A Programming-by-example

PBE is a specialized form of program synthesis in which the desired computation is specified through concrete I/O examples rather than formal descriptions or explicit code [13]. Formally, a PBE task is defined by a set of n input-output pairs:

$$\mathcal{E} = \{(I_1, O_1), (I_2, O_2), \ldots, (I_n, O_n)\}$$

where each pair (I_i, O_i) represents a single demonstration of the intended transformation. The goal of any PBE solver – regardless of paradigm – is to discover a latent function p consistent with these examples:

$$\forall (I_i, O_i) \in \mathcal{E}, \quad p(I_i) = O_i$$

What differs across paradigms is what p represents and how it is found.

In *inductive* PBE, p is an explicit, symbolic program that can be inspected, reused, and applied to any future input. To make the search for p tractable, inductive approaches frequently constrain the solution space using a DSL: a restricted set of primitive functions, identifiers, and constants tailored to a specific problem domain (e.g., list manipulation, string editing). The set of all valid programs expressible in the DSL is denoted $\mathcal{P}$, and the synthesizer searches $\mathcal{P}$ for a program consistent with $\mathcal{E}$. The expressivity of the DSL directly governs the difficulty of tasks that can be solved but also of search – composing a program of length 5 from a DSL of 20 already yields a search space exceeding three million candidates, growing exponentially with program length.

In *transductive* PBE, p is never explicitly constructed. Instead, the latent function is implicitly embodied in the solver, which takes as input both the specifications $\mathcal{E}$ *and* the test input I_{test}, directly predicting the corresponding output O_{test} by pattern-matching across the provided demonstrations. This sidesteps the need for a DSL entirely, at the cost of producing no reusable or inspectable artifact. Moreover, this contrasts with inductive approaches, which take only $\mathcal{E}$ as input and produce a program that can subsequently be applied to any I_{test}.

B Domains

B.1 RobustFill Domain

The RobustFill domain focuses on transforming input strings into output strings using an DSL composed of operations such as substring extraction, modification, and composition. Each example consists of a single input string, and the corresponding output is also a single string. [8]

DSL Operations and Program Structure Figure B.1 shows all DSL operations used in this work. In the string manipulation domain, programs are structured as concatenations of subprograms, with the exception of the `Compose` operation that permits nesting of depth two. An example of such a program can be seen in Figure B.2. For a given task specified as $\{(I_i, O_i)\}$, the decomposition procedure maintains an updated specification after each predicted subprogram. In this domain, the inputs $\{I_i\}$ remain unchanged across all steps, while the outputs are shortened by removing the portion of the string already explained by the current subprogram. Concretely, if the predicted subprogram applied to the input yields $\{\hat{O}_i\}$, and the current target output is $\{O_i\}$, then the new state becomes $\{(I_i, O_i - \hat{O}_i)\}$. For example, suppose the input is the string “foobar" and the ground-truth program is composed of two subprograms: one that extracts “foo" and another that extracts “bar". The target output “foobar" is first updated to “bar" after predicting the first subprogram which yields “foo", and then to the empty string after predicting the second. In this way, the residual output always represents the part of the task that still needs to be solved.

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

Fig. B.1: String manipulation primitives.

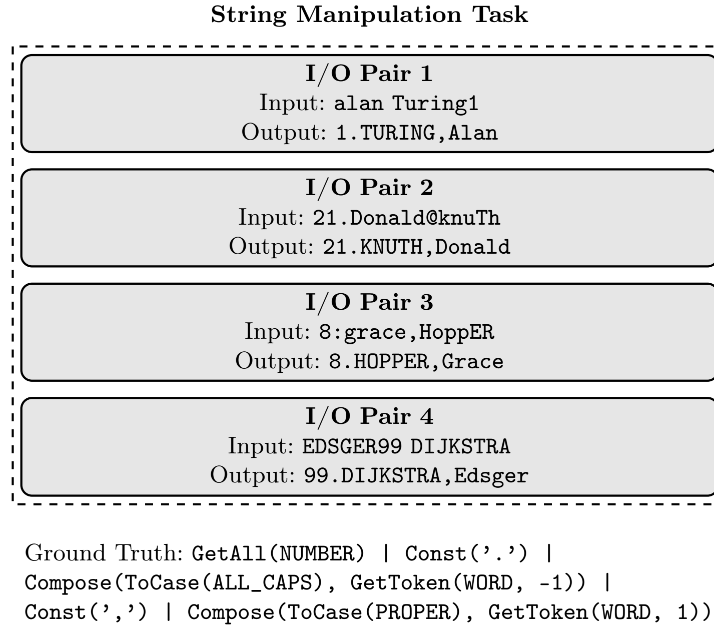


Fig. B.2: Exemplary task from the string manipulation domain. The task rearranges the input string so it starts with the number, followed by the last name in caps, and the first name in title case.

