COOL: Efficient and Reliable Chain-Oriented Objective Logic with Neural Networks Feedback Control for Program Synthesis

Anonymous Authors[1]

Abstract

Modern program synthesis methods—whether formal or neural—fail to balance automation with fine-grained control and modular adaptability, limiting their utility in complex, real-world software development. This shortfall stems from rigid Domain-Specific Language (DSL) frameworks and neural network mispredictions. We address these challenges with **COOL (Chain-Oriented Objective Logic)**, a neural-symbolic framework that introduces: (1) **Chain-of-Logic (CoL)**: Structures synthesis into hierarchical, expert-guided activities using heuristic vectors to decompose DSLs and prioritize rule applications; and (2) **Neural Network Feedback Control (NNFC)**: A self-correcting mechanism that isolates neural components into reusable libraries, filtering erroneous predictions via sequential network coupling. Evaluated on relational and symbolic tasks, CoL achieves **70% higher accuracy** than traditional DSLs while reducing computational overhead by **91% fewer tree operations** and **95% faster synthesis**. Under adversarial conditions—insufficient training data, increased complexity, and multidomain requirements—NNFC further improves accuracy by **6%** and reduces tree operations by **64%**. These improvements confirm COOL as a highly efficient and reliable program synthesis framework.

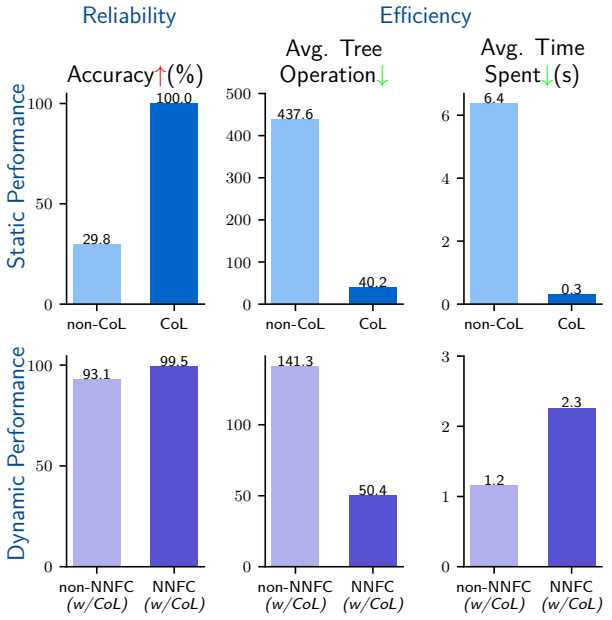

Figure 1. Performance Enhancements with CoL and NNFC. The CoL DSL surpasses non-CoL DSL in all metrics. While NNFC increases computation time due to neural network calls, it significantly boosts accuracy in dynamic experiments, enhancing reliability.

1. Introduction

Program synthesis is becoming increasingly important in computer science for enhancing development efficiency (Gulwani et al., 2017; Jin et al., 2024). Despite the effectiveness of current state-of-the-art methods in dealing with common, routine tasks, the complexity and heterogeneous requirements of modern software systems demand more domain-specific customization and sophisticated synthesis approaches (Sobania et al., 2022).

To address these challenges, a robust synthesis framework must provide two foundational capabilities: **(1)** fine-grained control to steer the synthesis process toward task-specific objectives while maintaining interpretability, and **(2)** flexible modularity to decompose complex tasks into reusable, verifiable components. These principles are interdependent: control ensures synthesis paths align with domain expert intuition, while modularity enables systematic reuse of validated logic across tasks, reducing chances of errors (Groner et al., 2014; Sullivan et al., 2001).

However, existing methods fail to harmonize these requirements. Symbolic approaches (e.g., SyGus (Alur et al., 2013), Escher (Albarghouthi et al., 2013), and FlashFill++ (Cambronero et al., 2023)) rely on rigid, traversal-based DSLs that cannot adapt synthesis strategies mid-process, forcing programmers to accept suboptimal paths or abandon synthesis entirely for complex tasks. A compensatory strategy

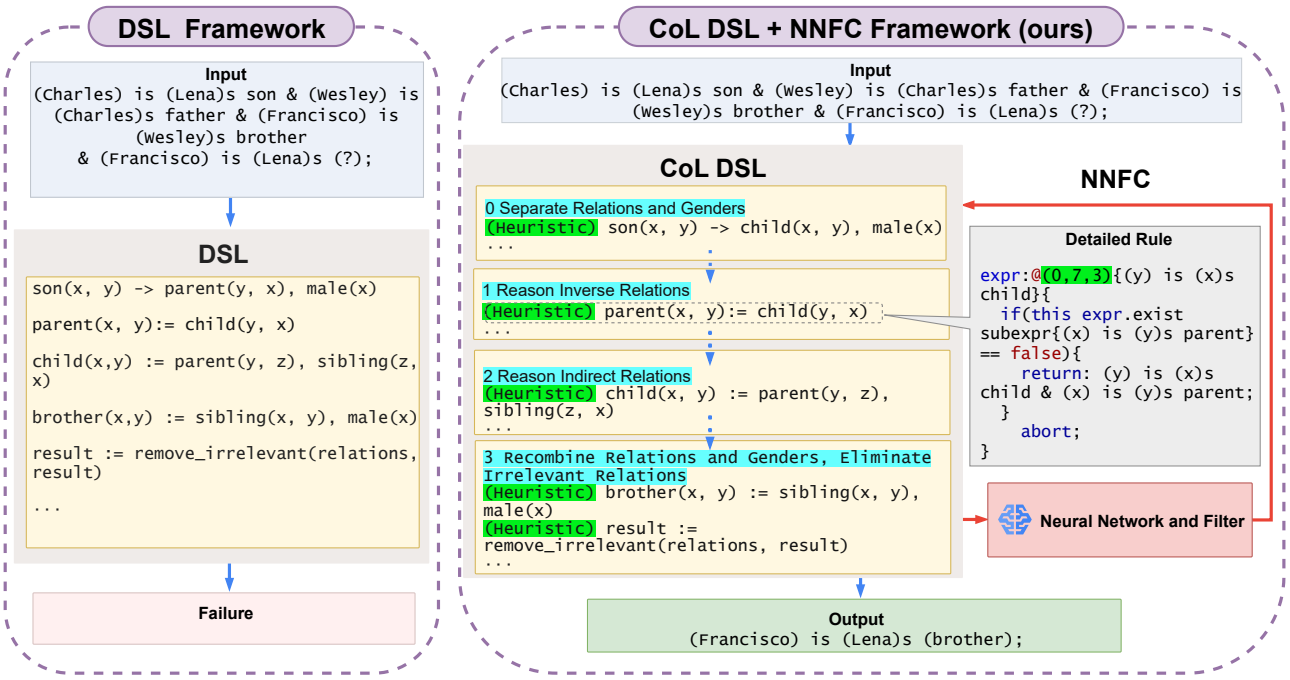

Figure 2. Chain-of-Logic (highlighted part) organizes the rule application into a structured activity flow, enhancing the Domain-Specific Language (DSL) framework's ability to handle complex tasks. The Neural Network Feedback Control mechanism (red path) utilizes data during synthesis to improve the performance of the synthesis process dynamically.

involves using neural networks for guidance or search space pruning, as seen in projects such as Neo (Feng et al., 2018), LambdaBeam (Shi et al., 2023a), Bustle (Odena et al., 2020), DreamCoder (Ellis et al., 2023), and Algo (Zhang et al., 2023), but their black-box heuristics alienate programmers from influencing critical control decisions. Even LLM-based tools (e.g., Code Llama (Roziere et al., 2023), CodeGen (Nijkamp et al., 2022)) while flexible, remain prone to dataset biases and logic hallucinations, with unavoidable high computational costs even for optimized models like DeepSeek (Guo et al., 2025). This gap underscores a critical need: synthesis frameworks must embed programmer expertise directly into the control flow while isolating neural components to prevent error cascades—a dual requirement unmet by current paradigms.

