Solver-Aware Decompositions for Programming-by-Example: When Dividing Requires Knowing How to Conquer

Janis Zenkner **Tobias Sesterhenn** **Tim Grams** **Christian Bartelt**
Clausthal University of Technology
Clausthal-Zellerfeld, Germany
janis.zenkner@tu-clausthal.de

Abstract

Decomposition-based Programming-by-Example splits a task into subtasks: a decomposer predicts intermediate subgoals, and a learned synthesizer generates a program for each subgoal. Execution-decomposition approaches train the decomposer to imitate ground-truth (GT) subgoals and thereby treat decomposition quality as a property of the task alone. A learned synthesizer, however, has its own inductive biases, and GT subgoals reflect how the annotator factorized the task, not what this synthesizer can solve. A decomposer that imitates them can therefore propose subgoals that are logically valid but intractable for the synthesizer. Motivated by this observation, we propose Solver-Aware Decomposition (SAD), which keeps supervised training on GT subgoals as a structural scaffold and additionally trains the decomposer online with self-critical policy gradients against a frozen synthesizer. Each sampled subgoal is rewarded with the negative cross-entropy of the synthesizer on the target program, a continuous signal of how difficult the subtask is for this synthesizer. Our experiments show an *accuracy paradox*: with the synthesizer fixed, a GT-supervised decomposer agrees more often with the GT subgoals, yet its subgoals lead to lower synthesis accuracy, although the synthesizer was trained on the same GT data. On two list-processing domains, SAD improves synthesis and task accuracy over the GT-supervised decomposer, both in-distribution and under length generalization; the same holds under zero-shot transfer to an external benchmark and for a fine-tuned language model. Even an oracle that receives the GT subgoals at inference time, with the same synthesizer and search, fails on tasks that SAD solves, and fewer than 2% of SAD's solutions to these tasks match the GT program. Decomposition quality is therefore not a property of the task alone: it depends on the synthesizer that executes the subgoals, and the decomposer should be trained against it.[1]

1 Introduction

Divide-and-conquer is among the most powerful principles in algorithm design: decompose a complex problem into manageable subproblems, solve each, and recombine [17]. How useful a given decomposition is, however, depends on the solver that executes it [1]: a partition that a comparison-based sorter can exploit may be of no use to a radix sorter. The same intermediate structure that one algorithm exploits, another cannot act on at all. Therefore, a decomposition that is optimal for an ideal solver may be suboptimal for a bounded one, and since practical solvers are bounded [36], decomposition quality must be defined relative to solver capabilities.

[1] Code: https://github.com/jzenkner/SAD

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

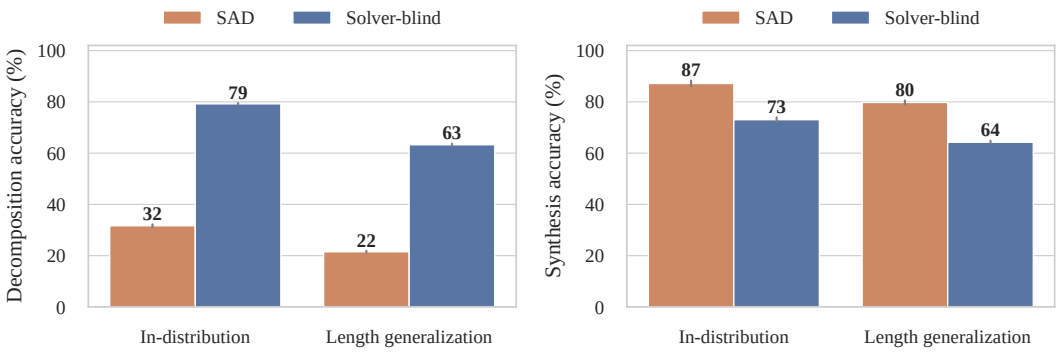
We study this dependence in Programming-by-Example (PBE), where a task is specified by input-output (I/O) examples and the goal is a program in a Domain-Specific Language (DSL) that is consistent with all of them [10]. Consider a task that maps the input list `[9,1,4,0,1]` to `[14,4]`. In a DSL with operations such as `Map`, `Filter`, and `ZipWith`, the target program is to subtract 2 from every element, keep the positive values, and double them: `x1 = Map - 2 x0 | x2 = Filter > 0 x1 | x3 = Map * 2 x2`. Recent approaches build such programs step by step [39, 49]: a decomposer predicts an intermediate subgoal, here the output of the first step `[7,-1,2,-2,-1]`, and a synthesizer then generates a subprogram that produces it, here `Map - 2`. The decomposer thus decides *what* to solve next, and the synthesizer decides *how*.

To learn the first role, execution-decomposition approaches train the decomposer to imitate ground-truth (GT) subgoals [39, 49], assuming that similarity to GT is a reliable proxy for usefulness to the synthesizer. The assumption is plausible, since the synthesizer is trained on the same GT data. A learned synthesizer, however, develops its own inductive biases and does not reproduce the annotator's choices exactly [40], so a GT subgoal can still lie outside what the synthesizer can solve. Supervised training cannot detect this, because it assigns a similar loss to a subgoal that is close to GT but unrealizable and to one that is slightly off but synthesizable.

Take the example from above. A decomposer trained only on GT subgoals predicts `[7,0,2,3,-1]` as the first subgoal, and since this list matches the GT subgoal in three of five positions, the supervised loss is small. However, no DSL operation maps the input to this list (no `Map` can, because the two `1`s in the input would have to map to different values), so no subprogram is consistent with this subgoal, and the synthesizer has no correct target to aim for. Training with synthesizer feedback separates these cases: a subgoal from which the synthesizer cannot produce the target subprogram receives a low reward however close it is to GT, while a deviation the synthesizer can act on is penalized less. A decomposer trained this way therefore predicts the realizable subgoal `[7,-1,2,-2,-1]`, from which the synthesizer solves the entire task. Closeness to GT did not make the first subgoal useful; what decides is whether the synthesizer can act on it. Appendix B gives two further cases from our evaluation: a subgoal that no single operation produces (Figure B.1), and a realizable GT path that the synthesizer cannot follow (Figure B.2).

The same pattern appears in aggregate. We call a decomposer that is trained only on GT subgoals *solver-blind*. With the synthesizer fixed, it matches the GT subgoals more often than a decomposer trained with synthesizer feedback, yet its subgoals lead to lower synthesis accuracy (Figure 1). We refer to this result as the *accuracy paradox* (Section 5.2): the solver-blind decomposer learns how the annotator decomposes tasks, which is not how the synthesizer solves them most easily.

Accounting for the downstream solver is not new in itself: prior work selects supervision targets with the solver in mind, for example by choosing among GT-derived sketches by their estimated enumeration cost [26]. We instead train the decomposer online against the learned synthesizer that executes its subgoals, which leads to the following contributions:



(a) Decomposition accuracy (b) Synthesis accuracy

Figure 1: Accuracy paradox on Deepcoder. (a) Decomposition accuracy: predicted subgoal matches the GT subgoal exactly. (b) Synthesis accuracy: the synthesizer produces the correct subprogram from the predicted subgoal. Both are single-step metrics under teacher forcing. Solver-blind training reaches higher decomposition accuracy but lower synthesis accuracy than solver-aware training.

1. We introduce Solver-Aware Decomposition (SAD), which keeps supervised training on GT subgoals as a structural scaffold and adds Self-Critical Sequence Training (SCST) against a frozen learned synthesizer. The synthesizer scores each sampled decomposition online by its cross-entropy (CE) on the GT program, a continuous reward that stays informative for partially correct programs. Because the greedy decomposition serves as baseline, the advantage measures how much easier the sampled subgoal is for the synthesizer than the greedy one.
2. On Deepcoder and Lambdabeam, the accuracy paradox holds in every setting, and SAD improves task accuracy over the solver-blind decomposer both in-distribution and under length generalization (Section 5.2). The improvement carries over to zero-shot transfer to the benchmark of Rule et al. [35] (Appendix F) and to a fine-tuned Llama-3.1-8B (Section 6). On Robustfill, where the decomposition order is fixed, both decomposers perform equally, as predicted (Section 5.2).
3. With the same synthesizer and search, SAD solves tasks that an oracle with GT subgoals fails, and fewer than 2% of these solutions match the GT program (Section 5.2). GT decompositions are therefore not optimal for this synthesizer, and different decomposers reach different programs, which makes decomposition diversity a resource for future work, for example for ensembles or robustness under distribution shift.

2 Related Work

The central claim of this paper rests on a specific structural condition: a learned decomposer interacting with a bounded, learned synthesizer whose tractable distribution can diverge from the annotator's choices. This condition is absent in much prior work, and recognizing precisely where it does and does not arise is what defines the scope of SAD's contribution.

Enumerative [41, 10, 2, 7], neurally guided [3, 48, 20], and Large Language Model (LLM)-based approaches [31, 21, 28, 27, 23] generate the program from the task specification without a separate decomposer. In these approaches, there is no decomposer–synthesizer interface to align.

Other approaches use intermediate structure, such as sketches [25], latent codes [12], plans or as-needed decompositions [50, 30, 46], and compositional modules [5, 16]. Hierarchical programmatic Reinforcement Learning (RL) methods such as LEAPS [43] and HPRL [22] execute their programs with a deterministic interpreter. In both groups, the intermediate structure is either produced and consumed by one model or executed by a fixed procedure, so there is no separately trained learned consumer whose preferences can diverge from the annotation.

REINFORCE [45], sequence-level training [33], SCST [34], and CodeRL [19] optimize the model that produces the final output. SAD uses the same machinery, but the optimized model produces subgoals that a separate, frozen synthesizer consumes. The feedback therefore changes the problem handed to the synthesizer, not the synthesizer's solution. Adaptive curriculum methods [9, 24, 29] also use downstream performance to guide learning, but they select tasks from a fixed distribution instead of generating intermediate problems.

Exedec [39] and TIIPS [49] use the same pipeline as SAD: a learned decomposer predicts subgoals, and a learned synthesizer generates subprograms for them. Both train the decomposer to imitate GT subgoals, so the setting above arises, but the synthesizer's behavior is never used as a training signal. SAD keeps this pipeline and changes how the decomposer is trained. SketchAdapt [26] goes beyond GT imitation. It derives candidate sketches from the GT program and estimates the enumeration time of each from the likelihood under a learned recognizer. The sketch generator is then trained to imitate a candidate whose estimate fits a sampled time budget. SketchAdapt thus accounts for the solver through a proxy for its capabilities and does not run synthesis during training. SAD instead samples subgoals from the current policy, scores each sample by the CE of the frozen synthesizer on the target program, and updates the policy with SCST.

Outside program synthesis, solver feedback is used to train feasibility predictors for a fixed task-and-motion planner [47, 44], whereas SAD updates the model that generates the intermediate representation. Inala et al. [15] adapt supervision through trajectories generated by a teacher. In HyperTree Proof Search [18], the learned policy is part of the prover and is trained together with proof search, whereas the synthesizer in SAD is trained separately and then frozen.

In summary, SAD combines four elements: a frozen learned synthesizer whose predictive distribution defines the reward, a continuous CE reward instead of binary success, online scoring of every decomposition sampled by the current policy, and policy-gradient updates of the decomposer.

3 Background: Decomposition-based Synthesis

In PBE, a task is given as a set of I/O examples $\mathcal{E} = \{(I_i, O_i)\}_{i=1}^{n}$, and the goal is to find a program p in a DSL such that $p(I_i) = O_i$ for all $(I_i, O_i) \in \mathcal{E}$ [10]. A DSL is a small programming language whose operations are tailored to one class of tasks, such as list or string transformations [11]. Restricting programs to such a language keeps the search space small while the task-specific operations keep the target programs expressible, so the design of a DSL trades solution coverage against search tractability [41, 4]. We build on Exedec [39], which solves a task by decomposing it sequentially into smaller synthesis problems: it alternates decomposition, synthesis, execution, and state update until the specification is satisfied or a step limit T is reached.