A key characteristic of this domain is how tasks are generated: subprograms are sampled at random and appended to form a complete program, which is then applied to a randomly chosen input. As a result, each task is inherently defined by the sequence of subprograms used to construct it, and therefore by the corresponding sequence of subtasks. This means that the decomposition pattern of a task is fixed or allows at most a very limited number of alternatives. Consider for instance the program `GetAll(NUMBER) | Const('.') | GetToken(WORD, -1)`. Here, the task requires first extracting all numbers from the input, then inserting a dot, and finally appending the last word yielding "1.Turning" for the input string "alan Turing1". There is no realistic way to permute or substitute the order of these subtasks: if the second and third operations were swapped, or if a different sequence of operations were attempted, the output would no longer match the target. The only conceivable alternative would be to decompose the last step in an artificially fine-grained way, for example, by producing the last word character by character rather than as a whole. Such alternatives, however, are unnatural and not consistent with how tasks are generated. Thus, while the syntactic realization of each subprogram may vary, the decomposition structure itself is essentially predetermined. The only softening of this rigidity is provided by the `Compose` operation, which permits limited nesting and thus introduces a small degree of flexibility.

Benchmark Creation In the string manipulation domain, programs consist of concatenated subprograms, with program length defined as the number of subprograms [31]. To evaluate compositional generalization, five benchmarks are used that test different aspects of a model's ability to generalize beyond the training distribution. These tasks can be grouped into three overarching themes: length generalization, mixing and matching concepts, and applying general principles.

- **Length Generalization:** This category measures whether a model can produce longer programs than those seen during training. Training programs have lengths 1–6, while test programs are longer, with lengths 7–10.
- **Compose Different Concepts:** This task evaluates whether a model can combine concepts in ways not seen during training. DSL operations are partitioned into "substring" and "non-substring" groups (excluding `Compose`). Training tasks contain operations from only one group, while test tasks combine operations from both groups. Program lengths range from 2–6 for both training and testing. This setup tests the model's ability to generalize to novel combinations of learned operations.
- **Switch Concept Order:** Here, the ability of a model to recombine concepts in sequences not seen during training is assessed. Training programs always place substring operations before non-substring operations, while test programs reverse this order. Program lengths for both training and testing are 2–6.
- **Compose New Operation:** This category examines the ability to integrate a new, previously isolated operation into larger compositions. One quarter of training tasks consist of single-op `Compose` programs, while the remaining training tasks (lengths 2–6) exclude `Compose`. Test programs (lengths 2–6) include `Compose`. The goal is to test whether a model can leverage knowledge of an isolated operation and incorporate it into more complex compositions.
- **Add Operation Functionality:** This task tests whether a model can extend its understanding of an operation by inferring new functionality from context or analogies with other operations. Training tasks (lengths 1–6) omit certain behaviors of substring operations within `Compose`, while test programs include these behaviors. This reflects scenarios such as using updated library functions with new parameters that can be inferred from analogous existing operations.

B.2 DeepCoder Domain

The DeepCoder domain [3] involves reasoning over integer lists using a DSL that supports both first-order and higher-order functions, such as `Map` and `Filter`. Tasks may include multiple input variables, each representing either integers or lists of integers, while the output is a single integer or a single integer list. Intermediate task specifications are derived by executing the current partial program on the input variables. The resulting execution output then serves as an input variable for the subsequent synthesis step. Contrary to the string manipulation