In this paper, following the principles of fine-grained control and flexible modularity, we present **COOL (Chain-Oriented Objective Logic)**, a neural-symbolic framework for complex program synthesis. At the core of our approach, we introduce the **Chain-of-Logic (CoL)**, which applies activity-diagram concepts to enable fine-grained control (Gomaa, 2011). As illustrated in Figure 2, domain experts can precisely organize rules into multiple activities and manage control flow using heuristics and keywords. Building on CoL, we introduce **Neural Network Feedback Control (NNFC)** (Turan & Jäschke, 2024) to dynamically refine the synthesis process. NNFC leverages historical synthesis records to train neural networks and filters out erroneous predictions, thereby improving overall reliability. Neural networks are encapsulated with corresponding CoL DSL library files, ensuring clear isolation and reusability. By combining CoL and NNFC, COOL delivers high efficiency and reliability, making it well-suited for complex program synthesis tasks.

We evaluate CoL and NNFC on two types of experiments: static (fixed-condition experiment with constant domain and difficulty, using pre-trained networks without further training) and dynamic (challenging experiment with evolving domain and difficulty, where networks are created and continuously trained). As shown in Figure 1, CoL alone significantly improves accuracy by 70%, while reducing tree operations by 91% and time by 95% in static experiments. In dynamic experiments, NNFC further boosts accuracy by 6% and cuts tree operations by 64%. These results underscore the importance of fine-grained control and flexible modularity in DSL program synthesis.

The contributions of our work are as follows:

1. We propose the **Chain-of-Logic (CoL)**, a structured synthesis paradigm that decomposes DSLs into expert-guided activities using heuristic vectors, enabling interpretable, fine-grained control over rule applications.

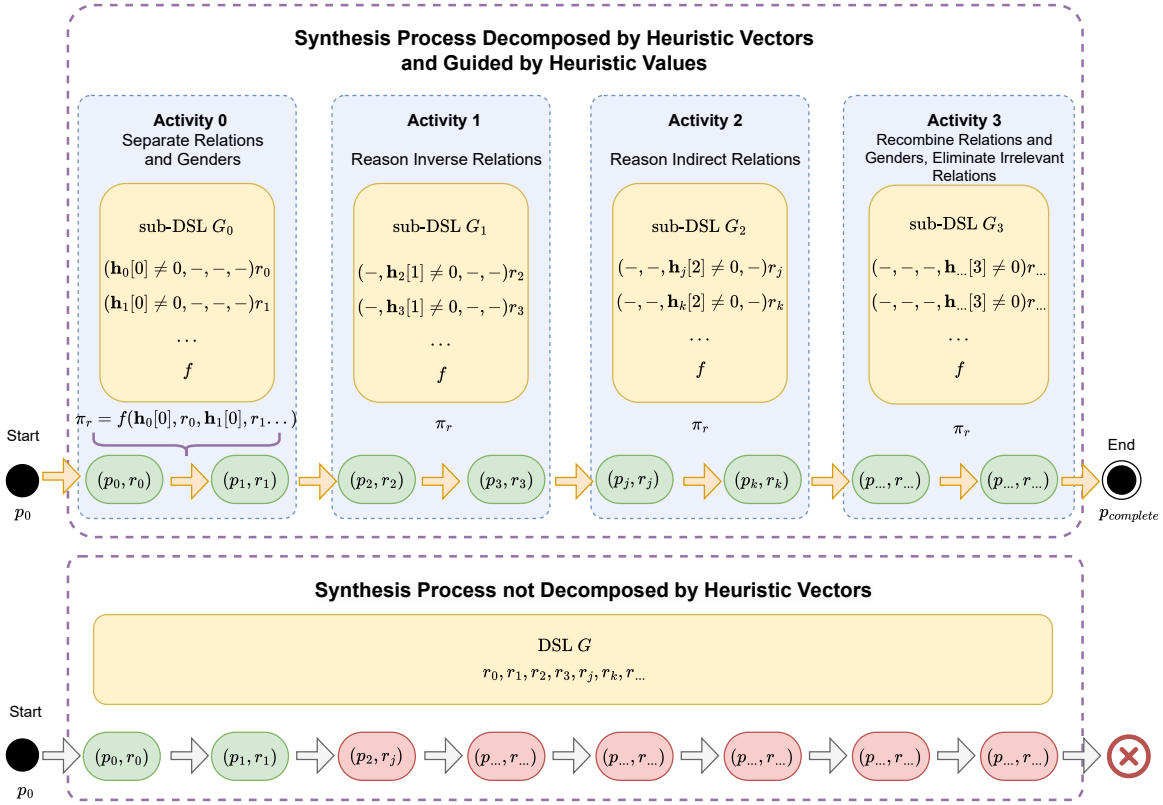

Figure 3. Heuristic-guided decomposition of the DSL G in Chain-of-Logic. Domain experts provide heuristic vectors $\mathbf{h}$ to decompose the DSL G into multiple sub-DSLs (G_0, G_1, G_2, G_3), where non-zero entries in $\mathbf{h}$ determine component inclusion. These sub-DSLs correspond to the activities depicted in Figure 2 and operate on partial programs p using rules r. The synthesis process for each activity is guided by rule application policies π_r, which are generated by a heuristic algorithm f that uses heuristic values $\mathbf{h}[n]$ as input. In our experiments, $\mathbf{h}_i[n]$ acts as a reward for applying rule r_i during activity n (Algorithm 1), prioritizing rules with higher heuristic values.

2. We further introduce **Neural Network Feedback Control (NNFC)**, a self-correcting mechanism that isolates neural components into reusable libraries, filtering erroneous predictions via sequential network coupling to ensure reliability.

3. We present **COOL**, which unifies CoL and NNFC into a neural-symbolic architecture, achieving efficient, reliable, and human-aligned program synthesis.

2. Method

In this section, we detail the implementation of CoL and NNFC, outlining the principles that ensure high efficiency and reliability for complex program synthesis tasks.