Each iteration t transforms the current state $(I^{(t)}, O^{(t)})$, which initially is the task itself, in four stages. In the *decomposition* stage, the decomposer reads the current state and predicts an intermediate subgoal $O^{(t)}_{\text{pred}}$, i.e., the outputs that the next subprogram should produce from the current inputs. Together with these inputs, the subgoal forms the subtask $(I^{(t)}, O^{(t)}_{\text{pred}})$, which is atomic: it is meant to be solved by a single minimal subprogram. In the *synthesis* stage, the synthesizer, a learned sequence model, reads the I/O examples of this subtask and generates such a subprogram $P^{(t)}$, aiming to map each input $I^{(t)}_i$ to its subgoal $O^{(t)}_{\text{pred},i}$. In the *execution* stage, $P^{(t)}$ is run on the current inputs, which yields the results $E^{(t)}_i = P^{(t)}(I^{(t)}_i)$. If these results equal the target outputs O_i for all examples, the subprograms found so far form the solution; otherwise, the *update* stage incorporates $E^{(t)}$ into the state, and the next iteration begins. The execution results are never compared with the predicted subgoal, so a subprogram whose output differs from $O^{(t)}_{\text{pred}}$ is still kept; the subgoal thus guides the synthesizer without constraining it. Algorithm 1 describes inference, where the synthesizer receives the predicted subtask; during training, the SAD synthesizer is conditioned on the full current task instead (Section 4.1).

The update stage depends on the domain, and it determines how much freedom the decomposer has. In Deepcoder and Lambdabeam [3, 38], each step adds its output to the inputs as a new variable, while the target $O^{(t)}$ stays fixed. A subgoal that is plausible locally but does not lead toward the target therefore passes unnoticed at that step: it uses up one of the T steps, and later decompositions are conditioned on it. This is why the cost of misaligned subgoals grows with program length

Algorithm 1 Inference loop of decomposition-based synthesis (Exedec).

Require: Decomposition step limit T
function GENERATEPROGRAM($\{(I_i, O_i)\}$)
  $t \leftarrow 1$
  $(I^{(1)}_i, O^{(1)}_i) \leftarrow (I_i, O_i),\ \forall i$
  while $t \leq T$ **do**
    $\{O^{(t)}_{\text{pred},i}\} \leftarrow$ DecompositionModel($\{(I^{(t)}_i, O^{(t)}_i)\}$) ▷ Predict subgoals
    $P^{(t)} \leftarrow$ Synthesizer($\{(I^{(t)}_i, O^{(t)}_{\text{pred},i})\}$) ▷ Generate subprogram
    $E^{(t)}_i \leftarrow$ Execute($P^{(t)}, I^{(t)}_i$), $\forall i$
    if $\forall i.\ E^{(t)}_i = O_i$ **then** ▷ Solution found
      return Combine($P^{(1)}, \ldots, P^{(t)}$)
    if RobustFill **then** ▷ Update program state (domain-specific)
      $(I^{(t+1)}_i, O^{(t+1)}_i) \leftarrow (I^{(t)}_i,$ REMOVEPREFIX$(O^{(t)}_i, E^{(t)}_i))$
    else
      $(I^{(t+1)}_i, O^{(t+1)}_i) \leftarrow (I^{(t)}_i \cup E^{(t)}_i,\ O^{(t)}_i)$
    $t \leftarrow t + 1$
  return Failure

(Appendix D.2). In Robustfill [6], by contrast, each step removes its output from the front of the remaining target string, so the residual shows what is left to produce. Since each step must produce a prefix of this residual, the decomposition order is fixed; Section 5.1 discusses what this means for the evaluation.

If no solution is found within T iterations, the loop returns a failure. At test time, a beam search runs several rollouts of this loop in parallel (Appendix C.2).

4 Solver-Aware Decomposition

SAD keeps the pipeline of Section 3 and changes only how the decomposer is trained. A synthesizer is first trained and then frozen (Section 4.1); the decomposer is then trained against it with GT supervision and SCST (Section 4.2), and at inference both models are combined in a beam search (Section 4.3). Throughout, we write $s_t = (I_i^{(t)}, O_i^{(t)})$ for the state at step t and p_t^* and O_t^* for the GT subprogram and the GT subgoal of this step.

4.1 Synthesizer Training

The synthesizer is trained with teacher forcing to generate the next GT subprogram from the full current task, by minimizing the CE over program tokens,

$$\mathcal{L}_{\text{syn}} = \text{CE}\big(\text{Synth}(s_t),\, p_t^*\big).$$

Unlike the synthesizer used in the original Exedec study [39], which is trained on GT subtasks, it must therefore infer which subtask comes next as well as how to implement it. This makes the synthesizer informative also on subgoals that leave the GT trajectory, which is where SAD needs its feedback. At inference, the synthesizer receives the subtask $(I^{(t)}, O_{\text{pred}}^{(t)})$ induced by the predicted subgoal, as in Algorithm 1. After training, the synthesizer is frozen and serves as the fixed solver against which the decomposer learns to propose subgoals. The solver-blind baseline uses the same synthesizer (Section 5.1, Appendix C.1).

4.2 Solver-Aware Training

The decomposer defines a distribution $\pi_\theta(O \mid s_t)$ over subgoals O. SAD rewards a subgoal by how well the frozen synthesizer recovers the GT subprogram from the subtask that the subgoal induces:

$$R(O) = -\text{CE}\big(\text{Synth}(I^{(t)}, O),\, p_t^*\big).$$

A subgoal thus receives a high reward if it places the synthesizer where p_t^* is likely. In this reward, p_t^* fixes *what* must be synthesized, and the subgoal determines *how* easy this is for the synthesizer; the GT subgoal does not enter it. The GT program serves as a single verifiable target, since accepting alternative correct programs would require deciding semantic equivalence between programs. Because the CE is continuous, the reward also grades subgoals from which the synthesizer generates a partially correct program.

We optimize π_θ with SCST [34]; Figure 2 shows one training step. For each training state s_t, SAD proceeds as follows:

1. Sampling: draw a subgoal $O^s \sim \pi_\theta(\cdot \mid s_t)$.
2. Baseline: decode the greedy subgoal O^g from $\pi_\theta(\cdot \mid s_t)$.
3. Evaluation: compute $R(O^s)$ and $R(O^g)$ with the frozen synthesizer.
4. Advantage: $A = R(O^s) - R(O^g)$.
5. Update: increase the log-probability of O^s if $A > 0$ and decrease it if $A < 0$, i.e., minimize

$$\mathcal{L}_{\text{RL}} = -\mathbb{E}_{O^s \sim \pi_\theta(\cdot \mid s_t)}\big[A \cdot \log \pi_\theta(O^s \mid s_t)\big].$$

Since both subgoals are scored by the same synthesizer against the same target, A measures how much easier the synthesizer finds the sampled subgoal than the greedy one. If both subgoals lead to the same program, A compares the synthesizer's confidence in this program under each; if they lead to different programs, A also favors the subgoal under which the synthesizer is closer to p_t^*

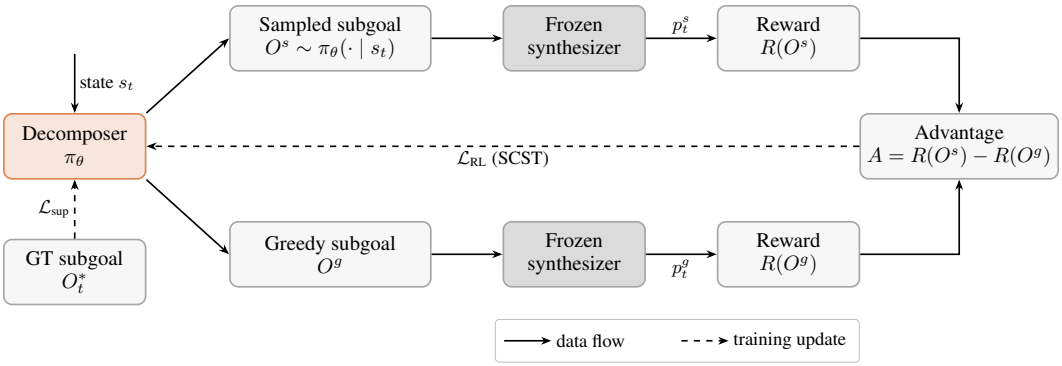


Figure 2: One SAD training step. For the state s_t, the decomposer samples a subgoal O^s and decodes a greedy subgoal O^g. The frozen synthesizer scores both by its CE on the GT subprogram p_t^*, and the advantage A weights the policy-gradient update $\mathcal{L}_{\text{RL}}$. The supervised loss $\mathcal{L}_{\text{sup}}$ on the GT subgoal O_t^* is applied in the same step.

(Appendix D.6). SCST is REINFORCE [45] with the greedy subgoal of the current policy as baseline, which reduces the variance of the gradient without a learned value model. Each training state yields one subgoal whose reward the frozen synthesizer returns directly, so there is no long-horizon credit assignment that would call for a value-based method such as PPO [37].

The supervised loss $\mathcal{L}_{\text{sup}} = \text{CE}\big(\pi_\theta(\cdot \mid s_t), O_t^*\big)$ on the GT subgoal complements $\mathcal{L}_{\text{RL}}$: it keeps the decomposer in the space of meaningful subgoals, and $\mathcal{L}_{\text{RL}}$ moves it within this space toward subgoals that the synthesizer can solve. Without $\mathcal{L}_{\text{sup}}$, the policy collapses to degenerate subgoals that the synthesizer tolerates on training data, with a training decomposition accuracy of 2–9% and a test task accuracy below 20% (single seed, Appendix D.3). The full training objective is

$$\mathcal{L} = \mathcal{L}_{\text{sup}} + \mathcal{L}_{\text{RL}} - \lambda H(\pi_\theta), \quad \lambda = 0.01,$$

where the entropy term $H(\pi_\theta)$ keeps the sampled subgoals diverse and prevents early convergence to deterministic decompositions.

4.3 Inference

At inference, SAD runs the loop of Algorithm 1. Instead of a single rollout, a beam search runs over full decomposition-synthesis trajectories, where each beam holds a partial program. Candidates are scored by the sum of the decomposer and synthesizer log-probabilities and pruned if they fail to execute or are redundant with a higher-scoring candidate; all methods use a beam size of 10 (Appendix C.2).

5 Experiments with Transformers Trained from Scratch

We train Transformer decomposers and synthesizers from scratch on three PBE domains and compare SAD with a solver-blind baseline that shares its synthesizer, data, and search, so that the two differ only in the training signal of the decomposer. Section 6 repeats this comparison with pretrained LLMs.

5.1 Evaluation

Domains Deepcoder [3] operates on integer lists with a DSL of first- and higher-order functions such as `Map` and `Filter`. The output of each step becomes a named variable that later steps can use, and a task usually admits several valid programs and decompositions. Lambdabeam [38] extends Deepcoder with lambda functions built during synthesis and a conditional `If`, which enlarges the search space, so a subgoal that the synthesizer cannot act on costs more search. Robustfill [6] covers string transformations, where a program is a concatenation of subprograms. Its residual state update fixes the decomposition order (Section 3), so the decomposer has no alternative decompositions to choose among. We therefore use Robustfill as a negative control: if the gains of SAD come from

adapting decompositions to the synthesizer, they should vanish in this domain. Appendix A gives the DSLs, example tasks, and data generation.