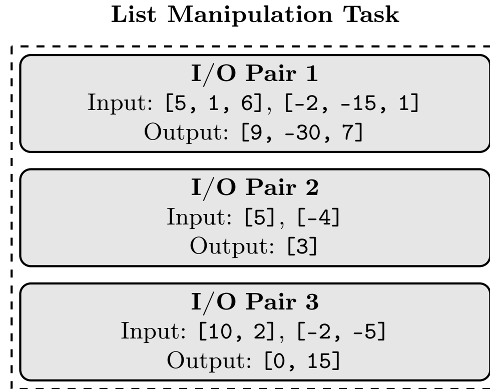
$$
\begin{aligned}
\text{Program } P &:= i_1;\ i_2;\ \ldots;\ a_1;\ a_2;\ \ldots \\
\text{Initialization } i &:= v \leftarrow \texttt{INPUT} \\
\text{Assignment } a &:= v \leftarrow f \mid v \leftarrow h \\
\text{First-Order Operation } f &:= \texttt{Head}(l) \mid \texttt{Last}(l) \mid \texttt{Access}(n, l) \mid \texttt{Minimum}(l) \mid \texttt{Maximum}(l) \\
&\quad \mid \texttt{Sum}(l) \mid \texttt{Take}(n, l) \mid \texttt{Drop}(n, l) \mid \texttt{Reverse}(l) \mid \texttt{Sort}(l) \\
\text{Higher-Order Operation } h &:= \texttt{Map}(\lambda, l) \mid \texttt{Filter}(\beta, l) \mid \texttt{Count}(\beta, l) \mid \texttt{Zip}(\Sigma, l, l) \\
&\quad \mid \texttt{Scanl1}(\Sigma, l) \\
\text{int} \rightarrow \text{int Lambda } \lambda &:= (+1) \mid (-1) \mid (*2) \mid (/2) \mid (*(-1)) \mid (**2) \mid (*3) \mid (/3) \mid (*4) \mid (/4) \\
\text{int} \rightarrow \text{bool Lambda } \beta &:= (>0) \mid (<0) \mid (\%2 == 0) \mid (\%2 == 1) \\
(\text{int}, \text{int}) \rightarrow \text{int Lambda } \Sigma &:= (+) \mid (-) \mid (*) \mid (\texttt{min}) \mid (\texttt{max}) \\
\text{Integer Variable } n &:= v \\
\text{List Variable } l &:= v \\
\text{Variable Name } v &:= x_1 \mid x_2 \mid \ldots
\end{aligned}
$$

Fig. B.3: First and Higher-Order Functions contained in the DSL for the DeepCoder domain.

domain, intermediate program states do not directly capture what is left to do. This information must be derived by comparing the current input variable and the overall target output.

DSL Operations and Program Structure Figure B.3 shows all DSL operations used in the list domain. They are the same as used in the original ExeDec study [31]. Programs are constructed sequentially, with each line depending on the outputs of preceding expressions and the initial output variable (Figure B.4). This structure mirrors human coding practices and enables opportunities for intermediate error correction. The domain's design results in a larger combinatorial space and supports more expressive decomposition strategies, thereby increasing the complexity of the synthesis process. A real example of a final synthesized program for a representative task is shown in Appendix B.2.

Benchmark Creation In the list manipulation domain, programs are structured line-by-line, with task length defined by the number of non-input lines [31]. In the DeepCoder list domain, each non-input line of a program is treated as a subprogram, so program length corresponds to the number of computational steps. Programs are evaluated using I/O examples, with at most two input lists. Five compositional generalization tasks are used that assess a model's ability to



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

Fig. B.4: Example from the DeepCoder domain. The task requires to extract the running maximum of the first input, add this to the second input, and multiply the resulting list by 3.

synthesize longer programs, recombine operations in new ways, and extend operation functionality, mirroring the meta-benchmark used in the string domain.

- **Length Generalization:** Training programs have lengths 1–4, while test programs are length 5. This task evaluates whether a model can generalize to longer programs and more complex compositions than seen during training.
- **Compose Different Concepts:** This task assesses the ability to combine operations from different conceptual groups. Training programs include operations from one of two concepts: (i) all first-order operations plus `Map`, and (ii) all remaining higher-order operations. Test programs combine operations from both concepts, with lengths 1–4, to test generalization to novel combinations.
- **Switch Concept Order:** Similar to Compose Different Concepts, this task tests whether a model can apply known concepts in a different order. Training programs follow one fixed sequence of operations across the two concepts, while test programs reverse the order. Program lengths range from 1–4 for both training and testing.
- **Compose New Operation:** This task evaluates the integration of a previously isolated operation into larger compositions. 25% of training tasks are length-1 programs containing only the `Scanl1` operation, while the remainder are length 2–4 programs that exclude `Scanl1`. Test programs are length 2–4 and include `Scanl1`, measuring whether the model can leverage isolated knowledge to compose more complex programs.

– **Add Operation Functionality:** This task assesses whether a model can extend its understanding of an operation by generalizing to previously unseen use cases. Training programs (lengths 1–4) use `Scanl1` only with the lambdas `(-)` and `(min)`, while test programs apply `Scanl1` with other lambdas `(+)`, `(*)`, and `(max)`. This evaluates the model's ability to infer new operation behaviors from existing examples.