2.1. Chain-of-Logic (CoL)

Activity diagrams, a foundational modeling tool in software engineering, effectively represent the transition from an initial state to a final state through a sequence of structured activities. This process aligns closely with DSL-based program synthesis, where a Domain-Specific Language (DSL)—formally defined as a context-free grammar—undergoes derivation by incrementally transforming partial programs containing nonterminal symbols into complete programs through the application of rules (Appendix A). However, as the rule set grows, traditional DSL becomes computationally inefficient in exploring partial program derivations. To enhance the efficiency of DSL, the Chain-of-Logic, drawing inspiration from activity diagrams, organizes the synthesis process into well-defined, expert-guided activities.

CoL improves the synthesis workflow of the DSL with two key features: *heuristic vectors* and *keywords*:

As shown in Figure 3, heuristic vectors decompose DSL into multiple sub-DSLs that are consistent with the activity flow. By partitioning rule scopes, CoL enables more efficient synthesis within each activity. For example, in Figure 2, a rule with the heuristic vector `(0,7,3)` belongs to sub-DSLs in activities 1 and 2 with heuristic values of 7 and 3, respectively. By dynamically pruning the search space

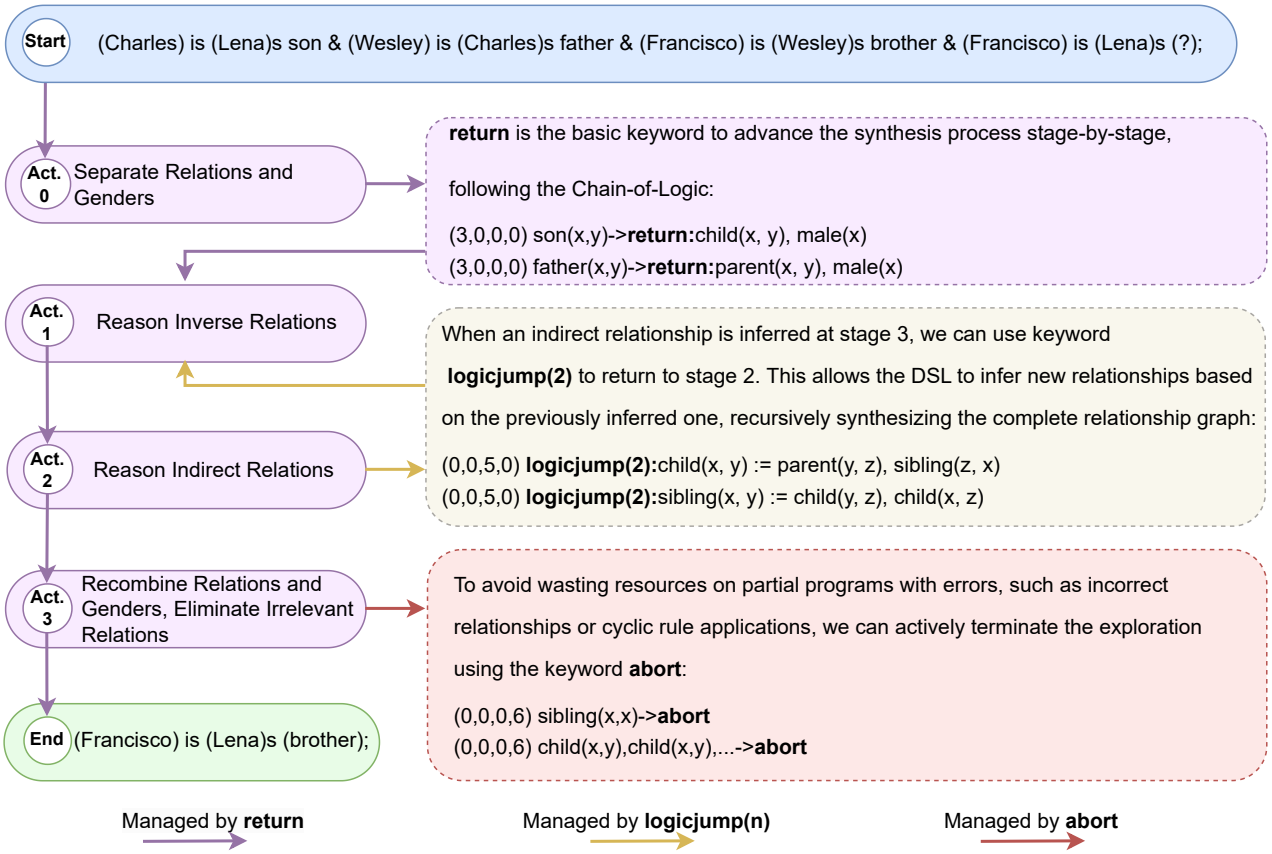

Figure 4. Keywords in Chain-of-Logic. In this illustrative CoL DSL, each node represents an activity where a set of rules can be applied to generate partial programs. The flow between activities is managed by keywords **return**, **logicjump(n)**, and **abort**, allowing for the implementation of complex control flow in program synthesis.

and providing search guidance, heuristic vectors promote program synthesis efficiency.

In addition to heuristic vectors, CoL introduces three control keywords—`return`, `logicjump(n)`, and `abort`—which dynamically manage state transitions within and across activities during synthesis (Figure 4):

1. **return**: Ends the current rule, staying within the current activity or advancing to subsequent activities.
2. **logicjump(n)**: Jumps directly to activity n, enabling branching and loops within activity flow.
3. **abort**: Terminates the current synthesis branch, pruning the search space.

Based on the principle of activity diagrams, CoL provides fine-grained control through heuristic vectors and keywords. This systematic approach enhances the efficiency of DSL synthesis. (Refer to Appendix A.4 for a detailed explanation of the CoL DSL synthesis process.)

2.2. Neural Network Feedback Control (NNFC)

While CoL enables domain experts to embed specialized knowledge into the synthesis process, its control flow may lack the flexibility to fully accommodate task-specific variations or adapt to evolving scenarios. To overcome this limitation, **Neural Network Feedback Control (NNFC)** dynamically refines the control flow by leveraging historical neural network feedback, enhancing both precision and adaptability. However, erroneous neural predictions introduce a potential risk to system reliability.

A robust control mechanism within NNFC is essential for maintaining overall system performance. As illustrated in Figure 5, NNFC enhances the CoL DSL in the following aspects: **(1) Forward Flow:** The Clipper prioritizes control signals aligned with DSNN guidance by capping any inconsistent signals (Appendix C), while the CoL DSL applies rules based on the adjusted heuristic values. **(2) Feedback Loop:** The DSNN generates error signals from partial programs. To suppress the impact of mispredictions, the Filter

