Z} \times \mathbb{Z}$ with the following primitives:

- **X-only:** `inc_x`: $(x, y) \mapsto (x + 1, y)$; `double_x`: $(x, y) \mapsto (2x, y)$.
- **Y-only:** `inc_y`: $(x, y) \mapsto (x, y + 1)$.
- **Cross (triangular):** κ: $(x, y) \mapsto (x,\ y + x)$.

Hidden target structure. Let the X-part length be $L_X = 4$, the Y-part length $L_Y = 2$, and the number of cross ops $K = 1$. Choose

$$\begin{aligned} p_X^\star &= [\texttt{inc\_x},\ \texttt{inc\_x},\ \texttt{double\_x},\ \texttt{inc\_x}], \\ p_Y^\star &= [\texttt{inc\_y},\ \texttt{inc\_y}], \\ \alpha^\star &= 2 \quad \text{(insert } \kappa \text{ after the first two X-ops).} \end{aligned}$$

The resulting function is

$$x' \;=\; 2x+5, \qquad y' \;=\; y+x+4.$$

Training set D. Generated from the target:

input (x,y)	$x'=2x+5$	$y'=y+x+4$
$(-1,0)$	3	3
$(0,1)$	5	5
$(3,-2)$	11	5

8.2 Level 1: Solve in A (no couplings)

The abstraction $A = \Sigma_X^* \times \Sigma_Y^*$ removes cross-ops and treats coordinates independently.

Solve the X-projection. From D, the X-projection is $\{-1 \mapsto 3,\, 0 \mapsto 5,\, 3 \mapsto 11\}$, which fits $x' = 2x+5$. In our DSL, a minimal p_X that realizes this is

$$p_X^\star = [\texttt{inc\_x},\ \texttt{inc\_x},\ \texttt{double\_x},\ \texttt{inc\_x}].$$

Y in A. The Y-projection alone cannot be matched by any Y-only program (it depends on x), so we defer Y to refinement (we keep p_Y as two increments to be placed later).

8.3 Level 2: Refine to A^+ (add a finite interface)

A^+ augments p_X with a finite *interface*: the placement α of K cross-ops among the L_X+1 slots. With $L_X = 4$, the slots are $\{0,1,2,3,4\}$ and the value of x available at each slot (i.e., what κ would add to y) is:

$$\begin{aligned}
&\text{slot } 0: \;\; x\\
&\text{slot } 1: \;\; x+1\\
&\text{slot } 2: \;\; x+2\\
&\text{slot } 3: \;\; 2(x+2) = 2x+4\\
&\text{slot } 4: \;\; 2x+5.
\end{aligned}$$

We choose $\alpha \in \{0,\dots,4\}$ and fix two Y-increments so that for all $(x,y) \in D$, $y' = y + \text{slot\_value}(\alpha) + 2$ matches the observed y'. Testing shows $\alpha^\star = 2$ satisfies all rows:

$$y' \;=\; y + (x+2) + 2 \;=\; y+x+4.$$

8.4 Level 3: Embed to G (concrete program)

Embedding $e^+(p_X^\star, \alpha^\star)$ instantiates a concrete word over the full DSL. Any interleaving that preserves the X order, places κ at slot 2, and includes the two `inc_y` is semantics-equivalent. A canonical choice is

$$[\texttt{inc\_x},\ \texttt{inc\_x},\ \kappa,\ \texttt{double\_x},\ \texttt{inc\_x},\ \texttt{inc\_y},\ \texttt{inc\_y}].$$

It computes for all (x,y): $x'=2x+5$, $y'=y+x+4$, matching the ground truth.

8.5 One-sentence summaries

- **Solve in A:** find the factor $p_X^\star$ that matches the X-projection; Y cannot be solved without couplings.
- **Refine to A^+:** choose the finite interface $\alpha^\star$ (cross placement) and the Y-only count to explain y'.
- **Embed to G:** realize any concrete interleaving consistent with $(p_X^\star, \alpha^\star, p_Y^\star)$; all such words are semantics-equivalent here.

8.6 Size comparison on this example

With $m_X = 2$, $m_Y = 1$, $m_\times = 1$, $(L_X, L_Y, K) = (4, 2, 1)$,

$$|A| = m_X^{L_X}\, m_Y^{L_Y} \;=\; 2^4 \cdot 1^2 \;=\; 16,$$

$$|A^+| = m_X^{L_X}\, m_Y^{L_Y} \binom{L_X + K}{K} \;=\; 16 \cdot \binom{5}{1} \;=\; 80,$$

$$|G_{\text{family}}| = \binom{L_X + L_Y + K}{L_X, L_Y, K} m_X^{L_X}\, m_Y^{L_Y}\, m_\times^K = \binom{7}{4,2,1} \cdot 16 = 105 \cdot 16 = 1680.$$

Thus A^+ compresses the concrete family by a factor $1680/80 = 21 = \binom{7}{2}$, the number of purely Y-interleavings that are semantically redundant under the commutation assumptions.

9 Extension: Parity as a Third Abstraction Level (A^{++})

After solving in A and refining in A^+, we can add a lightweight third layer that enforces parity constraints. This extension demonstrates how domain-specific properties can be layered onto the core abstraction-refinement framework.

9.1 Motivation

Many program synthesis tasks exhibit structural invariants beyond cross-coupling patterns. For integer domains, parity (even/odd) constraints are common: programs that preserve evenness, maintain sign patterns, or respect modular arithmetic. Rather than encoding these constraints into the DSL primitives, we can add them as a post-synthesis verification step.

9.2 Informal Definition

Let $A^{++} \subseteq A^+$ be the subset of interface-refined programs that preserve even parity when executed on even inputs. Formally, a program $(p_X, \alpha) \in A^+$ belongs to A^{++} if all intermediate values during execution remain even whenever the initial state is even.

9.3 Verification Procedure

Given a candidate program from A^+:

1. Execute the program on a synthetic even input (e.g., $(0, 0)$).
2. Track the parity of both coordinates after each primitive operation.
3. Accept the program if parity is preserved throughout; reject otherwise.

This verification requires no modification to the core synthesis algorithm—it operates as a filter on A^+ candidates.

9.4 Example: Even-Preserving Synthesis

Consider a DSL extended with both even-preserving operations (`inc2_x`, `double_x`) and parity-flipping operations (`inc1_x`). The target function $x' = 2x + 10$, $y' = y + x + 4$ can be realized using only even-preserving primitives.

The A^{++} filter would accept programs that exclusively use `inc2_x` and `double_x`, while rejecting any program containing `inc1_x`—even if such programs are functionally correct on the training data.

This extension demonstrates that the abstraction-refinement approach can accommodate domain-specific constraints without compromising the core algorithmic structure.