B.3 LambdaBeam Domain

The LambdaBeam domain [30] extends the DeepCoder domain with two key modifications. First, lambda functions are no longer restricted to a fixed set – where DeepCoder hardcodes lambdas such as `(+1)` or `(*2)`, LambdaBeam requires the synthesizer to construct lambda functions dynamically by selecting both the operation and its arguments (two constants, a constant and a variable, or two variables). Second, the DSL introduces conditional operations such as `If`, which breaks the additive, sequential structure of DeepCoder: program execution becomes branching, and the path to the solution is no longer a single linear chain. Together, these modifications substantially increase the combinatorial complexity of the search space and the difficulty of compositional generalization. The task structure, state representation, and state update semantics are otherwise identical to the DeepCoder domain (Sec. B.2).

DSL Operations and Program Structure Figure B.5 shows all DSL operations used in the LambdaBeam domain.

Benchmark Creation Benchmark creation follows the same protocol as the DeepCoder domain (Sec. B.2), with the same five compositional generalization categories: length generalization, compose different concepts, switch concept order, compose new operation, and add operation functionality. The definitions and train/test split construction are identical; we refer the reader to Sec. B.2 for details.

$$
\begin{aligned}
\text{Program } P &:= i_1;\ i_2;\ \ldots;\ a_1;\ a_2;\ \ldots \\
\text{Initialization } i &:= v \leftarrow \texttt{INPUT} \\
\text{Assignment } a &:= v \leftarrow f \mid v \leftarrow h \mid v \leftarrow c \\
\text{First-Order Operation } f &:= \texttt{Head}(l) \mid \texttt{Last}(l) \mid \texttt{Minimum}(l) \mid \texttt{Maximum}(l) \mid \texttt{Sum}(l) \mid \\
&\quad\ \texttt{Take}(n, l) \mid \texttt{Drop}(n, l) \mid \texttt{Reverse}(l) \mid \texttt{Sort}(l) \\
\text{Higher-Order Operation } h &:= \texttt{Map}(\lambda, l) \mid \texttt{Filter}(\beta, l) \mid \texttt{Count}(\beta, l) \mid \texttt{ZipWith}(\Sigma, l, l) \mid \\
&\quad\ \texttt{Scanl1}(\Sigma, l) \\
\text{Conditional Operation } c &:= \texttt{If}(\beta, l, l) \\
\text{int} \rightarrow \text{int Lambda } \lambda &:= (+\, e) \mid (-\, e) \mid (*\, e) \mid (/\, e) \quad \text{where } e \in [-5, 5] \\
\text{int} \rightarrow \text{bool Lambda } \beta &:= (>\ e) \mid (<\ e) \mid (\%2 == 0) \mid (\%2 == 1) \quad \text{where } e \in [-5, 5] \\
(\text{int}, \text{int}) \rightarrow \text{int Lambda } \Sigma &:= (+) \mid (-) \mid (*) \mid (\texttt{min}) \mid (\texttt{max}) \\
\text{Integer Variable } n &:= v \\
\text{List Variable } l &:= v \\
\text{Variable Name } v &:= x_1 \mid x_2 \mid \ldots
\end{aligned}
$$

Fig. B.5: First-Order, Higher-Order, and Conditional operations contained in the DSL for the LambdaBeam domain. Unlike DeepCoder, lambda arguments are not restricted to a fixed set – the synthesizer must select both the operation and its arguments.

C TIIPS

C.1 Why is TIIPS cooperative?

In guided steps, the transductive guide f_{gui} actively contributes by proposing intermediate states that redirect search. In free steps, the inductive synthesizer f_{syn} actively contributes by reasoning toward the final outputs without constraint. Neither component is reduced to a preprocessing step or a translation role – both function as genuine problem-solving agents across the schedule – ensuring *Dual Agency*. Within a single trajectory at iteration j the switch between guided and free mode occurs at the step level. Across the full schedule, every step position $t \leq K$ is approached by transduction at some iteration and by induction at another. The schedule thus realizes *Interleaved Granularity* globally: no step is permanently assigned to a single paradigm. At iteration j, the inductive synthesizer reasons freely over all $K - j$ remaining steps after the last guided transition – unconstrained by any further consistency requirements. Across the schedule, each step position is eventually handled in inductive mode, meaning induction retains full search autonomy over all $j + 1$ steps at the appropriate iteration. No transition imposes constraints on the steps that follow it distilling *Search Autonomy Preservation*.