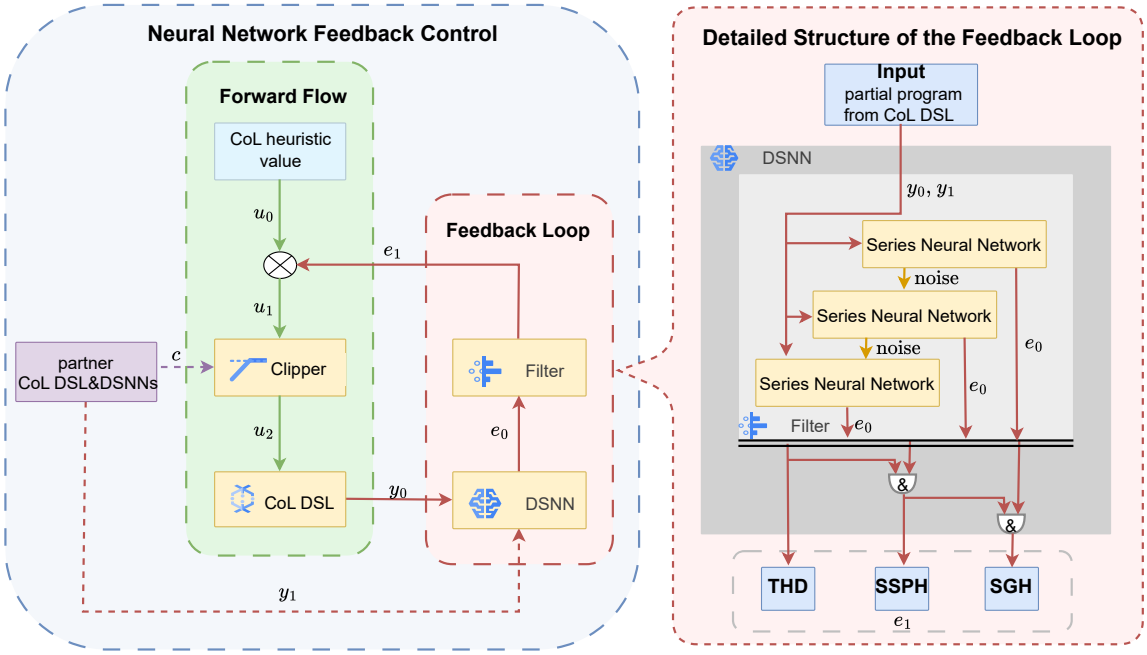

Figure 5. Neural Network Feedback Control. The left side illustrates the complete control loop of NNFC. In the forward flow (green path), heuristic values u guide the synthesis process as control signals. In the feedback loop (red path), the **DSNN (Domain-Specific Neural Network, the neural network paired with a DSL)** generates initial error signals e_0 from partial programs y. These singals are then filtered to produce high-quality error signals e_1, which adjust the initial heuristic values u_0. In multidomain synthesis, the CoL DSL and DSNN from the self-domain use partner domain information (dashed path) to clarify tasks and avoid competition, ensuring modularity. The right side details the feedback loop: The DSNN comprises multiple neural networks coupled in series via noise signals, with each network generating its own error signal e_0, then these signals with large discrepancies are filtered, retaining the final high-quality error signals e_1. (See Appendix B for details on the neural network's format, architecture, training, prediction, and role in synthesis.)

refines these signals before they influence the forward flow (Appendix B).

The quality of error signals generated in the feedback loop is crucial to NNFC's overall effectiveness. If these signals are inaccurate, NNFC may not only fail to enhance performance but also deteriorate the CoL DSL's effectiveness. To address this, we employ an inner coupling structure within the DSNN (Figure 5, right): During synthesis tasks, sequentially connected neural networks process partial programs, where each network receives both the partial programs and the intermediate outputs of its predecessor. Early-stage errors can propagate downstream as noise, amplifying at each subsequent stage. Consequently, the difference between consecutive network outputs is positively correlated with the cumulative error. To mitigate excessive error propagation, we define a threshold that filters out signals exhibiting large output discrepancies (Appendix B.5). Finally, the DSNN leverages the filtered signals to produce multi-head outputs, which in turn fine-tune the forward flow (Appendix B.6).

Figure 6 illustrates how the DSNN's multi-head outputs enhance the entire program synthesis pipeline (see Table 5 for detailed output features). Specifically, the DSNN generates three types of outputs:

1. **Task Detection Head (TDH)**: Improves modularity by determining whether the partial program contains components that the CoL DSL can process.

2. **Search Space Prune Head (SSPH)**: (Active when TDH returns true): Assesses the feasibility of deriving a complete program from the current partial program to prevent CoL DSL from exploring infeasible paths.

3. **Search Guidance Head (SGH)**: (Active when both TDH and SSPH are true) Guides the CoL DSL toward the most promising rule applications, optimizing the search process.

By integrating filtering mechanisms and multi-head outputs, the feedback loop supplies high-quality error signals to the forward path, ensuring that NNFC effectively enhances the synthesis process beyond CoL's foundational capabilities.

3. Experiments

We conduct the experiments in two stages to evaluate the improvements introduced by CoL to DSL and to assess how NNFC further enhances performance. First, we carry out

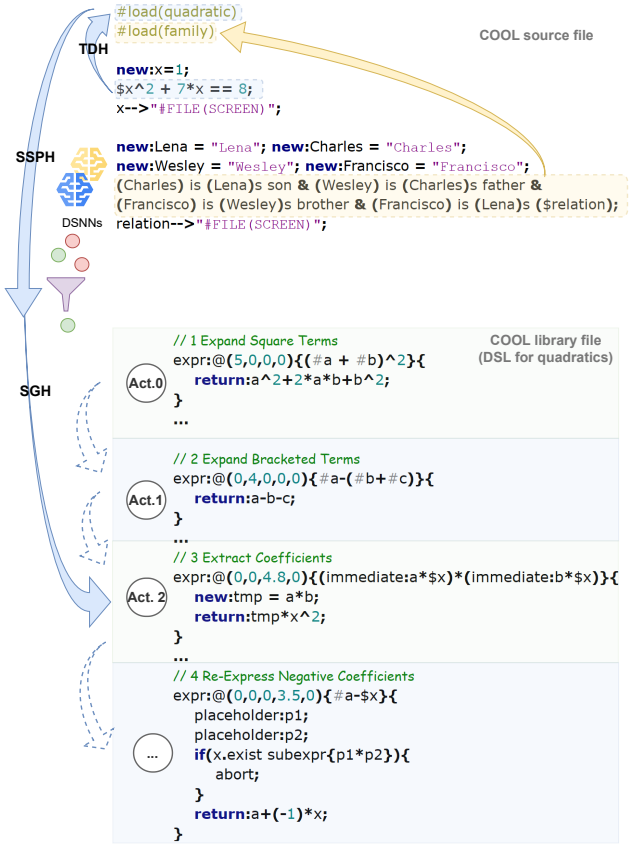

Figure 6. How DSNN's Multi-Head Outputs Function. When a specific DSL library is loaded, its corresponding DSNN (a specialized network) is loaded to assist in program synthesis by influencing rule application policies for source code partial programs and their derivatives. For instance, if the DSNN bound to the "quadratic" DSL reads a partial program containing a quadratic equation (blue background), it: (1) guides the solver to apply rules specific to its DSL via TDH, (2) prioritizes applying rules on partial programs that are easier to complete using SSPH, and (3) directs the solver to apply specific rules through SGH to progress toward a specified CoL activity. Simultaneously, it prevents the solver from applying quadratic DSL rules to unrelated tasks (e.g., yellow background), ensuring no interference with other DSL synthesis processes. These functions, achieved by modifying heuristic values of DSL-specific rules, demonstrate strong modularity (detailed in Appendix B.6).

static experiments under fixed conditions, including task domain, difficulty level, and neural network. These controlled conditions allow us to accurately measure CoL's impact on performance. Next, we proceed with dynamic experiments, where conditions vary throughout. This dynamic setup evaluates NNFC's ability to improve reliability under more realistic scenarios.