Settings Each domain is evaluated in-distribution on a standard test split and under length generalization, where models are trained on shorter programs and tested on longer ones: lengths 1–4 for training and 5 for testing in the list domains, and lengths 1–6 for training and 7–10 for testing in Robustfill. In both settings, training and test tasks are generated by the same procedure from the same DSL. To test whether the gains carry over to tasks from an independent source, we additionally use the list-function benchmark of Rule et al. [35], which is part of BIG-bench [42]. Its 250 tasks map integer lists to integer lists and were designed to study how humans learn rules, not generated from a synthesis DSL. We evaluate the Lambdabeam-trained models without retraining on the 129 deduplicated tasks for which enumerative search finds a Lambdabeam program, so this setting is a zero-shot transfer to an external task distribution (Appendix F).

Methods SAD differs from Exedec [39] in two respects: the input on which the synthesizer is trained (Section 4.1) and the training signal of the decomposer. To separate the two, we compare three methods: Exedec, with a synthesizer trained on GT subtasks and a decomposer trained on GT subgoals; the solver-blind baseline, which is Exedec with the synthesizer of SAD, trained on the full current task, and a decomposer trained on GT subgoals; and SAD, with the same synthesizer as the solver-blind baseline and a decomposer trained with $\mathcal{L}_{\text{sup}}$ and SCST. At inference, all three methods give the synthesizer the subtask induced by the predicted subgoal. Since SAD and the solver-blind baseline share architecture, synthesizer weights, training data, and search, our conclusions rest on this pair; Exedec connects the results to prior work, and its difference to SAD combines both changes. Appendix C.1 details the training of each method.

GT oracle The GT oracle replaces the predicted subgoals with GT subgoals at inference and uses the same synthesizer and beam as SAD. It gives an approximate ceiling for a decomposer that reproduces GT exactly, and the tasks that SAD solves but the oracle does not show where GT subgoals are not the best choice for this synthesizer. Appendix D.4 adds a beam oracle that separates which decompositions a model generates from how it ranks them.

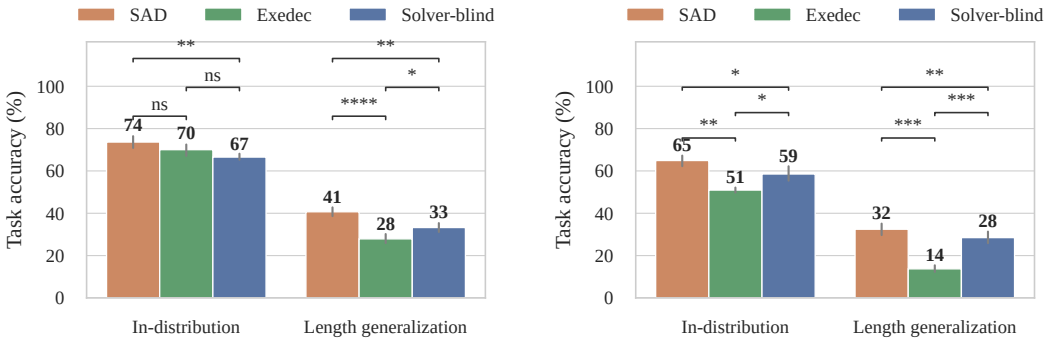
Metrics The primary metric is task accuracy, the percentage of tasks for which the synthesized program satisfies all I/O examples, including a held-out one. Two single-step diagnostics, both measured under teacher forcing, complement it. Decomposition accuracy is the percentage of steps in which the predicted subgoal equals the GT subgoal, and synthesis accuracy is the percentage of steps in which the synthesizer generates exactly the GT subprogram from the predicted subgoal. The first measures agreement with GT, the second how useful the subgoals are to the synthesizer; together they expose the accuracy paradox.

Setup All models are 3-layer Transformers with embedding dimension 512 and hidden dimension 1024. Inference uses a beam size of 10 and a step limit of $T = 10$ for the list domains and $T = 20$ for Robustfill. We follow a leave-one-out protocol in which the models receive $n - 1$ I/O examples and a solution counts only if it also satisfies the held-out example. Each result is averaged over 5 seeds with 1000 test tasks per seed, and differences are tested with paired t-tests at the 5% level. Appendix C gives the full training and inference details.

5.2 Results and Discussion

SAD solves more tasks than the solver-blind baseline, although its subgoals agree less often with the GT subgoals. Its subgoals also lead to a higher synthesis accuracy, even though the synthesizer was trained on the same GT data, and they let SAD solve tasks on which even an oracle with GT subgoals fails. On Robustfill, where the decomposition order is fixed, the gains disappear, as predicted.

Solver-aware training solves more tasks Figure 3 reports task accuracy on the list domains. In both domains and both settings, SAD solves more tasks than the solver-blind baseline. On Deepcoder, it reaches $73.6\% \pm 3.8\%$ against $66.5\% \pm 1.8\%$ in-distribution and $40.7\% \pm 2.6\%$ against $33.3\% \pm 2.7\%$ under length generalization; on Lambdabeam, it reaches $64.9\% \pm 3.2\%$ against $58.5\% \pm 4.4\%$ and $32.5\% \pm 3.5\%$ against $28.5\% \pm 3.4\%$. All four differences are statistically significant (paired t-tests, Appendix D.1). The gain also carries over to tasks from an independent source: without retraining, the Lambdabeam-trained SAD model solves $42.3\% \pm 7.1\%$ of the Rule et al. tasks, against $33.0\% \pm 3.5\%$ for the solver-blind baseline (Appendix F). Since the two methods share synthesizer, data, and search, this shows that the training signal of the decomposer accounts for



(a) Deepcoder (b) Lambdabeam

Figure 3: Task accuracy on Deepcoder and Lambdabeam, in-distribution and under length generalization, averaged over 5 seeds. Significance levels of the paired t-tests: ns: $p > 0.05$, *: $p < 0.05$, **: $p < 0.01$, ***: $p < 0.001$, ****: $p < 0.0001$.

the gain, and the transfer result indicates that the learned subgoals remain usable on tasks outside the training distribution. Exedec lies between the two methods on Deepcoder in-distribution and below both in the other three settings.

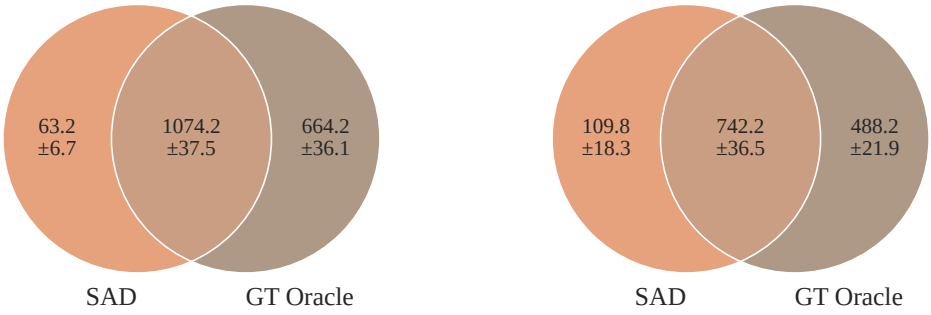
Compared to the solver-blind baseline, the relative gain is larger under length generalization than in-distribution: 22.4% against 10.7% on Deepcoder and 13.9% against 10.9% on Lambdabeam. The gap also widens with program length (Appendix D.2), although both decomposers are trained on single steps only. This indicates that solver-aware training matters more the further the test tasks lie from the training data. Two reasons are likely, although we have not tested them separately: on task structures not seen during training, imitating GT subgoals offers less guidance, and under the state update of the list domains (Section 3), a subgoal that the synthesizer cannot act on uses up a step and conditions all later steps, so errors accumulate with program length.

The accuracy paradox Table 1 compares the two decomposers at the level of single steps. The solver-blind decomposer reproduces the GT subgoal two to three times as often as SAD in every setting. Yet SAD reaches the higher synthesis accuracy in every setting. In other words, the subgoals of SAD differ from the GT subgoals more often but lead the synthesizer to the GT subprogram more often. This shows that a subgoal does not need to equal the GT subgoal to lead the synthesizer to the GT subprogram, and which subgoals do so depends on the inductive biases of the synthesizer. SAD trades agreement with GT for subgoals that this synthesizer resolves more reliably, so for this synthesizer, agreement with the GT subgoal is a weak indicator of whether a subgoal is useful. The decrease in decomposition accuracy also shows that the reward against the GT program does not pull the decomposer back toward the GT subgoals (Appendix D.6).

Generated decompositions A beam oracle that selects the GT decomposition wherever it appears in the beam improves task accuracy by less than 2 percentage points for both methods, and even with this oracle, the solver-blind baseline stays below SAD without it. This shows that ranking the GT subgoals higher does not close the gap, which therefore lies in the decompositions that each decomposer generates (Appendix D.4).

Table 1: Single-step decomposition and synthesis accuracy (%) under teacher forcing, mean $\pm$ std. over 5 seeds. The solver-blind decomposer agrees more often with the GT subgoals, whereas the subgoals of SAD more often lead the synthesizer to the GT subprogram.

Domain	**Method**	**Decomposition Acc.**		**Synthesis Acc.**	
		Test	Length Gen.	Test	Length Gen.
Deepcoder	Solver-blind	79.2 ± 0.4	63.3 ± 0.6	73.0 ± 1.2	64.3 ± 0.8
	SAD	31.7 ± 0.7	21.5 ± 0.5	87.2 ± 1.3	79.8 ± 1.1
Lambdabeam	Solver-blind	69.9 ± 0.7	48.5 ± 0.7	54.2 ± 0.5	41.3 ± 0.5
	SAD	33.7 ± 0.3	21.4 ± 0.3	61.9 ± 1.0	49.0 ± 1.2



(a) Deepcoder (b) Lambdabeam

Figure 4: Tasks solved by SAD and by the GT oracle, accumulated over the in-distribution and length-generalization test sets (mean $\pm$ std. over 5 seeds). The oracle solves more tasks overall, but a subset of tasks is solved only by SAD.

SAD solves tasks on which GT subgoals fail With GT subgoals, the same synthesizer and search solve $94.7\% \pm 0.3\%$ and $79.8\% \pm 1.9\%$ of the tasks on Deepcoder (in-distribution and length generalization) and $89.6\% \pm 2.1\%$ and $51.3\% \pm 5.5\%$ on Lambdabeam. These values are well above those of SAD, so a better decomposer still has considerable headroom. At the task level, however, SAD solves 63.2 ± 6.7 tasks on Deepcoder and 109.8 ± 18.3 on Lambdabeam on which the oracle fails, counted over both settings with 2×1000 tasks per seed (Figure 4). They differ from the GT programs in length and primitive usage and fewer than 2 percentage points of these solutions match the GT program on Deepcoder and none on Lambdabeam (Appendix D.5). This shows that for these tasks the GT decomposition is not the best one for this synthesizer, and SAD reaches a different program instead, on Lambdabeam mostly a shorter one. It does not show that the decompositions of SAD are better in general, as the oracle solves considerably more tasks. The accuracy paradox thus extends from single subgoals to complete trajectories.

No gain without decomposition ambiguity On Robustfill, SAD and the solver-blind baseline reach the same task accuracy: $94.1\% \pm 0.3\%$ against $94.1\% \pm 0.4\%$ in-distribution and $90.8\% \pm 0.3\%$ against $90.8\% \pm 0.5\%$ under length generalization. This is the outcome predicted in Section 5.1: when the decomposition order is fixed, SAD performs like supervised training, and the additional RL loss does not reduce accuracy. It also indicates that the gains in the list domains do not stem from a generic effect of the additional loss, such as regularization, which would appear here as well, but from the choice among decompositions that Robustfill does not offer. Overall, solver-aware training thus improves task accuracy where a task admits several decompositions and the synthesizer prefers some of them, and has no effect where the decomposition is fixed.