C.2 Model Training & Hyperparameters

Training data is generated through randomly sampling a program from the DSL and applying it to random inputs. Each program is a sequence of subprograms that are categorized as detailed in Appendices B.1, B.2, and B.3. For each subprogram, the training data incorporates three elements: (A) the program state resulting from executing preceding subprograms, (B) the execution result of the current subprogram, and (C) the subprogram itself. The transductive guidance model is trained to predict the execution result (B) of the current subprogram given the current program state (A). The inductive model learns to predict the subprogram (C) conditioned on subtask specifications–specifically, (A) augmented with (B). All models undergo separate training for each generalization task, enabling specialization to the specific characteristics of individual task domains. The same architectures and hyperparameters as in the original ExeDec paper were used for both guidance and synthesizer model [31]. The setup used an embedding dimension of 512, a hidden dimension of 1024, 3 layers, and 4 attention heads. For relative attention, 32 buckets for relative position embeddings, with logarithmically spaced bucket boundaries, were used. The maximum relative distance was determined based on the input and output sequence lengths. Models were trained using the Adam optimizer with a learning rate of 2×10^{-4}, employing linear warm-up for 16,000 steps followed by square root decay. We used a batch size of 128 and trained for 500,000 steps on freshly generated synthetic data, ensuring no repetition of examples. Due to hardware constraints, LambdaBeam uses a smaller model variant (embedding dimension 360, hidden

dimension 720); all baselines within this domain use identical architectures, preserving the validity of cross-approach comparisons. Additionally, we used a batch size of 64 and trained for 1,000,000 steps using a learning rate of 5×10^{-5}. Training required approximately one day per category, using a single RTX A6000 GPU per model.

C.3 Model Inference - Beam Search

While Alg. 1 describes a single synthesis attempt, the actual implementation runs multiple attempts efficiently in parallel using a modified beam search, following the design introduced in ExeDec [31]. We refer the reader to that work for full formal details and restrict ourselves to describing the key design choices relevant to TIIPS. The beam maintains k candidate sequences in parallel, where each candidate is a partial solution trajectory consisting of predicted intermediate states and their corresponding subprograms accumulated across steps. Candidates are scored by summing the log-probabilities assigned by f_{gui} and f_{syn} to their respective outputs at each step. A candidate is invalidated and dropped from the beam if any predicted state or subprogram fails to parse or execute, if it produces program functionality equivalent to a higher-scoring candidate already in the beam, or if a domain-specific or compute limit is reached.

Crucially, we do not run a separate beam search for each model call. Instead, a single beam search runs continuously throughout the full intervention schedule of Alg. 1: each call to f_{gui} or f_{syn} extends the ongoing candidates rather than restarting from scratch. This ensures that the cooperative interaction between guided and free steps is preserved across the entire trajectory within each beam element.

D Metrics

The the primary performance metric is task accuracy – the fraction of test tasks for which a correct program is found. To assess robustness beyond task accuracy, we measure how closely the synthesized programs mirror the ground-truth solution trajectories, using two complementary metrics. Intent match measures semantic alignment between the predicted and ground-truth execution traces. Let $T = \{t_1, \ldots, t_n\}$ be the ground-truth subtasks and $\hat{T} = \{\hat{t}_1, \ldots, \hat{t}_m\}$ the predicted subtasks (or execution results of predicted subprograms in unguided synthesis). Intent match is defined as:

$$I = \frac{|T \cap \hat{T}|}{|T|} \cdot 100\%$$

Syntactic overlap measures structural similarity between the predicted and ground-truth programs. Let $P = \{p_1, \ldots, p_n\}$ be the ground-truth subprograms and $\hat{P} = \{\hat{p}_1, \ldots, \hat{p}_m\}$ predicted subprograms. Syntactic overlap is defined as:

$$S = \frac{\left|\{p_i \in P \mid \exists \hat{p}_j \in \hat{P} : p_i = \hat{p}_j\}\right|}{|P|} \cdot 100\%$$

A program that closely mirrors the ground-truth in both semantics and syntax is more likely to generalize to unseen inputs, as it captures the intended transformation rather than an incidental one that happens to satisfy the training examples. Syntactic overlap serves as the primary robustness proxy, as it directly targets the symbolic artifact; intent match provides a secondary semantic signal.

E Additional Results

E.1 Significance Tests

All pairwise comparisons between approaches are assessed using paired t-tests at a significance level of $\alpha = 0.05$, applied per domain and category across 5 random seeds. Each test compares the per-seed task accuracy of two approaches, treating seeds as paired observations. Tables 2, 3, and 4 report the full p-values and t-statistics for RobustFill, DeepCoder, and LambdaBeam respectively. Cells with $p < 0.05$ indicate a statistically significant difference between the two approaches in that category.