Table 1. Benchmark configurations. Relational benchmarks are divided into easy and difficult groups based on the number of relationship edges, while symbolic benchmarks are based on the number of nodes in the tree.

Benchmark Type	Difficulty Level A	Difficulty Level B
relational	300 tasks with 3 edges	200 tasks with 4 edges
symbolic	300 tasks with around 5 nodes	200 tasks with around 9 nodes

3.1. Experimental Setup

We evaluate how CoL and NNFC improve DSL program synthesis across multiple benchmarks, measuring performance with various metrics.

Benchmarks. Our experiments include both relational and symbolic tasks of varying difficulty, as summarized in Table 1 (see Appendix D for detailed examples). Specifically, the *relational* tasks derive from the CLUTRR (Sinha et al., 2019) dataset, requiring the synthesis of programs that capture target relationships via human-like commonsense reasoning. In contrast, the *symbolic* tasks—generated by GPT (Achiam et al., 2023)—entail transforming non-standard quadratic equations into standard form using manual calculation steps. Although these tasks are straightforward for human solvers, they effectively demonstrate how fine-grained control can significantly enhance the efficiency of DSL program synthesis.

Metrics. We evaluate performance using the following metrics, noting that they reflect **user** interaction with DSL libraries rather than **expert** efficiency in creating them: **(1) Accuracy** is the proportion of synthetic tasks that are completed under constraints.[1] **(2) CPU Overhead** is assessed by the number of tree operations required for synthesis. **(3) Memory overhead** is assessed by the number of transformation pairs (a partial program paired with the rule to be applied. Appendix A.3).**(4) GPU Overhead** is measured by the number of neural network invocations. **(5) Time overhead** is referenced by the actual time spent on program synthesis tasks.

Chain-of-Logic. Chain-of-Logic (CoL). We incorporate CoL into DSLs to better align the synthesis process with human problem-solving strategies. For relational tasks, CoL mirrors typical human reasoning about family relationships by organizing synthesis into distinct activities (Figure 4). For symbolic tasks, CoL follows manual quadratic equa-

[1]Resource limit: Each partial program must be completed with a cost of at most 1000 transformation pairs and the length of the synthesis path cannot exceed 50.

Table 2. Static performance of DSL and CoL DSL for relational and symbolic tasks. CoL DSL significantly outperforms DSL in all metrics.

Benchmark	Group	Accuracy↑(%)	Avg. Tree Operation↓	Avg. Transformation Pair↓	Avg. Time Spent↓ (s)
relational	DSL	11.3	463.9	1432.2	9.43
	CoL DSL	**100.0**	**46.6**	**177.8**	**0.48**
symbolic	DSL	48.3	411.2	2285.3	3.31
	CoL DSL	**100.0**	**33.8**	**92.7**	**0.11**

tion simplification, structuring activities around activities such as term expansion, coefficient extraction, term permutation, and conversion to standard form. Table 7 presents the specific CoL DSL configurations used, highlighting the generality of CoL across diverse problem domains.

Groups. We use multiple groups to comprehensively evaluate CoL and NNFC (as shown in Table 6). First, in **static experiments**, we evaluate CoL by comparing DSL groups with and without CoL enhancements. Second, to isolate the impact of heuristic vectors—both as guides and as structuring tools for rule application—we create groups enhanced only by heuristic values. Third, we introduce groups enhanced by neural networks to assess whether combining CoL with neural networks yields better results and to explore the filtering effect of the inner coupling structure. In **dynamic experiments**, we design control groups with and without NNFC to evaluate its impact. Additionally, we include a group without the inner coupling structure to confirm its necessity.

Environment. Experiments are carried out on a computer equipped with an Intel i7-14700 processor, a GTX 4070 GPU, and 48GB RAM.

3.2. Static Experiments

We first evaluate CoL under fixed conditions (domain, difficulty, and pre-trained neural networks) to isolate its impact. Controlled experiments confirm CoL's systematic improvements across all metrics.

The results in Table 2 clearly demonstrate that **CoL significantly improves accuracy while minimizing overhead**. Most notably, CoL improves the accuracy of the DSL from less than 50% to 100% across both relational and symbolic benchmarks. Additionally, CoL achieves remarkable reductions in relational tasks, cutting tree operations by 90%, transformation pairs by 88%, and time by 95%. Similarly, in symbolic tasks, CoL reduces tree operations by 92%, transformation pairs by 96%, and time by 97%. These findings showcase CoL's substantial impact on improving performance across all key metrics.

Further ablation and extension experiments clarify the sources of CoL's enhancement, confirm CoL's effective integration with neural networks, and explore when filtering via inner coupling structures is most beneficial (Appendix D.4):

First, **CoL's enhancement stems from both heuristics and structured rule application stages**. As illustrated in Figure 8, the DSL (Heuristic) group outperforms the baseline DSL group in most metrics, and the CoL DSL group significantly surpasses DSL (Heuristic) in all metrics. Such results indicate that CoL positively impacts synthesis by guiding and structuring rule application. Furthermore, beyond heuristic guidance, the **introduction of structured rule application activities yields even greater performance gains**, underscoring the importance of explicit activity decomposition in program synthesis.

Second, **integrating CoL with neural networks further improves the search efficiency.** As shown in Figure 8, although this integration introduces additional GPU and time overhead, the top-performing CoL DSL + NN group reduces tree operations by 43% and transformation pairs by 19% in relational tasks compared to the CoL DSL group. In symbolic tasks, the CoL DSL + NN (Cp) group reduces tree operations by 64% and transformation pairs by 46%. The results showcase that **neural networks can further narrow the search space** for program synthesis beyond CoL. Moreover, the **inner coupling structure proves essential for maintaining reliability.** Groups incorporating this structure consistently outperform non-neural groups across both tasks. Conversely, its removal leads to accuracy decline in symbolic tasks, highlighting its role in filtering errors and enhancing synthesis robustness.

Third, the **inner coupling structure proves more effective when error tolerance is low.** As shown in Figure 8, for symbolic tasks, CoL DSL-based groups with the inner coupling structure significantly outperform those without it. However, for relational tasks and DSL-based groups lacking CoL or heuristic support, those without the structure perform better. This discrepancy arises because the structure filters both incorrect and correct predictions, meaning **its net benefit hinges on whether the gain from eliminating**

Table 3. Dynamic performance of CoL DSL and CoL DSL+NNFC(Cp). NNFC significantly improves the dynamic performance of CoL DSL in accuracy, tree operations, and transformation pairs.