6 Experiments with Pretrained LLMs

Section 5 uses Transformers trained from scratch on the target DSL; here we ask whether the effect holds when decomposer and synthesizer are a pretrained LLM. The mismatch between GT subgoals and the subgoals that the synthesizer can solve may be larger in this case, because the inductive biases of an LLM come mainly from pretraining and not from the DSL. Solver-aware training does not depend on where these biases come from, since it trains the decomposer against the synthesizer as it is.

6.1 Evaluation

Models and training We use Llama-3.1-8B [8] with two LoRA adapters [13], one as decomposer and one as synthesizer. Both adapters are fine-tuned on Deepcoder for 1000 steps, after which the synthesizer is frozen. From the same decomposer checkpoint, we then train for 250 further steps in two ways: using supervised fine-tuning (SFT) with $\mathcal{L}_{\text{sup}}$ only, which is the solver-blind control at equal budget, and with the full objective of Section 4.2 (SAD). Both conditions use the same data and settings.

Table 2: Llama-3.1-8B on the Deepcoder LLM benchmark (in-distribution, single seed).

Condition	Solved (of 200)	Solved (%)
Off-the-shelf, direct synthesis	5	2.5
Off-the-shelf, with decomposition	15	7.5
Fine-tuned starting checkpoint	56	28.0
+ 250 steps SFT (solver-blind)	61	30.5
+ 250 steps SAD	69	34.5

Benchmark and baselines We evaluate on the 200-task Deepcoder LLM benchmark of Exedec [39], with at most 5 decomposition steps, one sample per prompt at temperature 0, and a single training seed. Following Exedec, programs are written in a Pythonic version of the Deepcoder DSL, in which each primitive except `Scanl1` becomes an equivalent Python one-liner (Appendix E.1). This representation is closer to the code that the LLMs have seen during pretraining. We evaluate in-distribution only, because we cannot control the pretraining distribution and thus cannot construct a length-generalization split that the model has not seen. As reference, we evaluate the off-the-shelf Llama-3.1-8B with direct synthesis and with decomposition, where the same model predicts the subgoals and synthesizes the subprograms; Appendix E.1 adds two Qwen2.5 models.

6.2 Results and Discussion

Table 2 reports the results. Off-the-shelf, all three LLMs solve at most 10% of the tasks, with or without decomposition (Appendix E.1), which shows that these models need fine-tuning on the DSL to reach a useful accuracy. After fine-tuning, the starting checkpoint solves 28.0% of the tasks. Further training improves this by 2.5 points with SFT (9 tasks gained, 4 lost) and by 6.5 points with SAD (17 gained, 4 lost), so that SAD is 4.0 points ahead of SFT at equal budget (13 gained, 5 lost). This is the same ordering as with the transformers trained from scratch, and it shows that further supervised training accounts for less than half of the gain of SAD over the starting checkpoint. The results thus indicate that solver-aware training carries over to pretrained LLMs, in line with the results of Section 5.2.

7 Conclusion

SAD trains the decomposer against a frozen learned synthesizer, with supervised training on GT subgoals as a scaffold and SCST with the CE of the synthesizer on the GT program as reward. Although the solver-blind decomposer agrees more often with the GT subgoals, SAD reaches a higher synthesis and task accuracy on Deepcoder and Lambdabeam, under zero-shot transfer, and with a fine-tuned LLM, and it solves tasks on which the GT subgoals fail; on Robustfill, where the decomposition order is fixed, both perform equally. For this synthesizer, a subgoal that is faithful to GT can thus be one that it cannot act on, and a subgoal that deviates from GT can be one that it solves. Decomposition quality therefore depends on the synthesizer that executes the subgoals, and the decomposer can be trained against it.

The same situation may arise wherever a learned model produces intermediate representations for a separately trained component with its own inductive biases. Supervision with GT intermediates then does not reflect what this component can execute, and training the producer against the component is one way to account for it. Since SAD and the GT oracle solve partly different tasks with different programs, decomposition diversity is a resource for future work, for example for ensembles of decomposers or for robustness under distribution shift.

Limitations SAD trains the decomposer for one frozen synthesizer, so it must be retrained when the synthesizer changes; joint training of both is not studied. Each subgoal is rewarded for its own step only, so the granularity of the decomposition follows the GT programs; multi-step rewards could let both models adapt the subtask size. The reward needs a GT program at training time, which is standard in PBE but not always available, and a learned synthesizer: a search-based solver would only provide search success or cost, which we expect to be sparser, since it does not separate decompositions that all succeed, but have not tested. Finally, the LLM experiment covers one model, one seed, and in-distribution tasks only.

Acknowledgments

This work was supported by the German Federal Ministry for Research, Technology, and Space (BMFTR) and the German Federal Ministry for Economic Affairs and Energy (BMWE).

References

[1] Andrew Allman, Wentao Tang, and Prodromos Daoutidis. Towards a generic algorithm for identifying high-quality decompositions of optimization problems. In *Computer aided chemical engineering*, volume 44, pages 943–948. Elsevier, 2018.

[2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8. IEEE, 2013.

[3] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[4] José Cambronero, Sumit Gulwani, Vu Le, Daniel Perelman, Arjun Radhakrishna, Clint Simon, and Ashish Tiwari. Flashfill++: Scaling programming by example by cutting to the chase. *Proceedings of the ACM on Programming Languages*, 7(POPL):952–981, 2023.

[5] Xinyun Chen, Chen Liang, Adams Wei Yu, Dawn Song, and Denny Zhou. Compositional generalization via neural-symbolic stack machines. *Advances in Neural Information Processing Systems*, 33:1690–1701, 2020.

[6] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pages 990–998. PMLR, 2017.

[7] John K Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices*, 50(6):229–239, 2015.

[8] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[9] Alex Graves, Marc G Bellemare, Jacob Menick, Remi Munos, and Koray Kavukcuoglu. Automated curriculum learning for neural networks. In *international conference on machine learning*, pages 1311–1320. Pmlr, 2017.

[10] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

[11] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017.

[12] Joey Hong, David Dohan, Rishabh Singh, Charles Sutton, and Manzil Zaheer. Latent programmer: Discrete latent codes for program synthesis. In *International Conference on Machine Learning*, pages 4308–4318. PMLR, 2021.

[13] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

[14] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2.5-Coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

[15] Jeevana Priya Inala, Osbert Bastani, Zenna Tavares, and Armando Solar-Lezama. Synthesizing programmatic policies that inductively generalize. In *International Conference on Learning Representations*, 2020.

[16] Tim Klinger, Luke Liu, Soham Dan, Maxwell Crouse, Parikshit Ram, and Alexander Gray. Compositional program generation for systematic generalization. *arXiv preprint arXiv:2309.16467*, 2023.

[17] Donald E Knuth. *The Art of Computer Programming: Sorting and Searching, volume 3*. Addison-Wesley Professional, 1998.

[18] Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem proving. *Advances in neural information processing systems*, 35:26337–26349, 2022.

[19] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

[20] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. *ACM SIGPLAN Notices*, 53(4):436–449, 2018.

[21] Wen-Ding Li and Kevin Ellis. Is programming by example solved by llms? *arXiv preprint arXiv:2406.08316*, 2024.

[22] Guan-Ting Liu, En-Pei Hu, Pu-Jen Cheng, Hung-Yi Lee, and Shao-Hua Sun. Hierarchical programmatic reinforcement learning via learning to compose programs. In *International Conference on Machine Learning*, pages 21672–21697. PMLR, 2023.

[23] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, pages 46534–46594, 2023.

[24] Tambet Matiisen, Avital Oliver, Taco Cohen, and John Schulman. Teacher–student curriculum learning. *IEEE transactions on neural networks and learning systems*, 31(9):3732–3740, 2019.

[25] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. In *The International Conference on Learning Representations*, 2018. URL `https://openreview.net/forum?id=HkfXMz-Ab`.

[26] Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. In *International Conference on Machine Learning*, pages 4861–4870. PMLR, 2019.

[27] Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? In *International Conference on Learning Representations*, volume 2024, pages 36545–36593, 2024.

[28] Wasu T Piriyakulkij, Cassidy Langenfeld, Tuan A Le, and Kevin Ellis. Doing experiments and revising rules with natural language and probabilistic reasoning. *Advances in Neural Information Processing Systems*, 37:53102–53137, 2024.

[29] Rémy Portelas, Cédric Colas, Lilian Weng, Katja Hofmann, and Pierre-Yves Oudeyer. Automatic curriculum learning for deep rl: A short survey. *arXiv preprint arXiv:2003.04664*, 2020.

[30] Archiki Prasad, Alexander Koller, Mareike Hartmann, Peter Clark, Ashish Sabharwal, Mohit Bansal, and Tushar Khot. Adapt: As-needed decomposition and planning with language models. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pages 4226–4252, 2024.

[31] Linlu Qiu, Liwei Jiang, Ximing Lu, Melanie Sclar, Valentina Pyatkin, Chandra Bhagavatula, Bailin Wang, Yoon Kim, Yejin Choi, Nouha Dziri, et al. Phenomenal yet puzzling: Testing inductive reasoning capabilities of language models with hypothesis refinement. In *International Conference on Learning Representations*, volume 2024, pages 1452–1481, 2024.

[32] Qwen Team. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

[33] Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. Sequence level training with recurrent neural networks. In *The International Conference on Learning Representations*, 2016. URL http://arxiv.org/abs/1511.06732.

[34] Steven J Rennie, Etienne Marcheret, Youssef Mroueh, Jerret Ross, and Vaibhava Goel. Self-critical sequence training for image captioning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7008–7024, 2017.

[35] Joshua S. Rule, Steven T. Piantadosi, Andrew Cropper, Kevin Ellis, Maxwell Nye, and Joshua B. Tenenbaum. Symbolic metaprogram search improves learning efficiency and explains rule learning in humans. *Nature Communications*, 15:6847, 2024.

[36] Stuart J Russell and Devika Subramanian. Provably bounded-optimal agents. *Journal of Artificial Intelligence Research*, 2:575–609, 1994.

[37] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[38] Kensen Shi, Hanjun Dai, Wen-Ding Li, Kevin Ellis, and Charles Sutton. Lambdabeam: Neural program search with higher-order functions and lambdas. *Advances in Neural Information Processing Systems*, pages 51327–51346, 2023.

[39] Kensen Shi, Joey Hong, Yinlin Deng, Pengcheng Yin, Manzil Zaheer, and Charles Sutton. Exedec: Execution decomposition for compositional generalization in neural program synthesis. In *International Conference on Learning Representations*, volume 2024, pages 45106–45135, 2024.

[40] Richard Shin, Neel Kant, Kavi Gupta, Chris Bender, Brandon Trabucco, Rishabh Singh, and Dawn Song. Synthetic datasets for neural program synthesis. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=ryeOSnAqYm.

[41] Armando Solar-Lezama. *Program synthesis by sketching*. University of California, Berkeley, 2008.

[42] Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R. Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *Transactions on Machine Learning Research*, 2023.

[43] Dweep Trivedi, Jesse Zhang, Shao-Hua Sun, and Joseph J Lim. Learning to synthesize programs as interpretable and generalizable policies. *Advances in neural information processing systems*, 34:25146–25163, 2021.

[44] Andrew M Wells, Neil T Dantam, Anshumali Shrivastava, and Lydia E Kavraki. Learning feasibility for task and motion planning in tabletop environments. *IEEE robotics and automation letters*, 4(2):1255–1262, 2019.

[45] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, 1992.

[46] Jonas Witt, Stef Rasing, Sebastijan Dumančić, Tias Guns, and Claus-Christian Carbon. A divide-align-conquer strategy for program synthesis. *arXiv preprint arXiv:2301.03094*, 2023.

[47] Zhutian Yang, Caelan Reed Garrett, Tomás Lozano-Pérez, Leslie Kaelbling, and Dieter Fox. Sequence-based plan feasibility prediction for efficient task and motion planning. *arXiv preprint arXiv:2211.01576*, 2022.