Category	Baseline vs ExeDec	Baseline vs TIIPS	ExeDec vs TIIPS
Test on training distribution	$p = 3.305\times10^{-10}, t = -367.1$	$p = 3.295\times10^{-10}, t = -367.3$	$p = 7.048\times10^{-2}, t = -2.449$
Length generalization	$p = 1.241\times10^{-10}, t = -468.9$	$p = 1.544\times10^{-10}, t = -444.0$	$p = 3.739\times10^{-1}, t = -1.000$
Compose different concepts	$p = 5.179\times10^{-10}, t = -328.1$	$p = 3.896\times10^{-10}, t = -352.3$	$p = 7.048\times10^{-2}, t = -2.449$
Switch concept order	$p = 2.979\times10^{-9}, t = -211.8$	$p = 3.441\times10^{-9}, t = -204.3$	$p = 3.739\times10^{-1}, t = -1.000$
Compose new operation	$p = 1.321\times10^{-8}, t = -146.0$	$p = 9.722\times10^{-9}, t = -157.6$	$p = 2.890\times10^{-3}, t = -6.500$
Add operation functionality	$p = 1.484\times10^{-9}, t = -252.1$	$p = 1.867\times10^{-9}, t = -238.1$	$p = 2.080\times10^{-1}, t = -1.500$
Generalization average	$p = 3.011\times10^{-18}, t = -23.91$	$p = 2.936\times10^{-18}, t = -23.93$	$p = 4.322\times10^{-3}, t = -3.151$

Table 2: Paired t-tests on the RobustFill domain. Each cell shows p-value and t-statistic.

Category	Baseline vs ExeDec	Baseline vs TIIPS	ExeDec vs TIIPS
Test on training distribution	$p = 7.223\times10^{-4}, t = -9.371$	$p = 5.368\times10^{-5}, t = -18.19$	$p = 1.045\times10^{-2}, t = -4.546$
Length generalization	$p = 6.452\times10^{-6}, t = -31.00$	$p = 3.114\times10^{-5}, t = -20.87$	$p = 4.217\times10^{-3}, t = -5.866$
Compose different concepts	$p = 5.699\times10^{-4}, t = -9.964$	$p = 1.184\times10^{-5}, t = -26.62$	$p = 2.469\times10^{-4}, t = -12.35$
Switch concept order	$p = 2.251\times10^{-2}, t = -3.612$	$p = 1.312\times10^{-4}, t = -14.51$	$p = 2.868\times10^{-5}, t = -21.31$
Compose new operation	$p = 1.756\times10^{-1}, t = 1.644$	$p = 5.207\times10^{-6}, t = -32.71$	$p = 8.678\times10^{-4}, t = -8.935$
Add operation functionality	$p = 1.351\times10^{-3}, t = -7.957$	$p = 2.204\times10^{-5}, t = -22.77$	$p = 2.530\times10^{-4}, t = -12.27$
Generalization average	$p = 1.855\times10^{-5}, t = -5.319$	$p = 3.179\times10^{-16}, t = -19.50$	$p = 2.742\times10^{-9}, t = -9.143$

Table 3: Paired t-tests on the DeepCoder domain. Each cell shows p-value and t-statistic.