Benchmark	Group	Accuracy↑ (%)	Avg. Tree Operation↓	Avg. Transformation Pair↓	Avg. Neural Network Invocation↓	Avg. Time Spent↓(s)
relational	CoL DSL	**100.0**	70.0	259.8	**0**	**1.05**
	CoL DSL+NNFC (Cp)	**100.0**	**54.6**	**224.5**	21.7	2.08
symbolic	CoL DSL	82.6	233.5	977.1	**0**	1.42
	CoL DSL+NNFC (Cp)	**99.4**	**50.3**	**222.2**	21.6	**1.12**
multi-domain	CoL DSL	97.5	115.2	367.6	**0**	**0.99**
	CoL DSL+NNFC (Cp)	**99.0**	**45.6**	**250.5**	72.84	3.91

errors outweighs the cost of losing valid outcomes. Consequently, for relational tasks with limited search spaces and higher error tolerance, filtering exacts a net penalty. By contrast, in symbolic tasks, where errors are more critical, CoL DSL-based groups substantially benefit from the inner coupling structure.

3.3. Dynamic Experiments

We next evaluate NNFC under dynamic conditions (varying domains, difficulty, and evolving neural networks) to assess its adaptability. Results confirm that NNFC significantly enhances reliability, sustaining high accuracy in real-world scenarios while reducing computational overhead despite increasing task complexity.

The results in Table 3 confirm that **NNFC significantly enhances the reliability of CoL DSL in challenging conditions.** As task difficulty increases and multidomain scenarios emerge, the accuracy of the CoL DSL group declines compared to its performance in static experiments. However, the NNFC-enhanced group maintains an accuracy of at least 99%, demonstrating its strong reliability in challenging situations. Additionally, compared with the original CoL DSL group, it reduces tree operations by 22% and transformation pairs by 14%. For symbolic tasks, despite the added time for neural network invocations, the NNFC-enhanced group still shortens the time spent by 21%.

Further ablation experiments confirm that **NNFC's reliability relies primarily on the filtering effect of the inner coupling structure.** (Appendix D.4) As illustrated in Figures 9 and 10, incorporating this structure reduces misprediction-induced accuracy declines by 94%. Moreover, the dynamic performance highlights how the inner coupling structure enhances NNFC's ability to mitigate DSNN errors by filtering out erroneous signals in real time.

In the scenarios where a DSNN underperforms due to issues such as insufficient training data (Mikołajczyk & Grochowski, 2018) (as seen in Figure 9, tasks 51-100), inadequate generalization to more challenging tasks (Yosinski et al., 2014; Wei et al., 2019) (Figure 9, tasks 301-350), and catastrophic forgetting when tasks from a new domain are learned (Kirkpatrick et al., 2017; Van de Ven & Tolias, 2019) (Figure 10, tasks 1-100), incorrect predictions lead the actual synthesis path to deviate from the CoL, which in turn causes inefficiency and reduced accuracy. During these phases, for NNFC with the inner coupling structure, the attenuation ratio spikes, indicating that a large percentage of neural network predictions are filtered out. Consequently, the inner coupling structure ensures that the synthesis process adheres to the CoL, effectively mitigating the negative impact of DSNN mispredictions and enhancing reliability.

The inner coupling structure dynamically balances neural contributions in program synthesis. As the DSNN improves and stabilizes (Figure 9, tasks 101–300, 351–500; Figure 10, tasks 101–400), the attenuation ratio gradually decreases, reflecting a reduced need for filtering and an increased influence of DSNN on the synthesis policy. This adaptive regulation maintains both efficiency and robustness throughout the synthesis process.

4. Conclusion

We introduced Chain-Oriented Objective Logic (COOL) as a unified framework for fine-grained control and flexible modularity in complex program synthesis. Inspired by activity diagrams and control theory, we developed Chain-of-Logic (CoL) and Neural Network Feedback Control (NNFC) to address key limitations of existing DSL approaches. Static and dynamic experiments across relational, symbolic, and multidomain tasks confirm COOL's robust efficiency and reliability. We anticipate that further exploration of CoL and NNFC will drive progress not only in program synthesis but also in the broader field of neural network reasoning.

References

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Albarghouthi, A., Gulwani, S., and Kincaid, Z. Recursive program synthesis. In *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*, pp. 934–950. Springer, 2013.

Alur, R., Bodik, R., Juniwal, G., Martin, M. M., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. *Syntax-guided synthesis*. IEEE, 2013.

Bettini, L. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

Bunel, R., Hausknecht, M., Devlin, J., Singh, R., and Kohli, P. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276*, 2018.

Cambronero, J., Gulwani, S., Le, V., Perelman, D., Radhakrishna, A., Simon, C., and Tiwari, A. Flashfill++: Scaling programming by example by cutting to the chase. *Proceedings of the ACM on Programming Languages*, 7 (POPL):952–981, 2023.

Chaudhuri, S., Ellis, K., Polozov, O., Singh, R., Solar-Lezama, A., Yue, Y., et al. Neurosymbolic programming. *Foundations and Trends® in Programming Languages*, 7 (3):158–243, 2021.

Chen, X., Song, D., and Tian, Y. Latent execution for neural program synthesis beyond domain-specific languages. *Advances in Neural Information Processing Systems*, 34: 22196–22208, 2021.

Chen, X., Li, M., Gao, X., and Zhang, X. Towards improving faithfulness in abstractive summarization. *Advances in Neural Information Processing Systems*, 35:24516–24528, 2022.

Chen, Y., Wang, C., Bastani, O., Dillig, I., and Feng, Y. Program synthesis using deduction-guided reinforcement learning. In *Computer Aided Verification32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II*, volume 12225, pp. 587–610, 2020.

Cui, G. and Zhu, H. Differentiable synthesis of program architectures. *Advances in Neural Information Processing Systems*, 34:11123–11135, 2021.

Drori, I., Zhang, S., Shuttleworth, R., Tang, L., Lu, A., Ke, E., Liu, K., Chen, L., Tran, S., Cheng, N., et al. A neural network solves, explains, and generates university math problems by program synthesis and few-shot learning at human level. *Proceedings of the National Academy of Sciences*, 119(32):e2123433119, 2022.

Eberhardinger, M., Maucher, J., and Maghsudi, S. Towards explainable decision making with neural program synthesis and library learning. In *NeSy*, pp. 348–368, 2023.

Ellis, K., Wong, L., Nye, M., Sable-Meyer, M., Cary, L., Anaya Pozo, L., Hewitt, L., Solar-Lezama, A., and Tenenbaum, J. B. Dreamcoder: growing generalizable, interpretable knowledge with wake–sleep bayesian program learning. *Philosophical Transactions of the Royal Society A*, 381(2251):20220050, 2023.

Feng, Y., Martins, R., Bastani, O., and Dillig, I. Program synthesis using conflict-driven learning. *ACM SIGPLAN Notices*, 53(4):420–435, 2018.

Gomaa, H. *Software modeling and design: UML, use cases, patterns, and software architectures*. Cambridge University Press, 2011.

Groner, R., Groner, M., and Bischof, W. F. *Methods of heuristics*. Routledge, 2014.

Gulwani, S., Polozov, O., Singh, R., et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4 (1-2):1–119, 2017.

Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., and He, Y. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint*, arXiv:2501.12948, 2025. URL https://arxiv.org/abs/2501.12948.

Hart, P. E., Nilsson, N. J., and Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

Hrinchuk, O., Khrulkov, V., Mirvakhabova, L., Orlova, E., and Oseledets, I. Tensorized embedding layers for efficient model compression. *arXiv preprint arXiv:1901.10787*, 2019.