[48] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, 2017.

[49] Janis Zenkner, Tobias Sesterhenn, and Christian Bartelt. Beyond either-or reasoning: Transduction and induction as cooperative problem-solving paradigms. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 485–502. Springer, 2026.

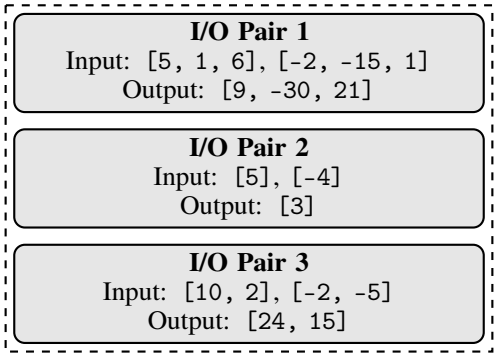
[50] Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. Planning with large language models for code generation. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=Lr8cOOtYbfL.

A Domains & DSLs

The three domains differ in program structure and state update, and therefore in how many valid decompositions a task admits. Deepcoder and Lambdabeam admit several decompositions per task, whereas Robustfill fixes the decomposition order and serves as the negative control of the evaluation.

A.1 Deepcoder

Deepcoder tasks specify transformations of integer lists: the inputs of each I/O example are one or two integer lists, possibly with scalar integers, and the output is an integer or a list. A program is a sequence of lines, each of which applies a first- or higher-order operation such as `Map`, `Filter`, `ZipWith`, `Sort`, or `Scanl1`. Its result becomes a new variable that later lines can use, and the last assigned variable is the output. Figure A.1 shows an example task, and Figure A.2 shows the DSL.



Ground Truth:
$x_0 = \texttt{INPUT}$ | $x_1 = \texttt{INPUT}$ | $x_2 = \texttt{Scanl1 (max)}\ x_0$
| $x_3 = \texttt{ZipWith (+) x1 x2}$ | $x_4 = \texttt{Map (*3)}\ x_3$

Figure A.1: Example task from the Deepcoder domain. The program takes the running maximum of the first input, adds the second input, and multiplies the result by 3.

$$
\begin{aligned}
\text{Program } P &:= \quad i_1;\ i_2;\ \ldots;\ a_1;\ a_2;\ \ldots \\
\text{Initialization } i &:= \quad v \leftarrow \texttt{INPUT} \\
\text{Assignment } a &:= \quad v \leftarrow f \mid v \leftarrow h \\
\text{First-Order Operation } f &:= \quad \texttt{Head}(l) \mid \texttt{Last}(l) \mid \texttt{Access}(n, l) \mid \texttt{Minimum}(l) \mid \texttt{Maximum}(l) \\
&\quad \mid \texttt{Sum}(l) \mid \texttt{Take}(n, l) \mid \texttt{Drop}(n, l) \mid \texttt{Reverse}(l) \mid \texttt{Sort}(l) \\
\text{Higher-Order Operation } h &:= \quad \texttt{Map}(\lambda, l) \mid \texttt{Filter}(\beta, l) \mid \texttt{Count}(\beta, l) \mid \texttt{ZipWith}(\Sigma, l, l) \\
&\quad \mid \texttt{Scanl1}(\Sigma, l) \\
\text{int} \to \text{int Lambda } \lambda &:= \quad (+1) \mid (-1) \mid (*2) \mid (/2) \mid (*(-1)) \mid (**2) \mid (*3) \mid (/3) \mid (*4) \mid (/4) \\
\text{int} \to \text{bool Lambda } \beta &:= \quad (>0) \mid (<0) \mid (\%2 == 0) \mid (\%2 == 1) \\
(\text{int}, \text{int}) \to \text{int Lambda } \Sigma &:= \quad (+) \mid (-) \mid (*) \mid (\texttt{min}) \mid (\texttt{max}) \\
\text{Integer Variable } n &:= \quad v \\
\text{List Variable } l &:= \quad v \\
\text{Variable Name } v &:= \quad x_0 \mid x_1 \mid \ldots
\end{aligned}
$$

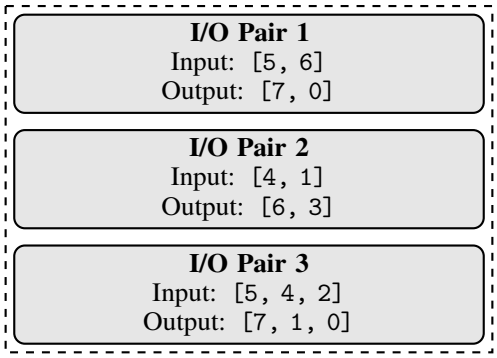
Figure A.2: DSL of the Deepcoder domain.

Since later lines can use all earlier results, the task does not fix the structure of the program, and several programs and decompositions usually produce the same outputs. Program length is the number of non-input lines, i.e., the number of decomposition steps.

For in-distribution evaluation, we train and test on programs of length 1–4; for length generalization, we train on lengths 1–4 and test on length 5. Each task has 3 I/O examples plus one hidden example, and the input lists have at most 5 elements. A task generated by sampling a program of length 5 can have an equivalent shorter program of a training length, so it would not test length generalization. We therefore enumerate all programs up to the maximum length and record all minimal-length solutions of each task. A training task needs at least one minimal-length solution of a training length, and a length-generalization test task needs all its minimal-length solutions to be of the test length, so no test task can be solved by a program of a training length.

A.2 Lambdabeam

Lambdabeam extends Deepcoder in two ways. First, lambda functions are built during synthesis: instead of selecting from a fixed set of lambdas such as `(+1)` or `(*2)`, the synthesizer chooses the operation and its argument, which is either an integer constant in $[-5, 5]$ or a program variable that holds such a constant. Second, a conditional `If` selects element-wise between two lists, depending on a condition on a third list. Apart from the `Access` operation, which Lambdabeam does not contain, the remaining operations, the state representation, and the state update are the same as in Deepcoder. Figure A.3 shows an example task, and Figure A.4 shows the rules of the DSL that differ from Deepcoder. Both extensions enlarge the set of candidate subprograms per step. A subgoal that leads



Ground Truth:
$x_0 =$ `INPUT` | $x_1 =$ `Map Subtract -2 x0` | $x_2 =$ `Scanl1 IntDivide` x_1
| $x_3 =$ `If Less 2` $x_0\ x_1\ x_2$

Figure A.3: Example task from the Lambdabeam domain. The program adds 2 to each element, computes a running integer division, and keeps the shifted value wherever the input element is less than 2.

$$
\begin{aligned}
\text{Assignment } a &:= \quad v \leftarrow f \mid v \leftarrow h \mid v \leftarrow c \\
\text{Conditional Operation } c &:= \quad \texttt{If}(\beta, l, l, l) \\
\text{int} \to \text{int Lambda } \lambda &:= \quad (+\ e) \mid (-\ e) \mid (*\ e) \mid (/\ e) \\
\text{int} \to \text{bool Lambda } \beta &:= \quad (>\ e) \mid (<\ e) \mid (\%2 == 0) \mid (\%2 == 1) \\
\text{Lambda Argument } e &:= \quad -5 \mid \ \ldots \ \mid 5 \mid n
\end{aligned}
$$

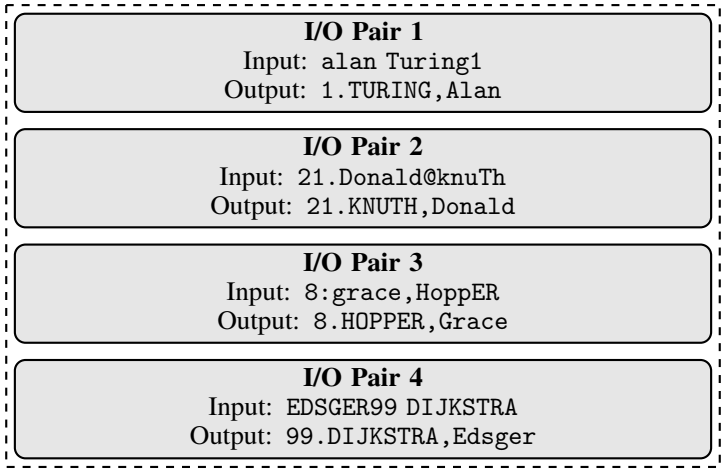
Figure A.4: Rules of the Lambdabeam DSL that differ from the Deepcoder DSL (Figure A.2) or are missing there. All other rules are the same, except that Lambdabeam has no `Access` operation.

the synthesizer to an unusual lambda or an unnecessary `If` is therefore more likely to fall outside the beam than in Deepcoder. Splits and data generation are the same as for Deepcoder.

A.3 Robustfill

Robustfill tasks map one input string to one output string with a DSL of substring, case, token-extraction, and composition operations. A program is a concatenation of expressions, each of which is applied to the original input and appends its result to the output. Figure A.5 shows an example task, and Figure A.6 shows the DSL.

The state update differs from the list domains. After each step, its output is removed from the front of the remaining target, so the residual shows what is left to produce, and the subtasks become more specified as the trajectory advances. Since each step must produce a prefix of the residual, the




```
Ground Truth: GetAll(NUMBER) | ConstStr('.')  |
Compose(ToCase(ALL_CAPS), GetToken(WORD, -1)) |
ConstStr(',') | Compose(ToCase(PROPER), GetToken(WORD, 1))
```

Figure A.5: Example task from the Robustfill domain. The program outputs the number, then the last name in capitals, and then the first name in title case.

$$
\begin{aligned}
\text{Program } P &:= \texttt{Concat}(e_1, e_2, \ldots) \\
\text{Expression } e &:= s \mid m \mid o \mid \texttt{ConstStr}(c) \\
\text{Compose } o &:= m_1(m_2) \mid m(s) \\
\text{Substring } s &:= \texttt{SubStr}(k_1, k_2) \mid \texttt{GetSpan}(r_1, i_1, b_1, r_2, i_2, b_2) \\
&\quad \mid \texttt{GetUpto}(r, i) \mid \texttt{GetFrom}(r, i) \mid \texttt{GetToken}(r, i) \\
\text{Modification } m &:= \texttt{ToCase}(a) \mid \texttt{Replace}(c_1, c_2) \mid \texttt{Trim()} \\
&\quad \mid \texttt{GetFirst}(r, i) \mid \texttt{GetAll}(r) \\
&\quad \mid \texttt{Substitute}(r, i, c) \mid \texttt{SubstituteAll}(r, c) \\
&\quad \mid \texttt{Remove}(r, i) \mid \texttt{RemoveAll}(r) \\
\text{Regex } r &:= \texttt{NUMBER} \mid \texttt{WORD} \mid \texttt{ALPHANUM} \mid \texttt{ALL_CAPS} \mid \texttt{PROPER_CASE} \\
&\quad \mid \texttt{LOWER} \mid \texttt{DIGIT} \mid \texttt{CHAR} \mid \delta \\
\text{Case } a &:= \texttt{ALL_CAPS} \mid \texttt{PROPER_CASE} \mid \texttt{LOWER} \\
\text{Position } k &:= -100 \mid -99 \mid \ldots \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots \mid 100 \\
\text{Index } i &:= -5 \mid -4 \mid \ldots \mid -1 \mid 1 \mid 2 \mid \ldots \mid 5 \\
\text{Boundary } b &:= \texttt{START} \mid \texttt{END} \\
\text{Character } c &:= A \mid \ldots \mid Z \mid a \mid \ldots \mid z \mid 0 \mid \ldots \mid 9 \mid \delta \\
\text{Delimiter } \delta &:= \texttt{\&}\ \texttt{,}\ \texttt{.}\ \ \texttt{?}\ \ \texttt{!}\ \ \texttt{@}\ \ \texttt{()}\ \ \texttt{[]}\ \ \texttt{\%}\ \ \texttt{\#}\ \ \texttt{\$}\ \ \texttt{"}\ \ \texttt{-}
\end{aligned}
$$

Figure A.6: DSL of the Robustfill domain.

decomposition order is fixed: a step whose output is not such a prefix cannot be continued. The implementation of a single step can still vary, but the sequence of subtasks is determined by the task.