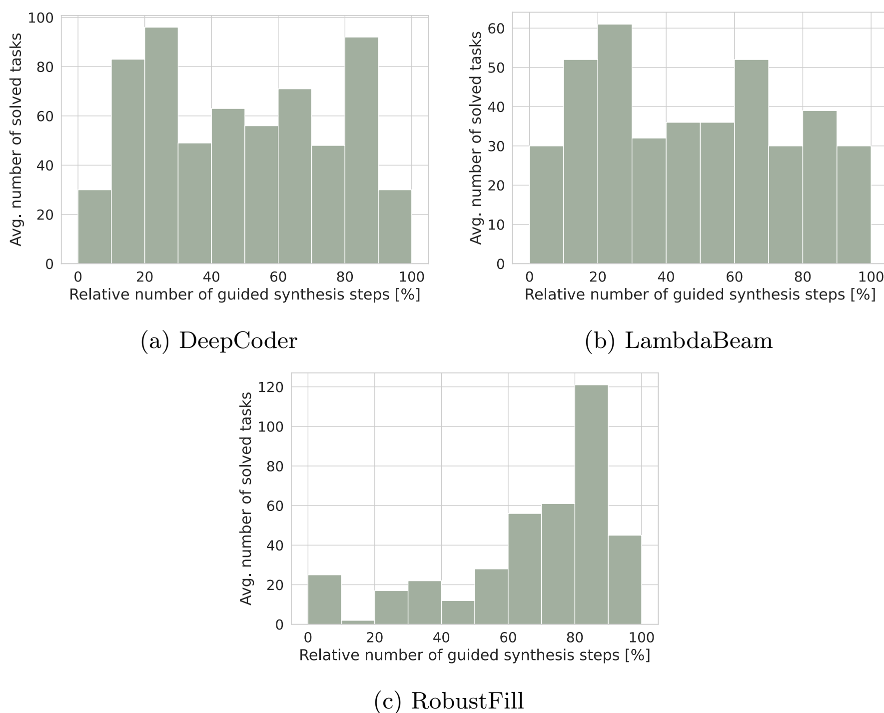
Category	Baseline vs ExeDec	Baseline vs TIIPS	ExeDec vs TIIPS
Test on training distribution	$p = 8.407\times10^{-3}, t = -4.839$	$p = 1.275\times10^{-4}, t = -14.62$	$p = 1.532\times10^{-3}, t = -7.698$
Length generalization	$p = 5.710\times10^{-3}, t = -5.395$	$p = 2.574\times10^{-4}, t = -12.22$	$p = 7.069\times10^{-3}, t = -5.082$
Compose different concepts	$p = 6.816\times10^{-5}, t = -17.13$	$p = 1.297\times10^{-5}, t = -26.01$	$p = 6.421\times10^{-5}, t = -17.39$
Switch concept order	$p = 1.173\times10^{-1}, t = 1.991$	$p = 8.871\times10^{-5}, t = -16.02$	$p = 1.483\times10^{-5}, t = -25.15$
Compose new operation	$p = 6.578\times10^{-4}, t = -9.601$	$p = 3.152\times10^{-5}, t = -20.81$	$p = 4.746\times10^{-3}, t = -5.679$
Add operation functionality	$p = 1.283\times10^{-1}, t = -1.913$	$p = 5.036\times10^{-4}, t = -10.29$	$p = 1.270\times10^{-3}, t = -8.088$
Generalization average	$p = 8.471\times10^{-6}, t = -5.632$	$p = 1.786\times10^{-13}, t = -14.66$	$p = 5.329\times10^{-10}, t = -9.959$

Table 4: Paired t-tests on the LambdaBeam domain.

E.2 Analysis of the number of Transductive Interventions

Figure E.6 shows the distribution of transductive intervention levels j at which tasks are solved across all three domains, aggregated over all categories and seeds.

In both list domains, solved tasks are distributed roughly uniformly across intervention levels $0 \leq j \leq J$. This is more informative than a clean plateau would have been: it indicates that the cooperative dividend is not concentrated at a specific bottleneck depth but is spread across the full trajectory. This supports the general claim motivating the cooperative framework – that syntactic bottlenecks occur throughout synthesis rather than only at the first step – and confirms that the incremental schedule is necessary to capture gains at all depths rather than a fixed prefix of length one or two being sufficient. Notably, a non-trivial fraction of tasks are solved at $j = 0$, i.e., by purely inductive search without any transductive guidance. This is expected: the cooperative framework does not claim that transductive guidance is always beneficial, only that it is beneficial when induction stalls. The schedule's value is precisely that it attempts induction first and escalates guidance only when needed.



(a) DeepCoder

(b) LambdaBeam

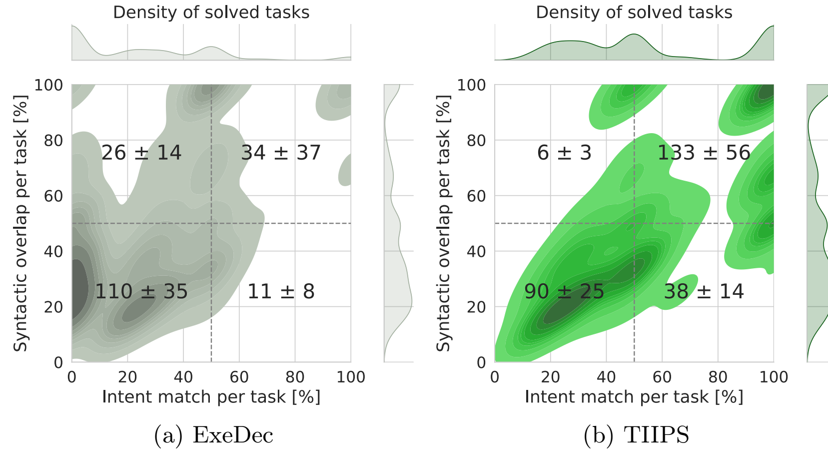
(c) RobustFill

Fig. E.6: Distribution of the degree of guidance of tasks solved by TIIPS. The x-axis shows the number of guided steps relative to the number of steps the discovered solution consists of.