Huang, Z., Xu, W., and Yu, K. Bidirectional lstm-crf models for sequence tagging. *arXiv preprint arXiv:1508.01991*, 2015.

Imambi, S., Prakash, K. B., and Kanagachidambaresan, G. Pytorch. *Programming with TensorFlow: solution for edge computing applications*, pp. 87–104, 2021.

Jacovi, A. and Goldberg, Y. Towards faithfully interpretable nlp systems: How should we define and evaluate faithfulness? *arXiv preprint arXiv:2004.03685*, 2020.

Jin, H., Huang, L., Cai, H., Yan, J., Li, B., and Chen, H. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*, 2024.

Johnson, S. C. et al. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

Kalyan, A., Mohta, A., Polozov, O., Batra, D., Jain, P., and Gulwani, S. Neural-guided deductive search for real-time program synthesis from examples. *arXiv preprint arXiv:1804.01186*, 2018.

King, P. A history of the groovy programming language. *Proceedings of the ACM on Programming Languages*, 4 (HOPL):1–53, 2020.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

Lesk, M. E. and Schmidt, E. *Lex: A lexical analyzer generator*, volume 39. Bell Laboratories Murray Hill, NJ, 1975.

Li, W., Wu, W., Chen, M., Liu, J., Xiao, X., and Wu, H. Faithfulness in natural language generation: A systematic survey of analysis, evaluation and optimization methods. *arXiv preprint arXiv:2203.05227*, 2022.

Li, Y., Parsert, J., and Polgreen, E. Guiding enumerative program synthesis with large language models. In *International Conference on Computer Aided Verification*, pp. 280–301. Springer, 2024.

Liang, C., Norouzi, M., Berant, J., Le, Q. V., and Lao, N. Memory augmented policy optimization for program synthesis and semantic parsing. *Advances in Neural Information Processing Systems*, 31, 2018.

Liu, M., Yu, C.-H., Lee, W.-H., Hung, C.-W., Chen, Y.-C., and Sun, S.-H. Synthesizing programmatic reinforcement learning policies with large language model guided search. *arXiv preprint arXiv:2405.16450*, 2024.

Mikołajczyk, A. and Grochowski, M. Data augmentation for improving deep learning in image classification problem. In *2018 international interdisciplinary PhD workshop (IIPhDW)*, pp. 117–122. IEEE, 2018.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

Nye, M., Solar-Lezama, A., Tenenbaum, J., and Lake, B. M. Learning compositional rules via neural program synthesis. *Advances in Neural Information Processing Systems*, 33:10832–10842, 2020.

Odena, A., Shi, K., Bieber, D., Singh, R., Sutton, C., and Dai, H. Bustle: Bottom-up program synthesis through learning-guided exploration. *arXiv preprint arXiv:2007.14381*, 2020.

Polozov, O. and Gulwani, S. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 107–126, 2015.

Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Shi, K., Dai, H., Li, W.-D., Ellis, K., and Sutton, C. Lambdabeam: Neural program search with higher-order functions and lambdas. *Advances in Neural Information Processing Systems*, 36:51327–51346, 2023a.

Shi, K., Hong, J., Deng, Y., Yin, P., Zaheer, M., and Sutton, C. Exedec: Execution decomposition for compositional generalization in neural program synthesis. *arXiv preprint arXiv:2307.13883*, 2023b.

Sinha, K., Sodhani, S., Dong, J., Pineau, J., and Hamilton, W. L. Clutrr: A diagnostic benchmark for inductive reasoning from text. *arXiv preprint arXiv:1908.06177*, 2019.

Sobania, D., Briesch, M., and Rothlauf, F. Choose your programming copilot: a comparison of the program synthesis performance of github copilot and genetic programming. In *Proceedings of the genetic and evolutionary computation conference*, pp. 1019–1027, 2022.

Sujeeth, A. K., Brown, K. J., Lee, H., Rompf, T., Chafi, H., Odersky, M., and Olukotun, K. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):1–25, 2014.

Sullivan, K. J., Griswold, W. G., Cai, Y., and Hallen, B. The structure and value of modularity in software design. *ACM SIGSOFT Software Engineering Notes*, 26(5):99–108, 2001.

Turan, E. M. and Jäschke, J. Closed-loop optimisation of neural networks for the design of feedback policies under

uncertainty. *Journal of Process Control*, 133:103144, 2024.

Van de Ven, G. M. and Tolias, A. S. Three scenarios for continual learning. *arXiv preprint arXiv:1904.07734*, 2019.

Velickovic, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y., et al. Graph attention networks. *stat*, 1050 (20):10–48550, 2017.

Wei, C., Lee, J. D., Liu, Q., and Ma, T. Regularization matters: Generalization and optimization of neural nets vs their induced kernel. *Advances in Neural Information Processing Systems*, 32, 2019.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Wu, L., Cui, P., Pei, J., Zhao, L., and Guo, X. Graph neural networks: foundation, frontiers and applications. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 4840–4841, 2022.

Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. How transferable are features in deep neural networks? *Advances in neural information processing systems*, 27, 2014.

Zelikman, E., Harik, G., Shao, Y., Jayasiri, V., Haber, N., and Goodman, N. D. Quiet-star: Language models can teach themselves to think before speaking. *arXiv preprint arXiv:2403.09629*, 2024.

Zhang, K., Wang, D., Xia, J., Wang, W. Y., and Li, L. Algo: Synthesizing algorithmic programs with generated oracle verifiers. *Advances in Neural Information Processing Systems*, 36:54769–54784, 2023.

Zheng, W., Sharan, S., Jaiswal, A. K., Wang, K., Xi, Y., Xu, D., and Wang, Z. Outline, then details: Syntactically guided coarse-to-fine code generation. In *International Conference on Machine Learning*, pp. 42403–42419. PMLR, 2023.

Zhou, D., Schärli, N., Hou, L., Wei, J., Scales, N., Wang, X., Schuurmans, D., Cui, C., Bousquet, O., Le, Q., et al. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022.

A. DSL Program Synthesis in COOL

COOL adopts a top-down synthesis strategy that converts input partial programs with nonterminals into complete programs by applying a sequence of well-defined transformation rules.

A.1. Input Program

In the relational reasoning tasks, the input is COOL code such as Code A.1:

Code A.1: Relational Reasoning Task Input Program

```
(Wesley) is (James)s son & (Martha) is (Wesley)s daughter & (Hugh) is
(Martha)s uncle & (Hugh) is (James)s ($relation);
```

where `$` specifies the nonterminal, indicating that the DSL solver needs to synthesize a complete program that calculates the correct value for `relation` (the relationship between `Hugh` and `James`) in order to satisfy the given specification.

The symbolic task input is The input for symbolic tasks is as follows (Code A.2):

Code A.2: Symbolic Reasoning Task Input Program

```
$x^2 + 4*$x == 3;
```

Similarly, the DSL solver needs to generate a complete program that calculates the value of `x`.