Without alternative decompositions, solver feedback has nothing to choose between, so SAD should perform like supervised training in this domain. Robustfill is therefore the negative control of the evaluation.

For in-distribution evaluation, we train and test on programs of length 1–6; for length generalization, we train on lengths 1–6 and test on lengths 7–10. Each task has 4 I/O examples plus one hidden example, and the input strings have at most 20 characters. A program is sampled from the target length range such that it executes on all inputs, and its outputs form the examples. Each concatenated expression is a subprogram, and program length is the number of subprograms.

B Failure cases

This appendix shows two Deepcoder tasks on which SAD succeeds and a method that uses GT subgoals fails. In the first, the solver-blind decomposer proposes a subgoal that no single DSL operation produces; in the second, every GT subgoal is realizable, but the GT oracle still does not solve the task.

B.1 Subgoal Outside the DSL

The task in Figure B.1 maps two input lists to one output list with a GT program of four steps, and both decomposers first predict the subgoal for `x2`. No single Deepcoder operation produces the subgoal of the solver-blind decomposer. In the second example, the value `1` must become `1` at the first position and `16` at the last, which rules out `Map` and `ZipWith` of `x0` with itself. `ZipWith` with `x1` shortens the list to the length of `x1` and `Scanl1` keeps the first element, which the third example changes from `3` to `16`. The remaining list operations only select or reorder existing values, none of which is `16`, and all other operations return a single integer. The synthesizer thus receives a subtask without a consistent solution, and the solver-blind method does not solve the task. SAD predicts the GT subgoal, which the synthesizer solves with `Map(/3) x0`, and the remaining steps follow the GT program to a correct solution. This shows a kind of error that the reward of SAD penalizes directly:

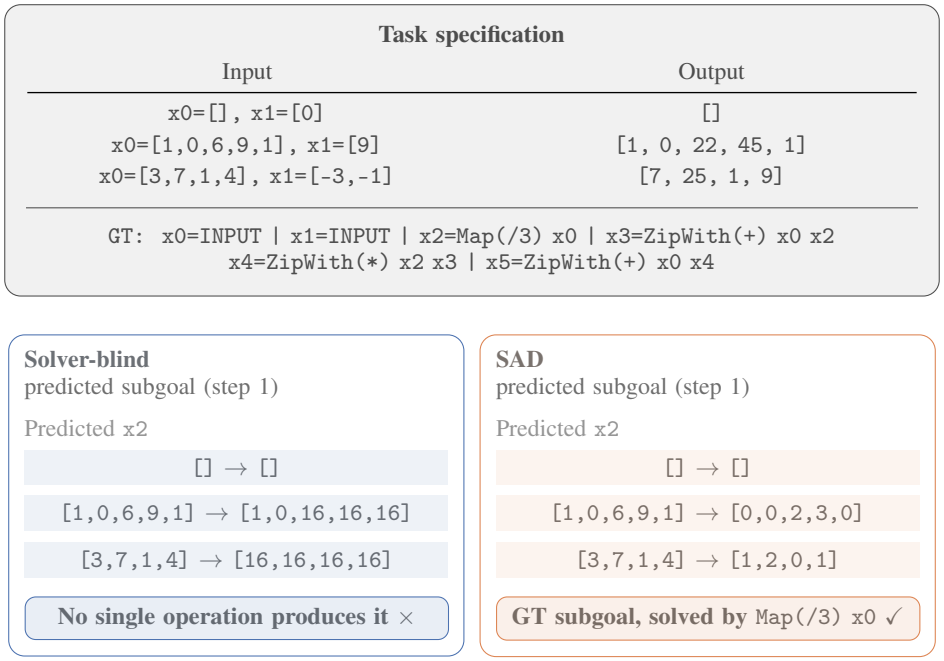


Figure B.1: First decomposition step on a Deepcoder task. No single DSL operation produces the subgoal of the solver-blind decomposer, so synthesis fails at this step. SAD predicts the GT subgoal, which the synthesizer solves with `Map(/3) x0`, and SAD solves the task.

since no subprogram is consistent with such a subgoal, the synthesizer is unlikely to assign a high probability to the GT subprogram, whereas $\mathcal{L}_{\text{sup}}$ only measures the distance of the subgoal to the GT subgoal.

B.2 Realizable GT Path That the Synthesizer Does Not Follow

The task in Figure B.2 shows a failure that remains when the decomposition is perfect. The GT oracle receives every GT subgoal at inference, and all of them are realizable in the DSL, yet neither the oracle nor the solver-blind method solves the task. The failure therefore lies in synthesis along the GT path, which takes five steps through `Scanl1(-)` and three `ZipWith(-)` operations. SAD solves the task in four steps with a different program, which multiplies `x0` element-wise with the halved `x1`, negates the result, and takes the element-wise maximum with `x1`. This path avoids `Scanl1` and the chain of subtractions, which is possibly where the synthesizer fails on the GT path. The example matches the aggregate analysis in Appendix D.5, where fewer than 2% of the SAD-only solutions on Deepcoder match the GT program and their primitive usage differs from GT; a single task, however, does not show which primitives are hard for the synthesizer in general. Training decomposers to find such alternative decompositions deliberately is left for future work.

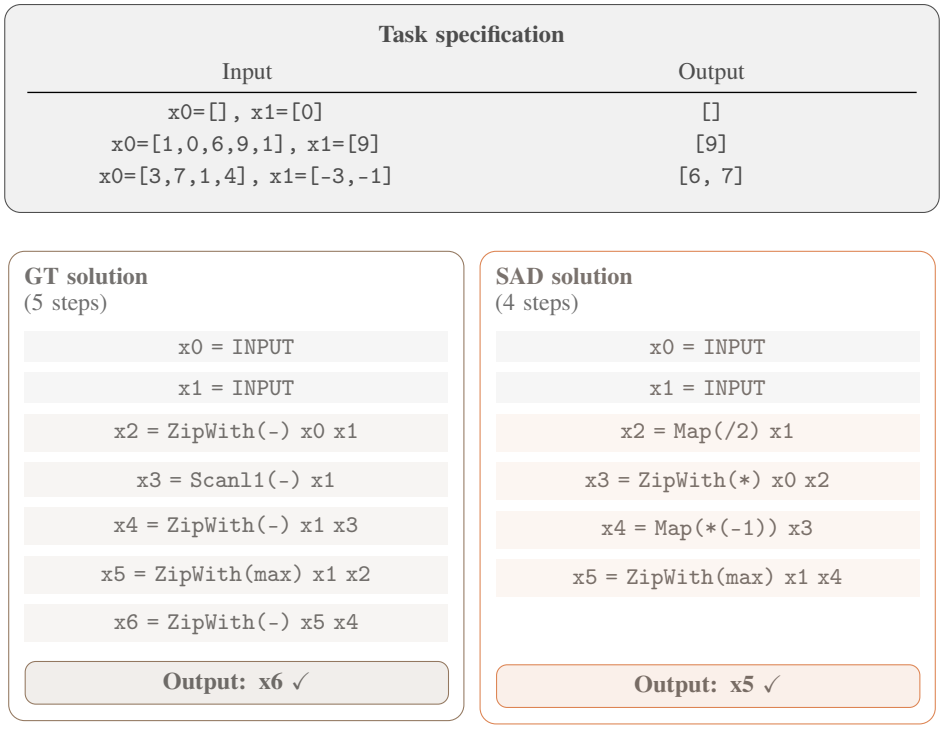


Figure B.2: GT program (5 steps) and SAD solution (4 steps) for a Deepcoder task; both satisfy the I/O examples. With the GT subgoals, the GT oracle does not solve the task, and neither does the solver-blind method. SAD solves it with a different program that avoids `Scanl1` and the chain of `ZipWith(-)` operations.

C Training and Inference Setup

SAD and the solver-blind baseline share architecture, hyperparameters, training data, synthesizer weights, and search, and differ only in the training signal of the decomposer.

C.1 Training Setup

Synthesizer and decomposer are 3-layer transformers with embedding dimension 512 and hidden dimension 1024. The synthesizer is trained with learning rate 2×10^{-4} and the decomposer with

1×10^{-4}, both with square-root decay after 16k linear warmup steps and for 500k steps in total. The batch size is 128 for the list domains and 64 for Robustfill, to fit into GPU memory.

Synthesizer training. All synthesizers are trained with teacher forcing to generate the next GT subprogram p_t^* and are frozen afterwards, so that the synthesizer stays fixed during decomposer training. They differ in the input they receive during training. The synthesizer of Exedec receives the GT subtask $(I^{(t)}, O_t^*)$ and thus learns to implement a decomposition step that is already given. The synthesizer of SAD and the solver-blind baseline receives the full current task $s_t = (I^{(t)}, O^{(t)})$, without a subgoal, and thus has to infer which subtask comes next as well as how to implement it. This makes it informative also on subgoals that leave the GT trajectory, which are the subgoals that the SCST reward has to score (Section 4.1). At inference, all three methods give the synthesizer the subtask $(I^{(t)}, O_{\text{pred}}^{(t)})$ induced by the predicted subgoal (Algorithm 1), and the SCST reward uses the same input.

Decomposer training and compared methods. Exedec and the solver-blind baseline train the decomposer with $\mathcal{L}_{\text{sup}}$ only, with identical settings. SAD trains it with the full objective of Section 4.2 from the start of training, without a supervised warm-up phase or a pretrained checkpoint. The three methods thus combine the two components as follows:

- **Exedec**: synthesizer trained on GT subtasks, decomposer trained with $\mathcal{L}_{\text{sup}}$.
- **Solver-blind**: synthesizer trained on the full current task, decomposer trained with $\mathcal{L}_{\text{sup}}$.
- **SAD**: synthesizer trained on the full current task, decomposer trained with $\mathcal{L}_{\text{sup}}$ and SCST.

Each adjacent pair differs in one component. SAD and the solver-blind baseline differ only in the training signal of the decomposer, so their comparison shows the effect of solver-aware training, and the conclusions in Section 5.2 rest on it. Exedec and the solver-blind baseline differ only in the synthesizer, so their comparison shows the effect of training the synthesizer on the full current task. Both SAD and the solver-blind baseline are trained on single decomposition steps; the larger gains of SAD on longer programs (Appendix D.2) thus arise without a multi-step objective.

Compute. Training the synthesizer and training the decomposer with $\mathcal{L}_{\text{sup}}$ each take 5–6 hours per seed on a single NVIDIA RTX PRO 6000. SAD training takes about 30% longer than supervised decomposer training, because the reward needs two additional forward passes of the frozen synthesizer per training state, one for the sampled and one for the greedy subgoal. Inference cost is the same for all methods.

C.2 Beam Search

Inference follows the beam search of Exedec [39]. A single beam of $k = 10$ candidate trajectories runs through the whole synthesis process and is extended in place at every model call, without restarting at each decomposition or synthesis step. Each candidate is a partial solution, that is, a sequence of predicted subgoals and the subprograms synthesized for them. Candidates are ranked by the sum of the log-probabilities of the decomposer and the synthesizer, and they are pruned if they fail to parse or execute, are functionally redundant with a higher-ranked candidate, or exceed domain-specific computational limits. The search stops after at most $T = 10$ steps for the list domains and $T = 20$ for Robustfill. The GT oracle uses the same beam search with the GT subgoals in place of the predicted ones.