In RobustFill, the distribution is strongly right-skewed: the majority of tasks are solved at or near $j = J$, meaning the schedule naturally escalates to full transductive guidance for most tasks. This is not a failure of the cooperative framework – it is a direct prediction of it. In a single-trace domain with a reliable transductive guide, the synthesizer rarely needs to exercise search autonomy, and cooperation converges to subordination. The schedule's adaptive character surfaces this convergence empirically rather than assuming it, providing an internal consistency check that validates both the domain-level analysis in Sec. 6 and the claim that RobustFill on-par performance reflects domain structure rather than a limitation of TIIPS. Far from being a post-hoc justification, this pattern was predicted by the cooperative framework before the experiments were run: a near-perfect guide makes search autonomy redundant, and the schedule correctly identifies this by escalating to full guidance.

E.3 Cooperation improves Robustness

Figure E.7 shows the kernel density plots for the LambdaBeam domain. The pattern mirrors DeepCoder. TIIPS solutions concentrate strongly in the top-right quadrant (133±56 tasks versus 34±37 for ExeDec), with a pronounced right-skew on the intent match axis. ExeDec solutions distribute broadly across the lower quadrants, with a substantial cluster in the bottom-left – programs that satisfy the I/O examples but deviate from the ground-truth trajectory in both semantics and syntax. This pattern is consistent with the subordination failure mode being amplified in LambdaBeam: the branching, non-additive program structure introduced by `If-Else` conditionals means that a single misdirected transduc-

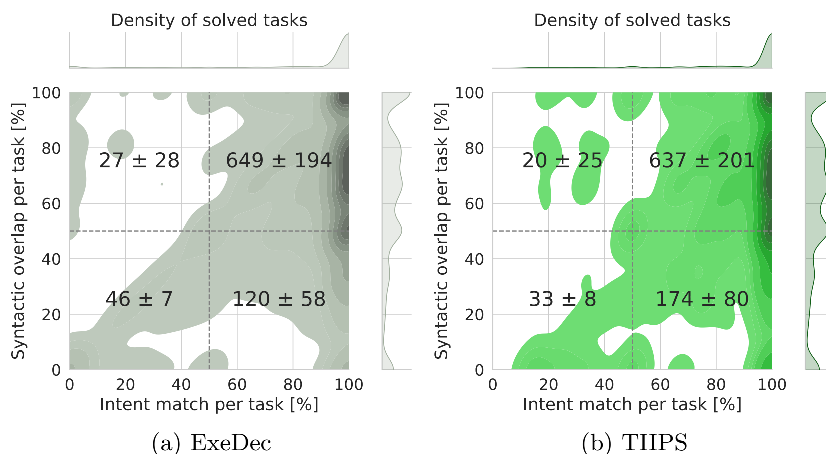


(a) ExeDec (b) TIIPS

Fig. E.7: Density plot of tasks solved by TIIPS and ExeDec in the LambdaBeam domain, grouped by intent match and syntactical overlap. TIIPS solutions show higher intent and syntactical overlap with the ground truth. Displayed numbers show the average number of task solutions and their standard deviation.

tive prediction can foreclose entire execution branches, forcing the synthesizer to generate increasingly convoluted programs to satisfy a wrong intermediate state. The result is a larger proportion of low-quality solutions for ExeDec compared to DeepCoder, and a correspondingly larger quality advantage for TIIPS.

Figure E.8 shows the kernel density plots for the RobustFill domain. As predicted by the convergence argument, the distributions of TIIPS and ExeDec are nearly indistinguishable: both concentrate strongly in the top-right quadrant, indicating that the majority of solved tasks mirror the ground-truth trajectory closely in both semantics and syntax. This is expected in a single-trace domain: When only one canonical decomposition exists, any approach that solves the task is likely to do so via the intended trajectory, regardless of whether it uses cooperative or subordinated reasoning. The near-identical density distributions provide a further internal consistency check: if TIIPS were solving tasks via qualitatively different trajectories than ExeDec in this domain, we would expect divergence in the density plots even where task accuracy is similar. The absence of such divergence confirms that the two approaches are genuinely converging to the same solutions in RobustFill, rather than TIIPS finding different programs that happen to achieve the same accuracy. This rules out the alternative explanation that TIIPS's on-par performance in RobustFill conceals qualitative differences in solution behavior, and strengthens the conclusion that convergence in this domain is structural rather than incidental.



(a) ExeDec (b) TIIPS

Fig. E.8: Density plot of tasks solved by TIIPS and ExeDec in the RobustFill domain, grouped by intent match and syntactical overlap. Both distributions closely align showing high intent and syntactical overlap with the ground truth. Displayed numbers show the average number of task solutions and their standard deviation.