D Additional Experiments and Analyses with Transformers

D.1 Statistical Significance Tests

We compare the task accuracies of the three methods with two-sided paired t-tests over the 5 seeds, which gives 4 degrees of freedom, at a significance level of 5%. Table D.1 reports the results. SAD is significantly better than the solver-blind baseline in all four settings of the list domains, and the two methods do not differ significantly on Robustfill. Against Exedec, SAD is significantly better in the list domains except on Deepcoder in-distribution, while on Robustfill in-distribution, Exedec is significantly better than both other methods. However, the effect is minimal there.

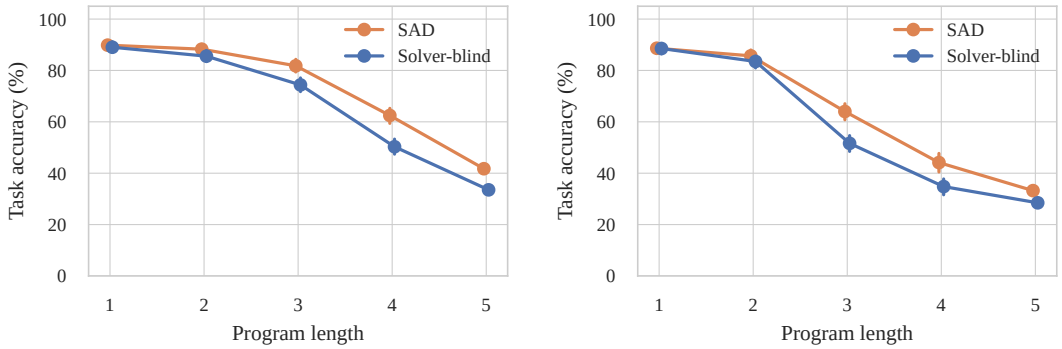
Table D.1: Paired t-tests on task accuracy over 5 seeds, reported as p-value (t-statistic). Bold p-values are significant at the 5% level.

Domain	Setting	SAD vs. solver-blind	Solver-blind vs. Exedec	SAD vs. Exedec
Deepcoder	In-distribution	$\mathbf{7.68 \times 10^{-3}}$ (5.00)	5.18×10^{-2} (−2.74)	1.77×10^{-1} (1.63)
	Length gen.	$\mathbf{6.88 \times 10^{-3}}$ (5.12)	$\mathbf{3.24 \times 10^{-2}}$ (3.22)	$\mathbf{2.18 \times 10^{-5}}$ (22.84)
Lambdabeam	In-distribution	$\mathbf{2.60 \times 10^{-2}}$ (3.45)	$\mathbf{2.23 \times 10^{-2}}$ (3.62)	$\mathbf{1.34 \times 10^{-4}}$ (7.98)
	Length gen.	$\mathbf{7.65 \times 10^{-3}}$ (4.97)	$\mathbf{5.76 \times 10^{-4}}$ (9.94)	$\mathbf{5.35 \times 10^{-4}}$ (10.13)
Robustfill	In-distribution	8.76×10^{-1} (0.17)	$\mathbf{3.57 \times 10^{-2}}$ (−3.12)	$\mathbf{1.31 \times 10^{-2}}$ (−4.26)
	Length gen.	6.14×10^{-1} (−0.55)	2.00×10^{-1} (−1.53)	9.53×10^{-2} (−2.18)

D.2 Performance Gap Grows with Program Length

In both list domains, the advantage of SAD over the solver-blind baseline grows with the length of the GT program (Figure D.1), although both decomposers are trained on single decomposition steps. This indicates that the benefit of solver-aware subgoals accumulates over the steps of a task. The state update of the list domains is a likely reason: a subgoal that the synthesizer cannot act on uses up one of the T steps and adds its result to the state on which all later steps are conditioned, so each such subgoal also affects the remaining steps, and longer programs offer more occasions for it.

On Robustfill (Figure D.2), accuracy decreases more slowly with program length for all methods, and SAD and the solver-blind baseline perform equally at every length, as predicted for a domain without decomposition ambiguity. Here, each step removes its output from the front of the remaining target



(a) Deepcoder (b) Lambdabeam

Figure D.1: Task accuracy over the length of the GT program in the list domains. The advantage of SAD over the solver-blind baseline grows with program length, although both decomposers are trained on single steps.

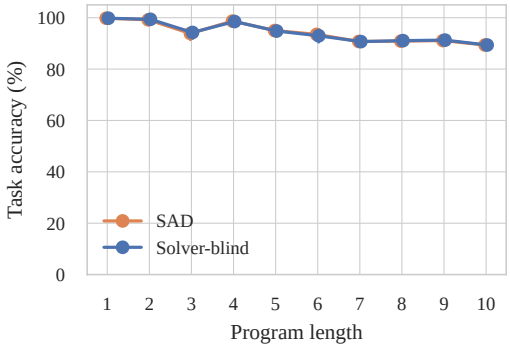


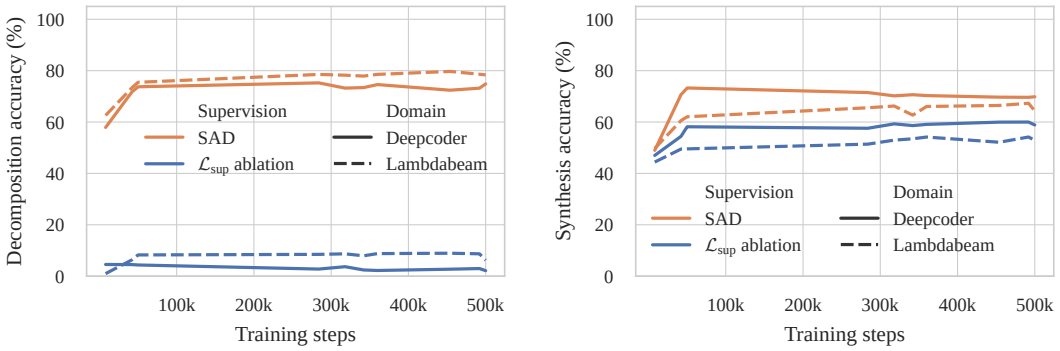
Figure D.2: Task accuracy over the length of the GT program on Robustfill. Accuracy decreases slowly for all methods, and SAD and the solver-blind baseline perform equally at every length.

string, so the residual shows what is left to produce, and the task likely becomes easier with each step. In the list domains, by contrast, the target stays fixed while intermediate results accumulate as inputs, so the remaining target does not show which operations are still needed. This possibly also explains why accuracy decreases more steeply with program length in the list domains for all methods.

D.3 $\mathcal{L}_{\text{sup}}$ Ablation

To measure the contribution of $\mathcal{L}_{\text{sup}}$, we train the decomposer of SAD without it, with the objective $\mathcal{L} = \mathcal{L}_{\text{RL}} - \lambda H(\pi_\theta)$ and $\lambda = 0.01$. With $\mathcal{L}_{\text{sup}}$, the training decomposition accuracy reaches 75–80% within the first 50k steps and stays there (Figure D.3a). Without it, decomposition accuracy drops to 2–9% early in training and does not recover. The reward still improves during this collapse: the training synthesis accuracy rises to 58–60% on Deepcoder and 53–54% on Lambdabeam, against 70–73% and 65–67% with $\mathcal{L}_{\text{sup}}$ (Figure D.3b). On the test tasks, however, the ablated model, which we evaluated end to end for a single seed, solves fewer than 20% of the tasks.

This shows that without $\mathcal{L}_{\text{sup}}$, the decomposer finds degenerate subgoals from which the synthesizer still recovers part of the GT subprograms on the training tasks, and that this does not transfer to new tasks. It indicates that solver feedback is useful only within a range of subgoals that resemble the subgoals on which the synthesizer was trained. $\mathcal{L}_{\text{sup}}$ keeps the decomposer within this range, and $\mathcal{L}_{\text{RL}}$ selects within it the subgoals that the synthesizer solves more reliably. Together with the accuracy paradox (Section 5.2), this shows that neither signal suffices alone: with $\mathcal{L}_{\text{sup}}$ only, the decomposer reproduces GT subgoals that the synthesizer does not always act on, and with $\mathcal{L}_{\text{RL}}$ only, it leaves the range in which the feedback of the synthesizer is informative.



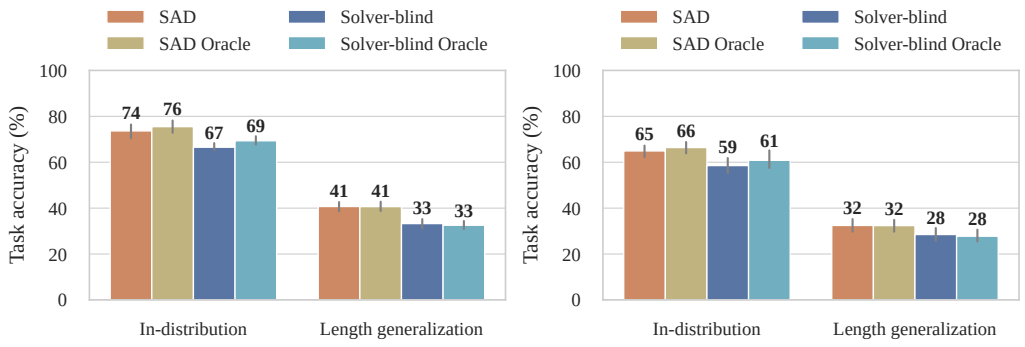
(a) Training decomposition accuracy

(b) Training synthesis accuracy

Figure D.3: Training decomposition and synthesis accuracy of SAD with and without $\mathcal{L}_{\text{sup}}$ in the list domains. Without $\mathcal{L}_{\text{sup}}$, decomposition accuracy drops to 2–9%, while synthesis accuracy on the training tasks still rises to a lower level than with $\mathcal{L}_{\text{sup}}$.

D.4 Beam Oracle

The beam oracle tests whether the gap between SAD and the solver-blind baseline would close if the decomposers ranked the GT subgoals higher. It selects the GT decomposition whenever it appears in the beam and otherwise keeps the original ranking, so it separates whether a decomposer generates the GT decomposition from how it ranks it. For both methods and both list domains, the oracle improves task accuracy by less than 2 percentage points (Figure D.4), and with the oracle, the solver-blind baseline still stays below SAD without it. This shows that ranking the GT subgoals higher does not close the gap. The gap therefore lies in the decompositions that each decomposer generates, and SAD likely generates decompositions that are rare in the beam of the solver-blind decomposer. The oracle covers only this one kind of reranking: since the GT decomposition is not always the best one for the synthesizer (Section 5.2), a reranking that selects any successful candidate could give different results.



(a) Deepcoder (b) Lambdabeam

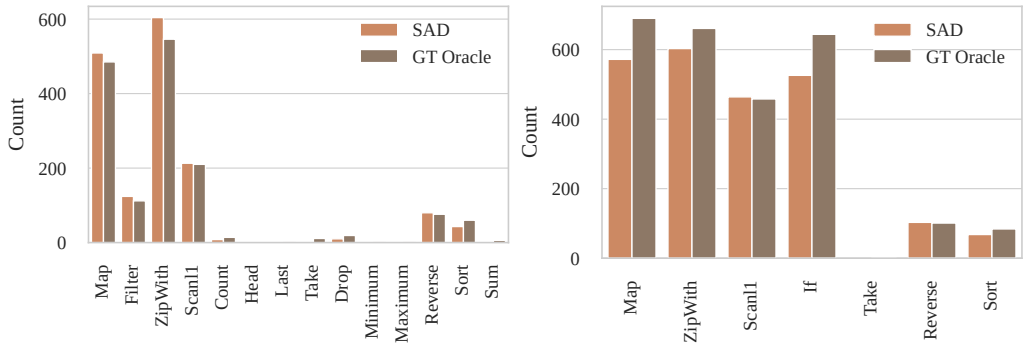
Figure D.4: Task accuracy with and without the beam oracle, which selects the GT decomposition whenever it appears in the beam. The oracle adds less than 2 percentage points for both methods, and the solver-blind baseline with the oracle stays below SAD without it.

D.5 Structure of SAD-only Solutions

On Lambdabeam, our program-parsing step fails on about 200 of the 2×1000 test tasks per seed due to an unsupported serialization of lambda arguments. Whether a task is excluded depends only on the task, so SAD and the GT oracle are compared on an identical task set. The excluded tasks are not systematically easier or harder: on the remaining tasks, SAD solves 48.7% and the oracle 70.4%, against 48.7% and 70.5% on the full test sets. Task accuracies in Figures 3b and Table D.1 are computed on all tasks and are unaffected.

We analyze the tasks that SAD solves and the GT oracle does not (Section 5.2) and compare the solutions of SAD with the GT programs of the same tasks in overlap, length, and primitive usage. Fewer than 2% of these solutions match the GT program on Deepcoder, and none does on Lambdabeam. On Deepcoder, 47.5% of the solutions have the same length as the GT program, 28.2% are longer, and 24.4% are shorter; on Lambdabeam, 56.8% are shorter and 17.1% are longer. This shows that SAD solves these tasks with different programs, of similar length on Deepcoder and mostly shorter on Lambdabeam. The shorter programs on Lambdabeam possibly reflect its larger set of candidate subprograms per step, which gives more room for paths that need fewer steps than the GT program.

The primitive usage shifts accordingly (Figure D.5). On Deepcoder, the solutions of SAD use `ZipWith`, `Map`, and `Filter` more often and `Sort`, `Drop`, `Take`, `Count`, `Sum`, and `Head` less often than the GT programs. On Lambdabeam, they use almost all operations less often, except `Reverse` and `Scanl1`; since these solutions are mostly shorter, part of this decrease likely follows from their length.



(a) Deepcoder (b) Lambdabeam

Figure D.5: DSL primitive usage in the solutions of SAD on tasks that the GT oracle does not solve, compared with the GT programs of the same tasks.

Together with the low overlap, this indicates that SAD reaches the target through other primitives than the GT programs, although its reward contains no term for deviating from GT.

D.6 Why a Reward Against the GT Program Does Not Pull Toward GT Subgoals

The reward $R(O) = -\mathrm{CE}(\mathrm{Synth}(I^{(t)}, O), p_t^*)$ contains the GT subprogram p_t^*, so $\mathcal{L}_{\mathrm{RL}}$ could in principle push the decomposer back toward the GT subgoals. To see why it does not, it helps to separate the GT *program*, which the reward contains, from the GT *decomposition*, which it does not. The aim of SAD is a decomposer that proposes subgoals the synthesizer can solve, also where they leave the GT trajectory. A subgoal that differs from the GT subgoal can still lead the synthesizer to p_t^*, and the reward favors it whenever the synthesizer recovers p_t^* from it more easily.

Why the reward uses the GT program Training needs a single verifiable target for the prediction of the synthesizer, and the GT program provides it by fixing the correct semantics. Other correct programs may exist, but finding them during training would require deciding semantic equivalence between arbitrary programs, which is computationally expensive in this setting. The GT program is therefore the practical target.

What the advantage compares The reward measures how well the frozen synthesizer recovers the correct subprogram from a proposed subgoal: p_t^* fixes *what* to synthesize, and the subgoal determines *how* easy this is. SCST optimizes the advantage $A = R(O^s) - R(O^g)$ and not the absolute reward. If the sampled and the greedy subgoal lead the synthesizer to the same program, the shared target does not separate them, and A only compares how confidently the synthesizer generates this program from each subgoal. If they lead to different programs, A also favors the subgoal under which the synthesizer is closer to p_t^*. The reward thus prefers the semantics of the GT program, but not the GT path to it.

Evidence If p_t^* pulled the decomposer toward the GT subgoals, decomposition accuracy would rise. Instead, SAD reaches a lower decomposition accuracy than the solver-blind baseline in every setting, together with a higher synthesis and task accuracy (Table 1). With $\mathcal{L}_{\mathrm{RL}}$ alone, decomposition accuracy drops to 2–9% and does not approach the GT subgoals (Appendix D.3). Finally, on the tasks that only SAD solves, its programs rarely match the GT program and use different primitives (Appendix D.5). This shows that the preference of the synthesizer for subgoals it can solve dominates the training signal. That SAD finds programs other than the GT program is a result of this preference; using such programs as additional supervision targets is left to future work.

E Details of the LLM Experiments

This appendix gives the setup and results of the off-the-shelf LLM baselines and the training details of SAD with Llama-3.1-8B in Section 6. All LLMs decode with temperature 0 and generate one sample per prompt, and each count below comes from a single run.

E.1 Off-the-Shelf LLM Baselines

We evaluate three open-weight LLMs without fine-tuning: Llama-3.1-8B [8], Qwen2.5-14B-Instruct [32], and Qwen2.5-Coder-14B-Instruct [14]. Each model is evaluated with direct synthesis and with decomposition, where the same model predicts each subgoal and then synthesizes the subprogram for the induced subtask in the Exedec loop with at most 5 steps. As in Section 6, we use the 200-task Deepcoder LLM benchmark of Exedec [39], which is smaller and easier than the 1000-task benchmark of the Transformer experiments, and evaluate in-distribution only. We use the prompts of Exedec, which contain the I/O specification, the list of DSL primitives, and four few-shot demonstrations. Programs are written in the Pythonic Deepcoder DSL of Exedec, in which each primitive except `Scanl1` becomes an equivalent Python one-liner; for example, `x0 = INPUT | x1 = Map (*2) x0` becomes

```
def program(x0):
    x1 = [x * 2 for x in x0]
    return x1
```

Table E.1 reports the results. No model solves more than 10% of the tasks. Decomposition increases the number of solved tasks for Llama-3.1-8B from 5 to 15 and for Qwen2.5-Coder-14B-Instruct

Table E.1: Off-the-shelf LLMs on the 200-task Deepcoder LLM benchmark (in-distribution). Tasks solved out of 200.

Method	Llama-3.1-8B	Qwen2.5-14B-Instruct	Qwen2.5-Coder-14B-Instruct
Direct synthesis	5 (2.5%)	14 (7.0%)	15 (7.5%)
With decomposition	15 (7.5%)	12 (6.0%)	20 (10.0%)

from 15 to 20, and decreases it for Qwen2.5-14B-Instruct from 14 to 12. This indicates that without fine-tuning on the DSL, neither direct synthesis nor decomposition reaches a useful accuracy, which is why the experiment in Section 6 starts from fine-tuned adapters.

E.2 SAD with Llama-3.1-8B

Decomposer and synthesizer are two LoRA adapters [13] on Llama-3.1-8B [8], with rank 32, scaling factor $\alpha = 64$, and dropout 0.05 on all attention and MLP projections (`q`, `k`, `v`, `o`, `gate`, `up`, `down`). We train them with AdamW at learning rate 10^{-4}, a cosine schedule with 100 warm-up steps, no weight decay, gradient clipping at 1.0, and an effective batch size of 16 (4 per device with 4 gradient-accumulation steps). Both adapters are first fine-tuned on Deepcoder for 1000 steps, after which the synthesizer is frozen. From the same decomposer checkpoint, we then compare three conditions:

- **Starting checkpoint**: the decomposer after the 1000 steps, without further training.
- **SFT**: 250 further steps with $\mathcal{L}_{\text{sup}}$ only, which is the solver-blind control at equal budget.
- **SAD**: 250 further steps with the full objective of Section 4.2, in which the frozen synthesizer provides the reward, the greedy subgoal serves as the SCST baseline, and $\mathcal{L}_{\text{sup}}$ and the entropy term with $\lambda = 0.01$ are kept.

The fine-tuned models use the same Pythonic DSL and prompt as the off-the-shelf baselines, including the four few-shot demonstrations. We evaluate on the 200 tasks of the in-distribution split with at most 5 decomposition steps and a single training seed; Table 2 in Section 6.2 reports the results.

F Zero-Shot Transfer to the Rule et al. Benchmark

The list-function benchmark of Rule et al. [35], which is part of BIG-bench [42], consists of 250 tasks that map integer lists to integer lists and were designed to study how humans learn rules, not generated from a synthesis DSL. This appendix describes how we select the tasks for the zero-shot transfer in Section 5.2 and reports the full result.

Setup. We use the Lambdabeam-trained SAD and solver-blind models without retraining. Each task of the benchmark contains several I/O examples, of which we always use the first four. As in the Lambdabeam tasks, the first three examples form the specification, and the fourth is the held-out example of the leave-one-out protocol. To select the tasks that the Lambdabeam DSL can express, we run enumerative search over Lambdabeam programs on the three specification examples, with the same settings as during inference on Lambdabeam, so that programs can have up to $T = 10$ steps, and keep a task if the search finds a program. Several tasks of the benchmark are solved by the same program on different inputs; we keep only one of them, so that each selected task requires a different program. This leaves 129 of the 250 tasks. Metric, beam size, and step limit are the same as for Lambdabeam (Section 5.1), and results are mean $\pm$ standard deviation over 5 seeds.

Results. SAD solves $42.25\% \pm 7.13\%$ of the tasks, against $32.95\% \pm 3.49\%$ for the solver-blind baseline, a gain of 9.3 percentage points or 28.2% relative. This shows that the advantage of SAD holds on tasks from an independent source that neither model was trained on, and it indicates that the subgoals learned against the synthesizer are not specific to the task generator of Lambdabeam. The larger standard deviation of SAD likely reflects the small number of tasks and the shift between the task distributions.

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 precisely state the paper's contributions and scope: SAD is evaluated exclusively in PBE with a fixed bounded synthesizer, empirical claims are scoped accordingly, and the accuracy paradox, Robustfill null result, and oracle findings are all accurately represented in both the abstract and introduction.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

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

   Answer: [Yes]

   Justification: Limitations are discussed in the section 7

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

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 makes no theoretical claims requiring proof; contributions are empirical.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

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: All architectural details, hyperparameters, training procedures, data splits, evaluation protocols, and inference configurations are fully specified in Appendix C. Code is released.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

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: [Yes]

   Justification: Code is released as noted in the paper and is provided as supplementary material.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

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: Training and evaluation details – including data splits, hyperparameters, batch sizes, learning rates, beam size, step limits, and the controlled setting ensuring all methods share identical architectures and synthesizer weights – are fully specified in Appendix C.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

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: All results are reported over 5 random seeds with mean and standard deviation. Paired t-tests at the 5% significance level are used for the main findings; full p-values and t-statistics are reported in Appendix D.1.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

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: [Yes]

   Justification: Training hardware (NVIDIA RTX PRO 6000), wall-clock runtimes, and the additional compute cost of SAD relative to solver-blind training are reported in Appendix C.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the 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 research conforms with the NeurIPS Code of Ethics. The work involves no human subjects, no sensitive data, and no applications with direct potential for misuse.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

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

    Answer: [N/A]

    Justification: The paper presents foundational research on decomposition training in program synthesis. There is no direct path to negative societal impact from this work.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

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 poses no risks of misuse; it releases no pretrained language models, scraped datasets, or high-risk assets.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

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: [Yes]

    Justification: All domains and baselines are properly cited. No proprietary datasets or code are used. Our code is built upon Exedec and the Lambdabeam DSL which are available under Apache 2.0 license.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.

- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

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 SAD training framework, code and associated benchmarks are released with full documentation.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

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 paper involves no crowdsourcing or human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

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: The paper involves no human subjects research.

    Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

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: [N/A]

    Justification: LLMs are not used as a component of the core methodology. The core methodology is model-agnostic and relies on models that are trained from scratch on the respective PBE domains. LLM experiments are additional results. The models used, their fine-tuning and sampling are described in the paper and appendix.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.