A.2. Output Program

As shown in Code A.3, for family relationship reasoning tasks, the synthesized program is:

Code A.3: Relational Reasoning Task Output Program

```
relation = "son";
```

For symbolic reasoning tasks, the generated output program is shown in Code :

Code A.4: Symbolic Reasoning Task Output Program

```
Invoke Quadratic Solution Formula( a=1, b=4, c=-3, x ); // a, b, c are
coefficients in "1*x^2 + 4*x + (-3) == 0"
```

In reality, the program synthesis takes place at the intermediate representation level (see Appendix P), and Codes A.3 A.4 are provided for explanatory purposes.

A.3. Traditional DSL

A DSL, defined as a context-free grammar:

$$G = \{V, \Sigma, R, S\}, \tag{1}$$

where V is the set of non-terminal symbols, Σ is the set of terminal symbols, R is the set of rules, and S is the starting symbol (in this context, it is a partial program). The DSL's derivation process converts partial programs with nonterminal symbols into complete programs by applying given rules.

The synthesis process for traditional DSLs involves iteratively transforming partial programs into complete programs by applying a series of rules. Each partial program (p) and the corresponding rule (r) to be applied to it form a transformation pair (p, r). When a rule is applied, it modifies the syntax tree of the partial program through what we refer to as a tree

operation. The synthesis process consists of a series of transformation pairs connected by tree operations, which is known as a synthesis path or trajectory. These paths are classified into three kinds:

- **Feasible Path**: Leads to a complete program.
- **Infeasible Path**: Proven to be unable to synthesize a complete program.
- **Unfinished Path**: Still in progress.

To clarify key concepts involved in the synthesis process, we provide the following definitions of terms:

- **Tree Operation/Manipulation**: Refers to the modification of the syntax tree of a partial program during the synthesis process. It is essential for transforming partial programs into complete programs and has an associated CPU cost.
- **Transformation Pair** (p, r): A combination of a partial program and a rule to be applied. It records the explored space and possible exploration directions, requiring memory storage.
- **Synthesis Path/Trajectory**: A sequence of transformation pairs, $\{(p_0, r_0), (p_1, r_1), \dots\}$, representing the process of transforming a partial program into a complete one. Its function is to track the entire synthesis process, whether it leads to a feasible, infeasible, or unfinished path.

A.4. CoL DSL

Compared with traditional DSLs that apply rules to input programs without a clear destination to synthesize output programs, the Chain-of-Logic (CoL) allows the programmer to outline the flow of activities to synthesize the complete program from an initial partial program. For example, in Figure 3, the activity flow is:

Start $\rightarrow$ 0 Separate Relations and Genders $\rightarrow$ 1 Reason Inverse Relations $\rightarrow$ 2 Reason Indirect Relations $\rightarrow$ 3 Recombine Relations and Genders, Eliminate Irrelevant Relations $\rightarrow$ End.

Each activity in the synthesis process has a corresponding sub-DSL decomposed from the original DSL for transforming the program from one state to another.

Therefore, a CoL DSL with n activities can be defined as multiple sub-DSLs in series:

$$\text{CoL } G = \{G_1, G_2, \dots, G_n\} \tag{2}$$

A sub-DSL for activity i is defined as:

$$G_i = \{V, \Sigma, \{(r, \mathbf{h}, k) \mid \mathbf{h}[n] \neq 0 \text{ and } r \in R\}, S, f\} \tag{3}$$

where $\mathbf{h} = (h_1, h_2, \dots)$ in $(r, \mathbf{h}, k)$ represents the heuristic vector bound with r, and the components are termed as heuristic values. $h[n]$ represents the n-th component of h, which is the effective heuristic value of rule r in activity n. $\mathbf{h}[n]$ is a parameter of the sub-DSL's program synthesis algorithm f. It guides the direction of program synthesis by affecting the application decisions of the rules bound to it, thereby improving synthesis efficiency and accuracy. The specific role of the heuristic value is determined by the search algorithm f used by the DSL program synthesis. To conduct controlled variable experiments, we regard the heuristic value as a reward (negative cost) and use the A* algorithm as the search algorithm for all DSLs, which means that in activity n, under the same circumstances, the rule with a larger $\mathbf{h}[n]$ will be applied first, and the derived program will also be considered promising and will be explored further with priority.

k represents keyword(s) in a rule's specific logic, which controls the program state transition within an activity or between activities.

The Chain-of-Logic provides a comprehensive methodology for achieving fine-grained control over DSL program synthesis. This approach allows programmers to explicitly break down the synthesis process into distinct phases or activities, with each activity corresponding to a specific sub-DSL.

A.5. A* Search in Program Synthesis

During the exploration phase of program synthesis, we leverage the A* algorithm to perform the heuristic search. This algorithm is renowned for its efficacy in discrete optimization tasks, utilizing heuristic guidance to navigate the search space effectively (Hart et al., 1968). Each action or decision is associated with a specific cost in this context. By evaluating the cumulative cost of actions taken so far and the estimated costs of future actions, A* seeks to determine the path with the least overall cost. In our approach, heuristic values promoting forward progression are considered rewards. Therefore, we treat them as negative costs in calculations. Algorithm 1 illustrates the implementation details.

Algorithm 1 Search Algorithm for DSL Program Synthesis

function A* Search ($initialPartialProgram, u_2$)
  $openSet \leftarrow$ priority queue containing only the initial partial program
  $gScore[startPartialProgram] \leftarrow 0$ {cost from start}
  $fScore[startPartialProgram] \leftarrow 0$
  while $openSet \neq \emptyset$ **do**
    $currentProgram \leftarrow openSet$.pop( ) {Partial program in openSet with lowest fScore value}
    if $currentProgram$ is complete program **then**
      return Success
    end if
    for all $neighbor \in$ neighbors of $currentProgram$ **do**
      { Neighbor is obtained by applying a rule to the current program}
      $tentative\_gScore \leftarrow gScore[current] - u_2[neighbor]$
      if $tentative\_gScore < gScore[neighbor]$ **then**
        $cameFrom[neighbor] \leftarrow current$
        $gScore[neighbor] \leftarrow tentative\_gScore$
        $fScore[neighbor] \leftarrow gScore[neighbor] - u_2[neighbor]$
        if $neighbor \notin openSet$ **then**
          $openSet$.add($neighbor$)
        end if
      end if
    end for
  end while
  return Failure
end function

A.6. CoL DSL Synthesis Process

The synthesis process in CoL DSLs is conducted through multiple stages, each corresponding to a defined activity. These stages operate sequentially, gradually refining the program through well-defined transformations. The key difference from traditional DSL synthesis is that, except for the sub-DSL of the final activity, intermediate sub-DSLs are allowed to generate partial programs, which are passed on to subsequent activities for further processing.

Each stage in the CoL DSL synthesis process focuses on a specific aspect of synthesis to transform the program incrementally. For instance, in Figure 4:

- In the first activity, relations and genders are separated, breaking down the initial partial program into simpler components for easier processing.
- The second activity reasons about inverse relationships, further structuring the intermediate program by identifying and processing inverse connections.
- The third activity deals with indirect relationships, providing additional context to relationships identified in earlier stages.
- The final activity recombines relations and genders while eliminating irrelevant relations to produce a fully synthesized and optimized program.

During each activity, the synthesis process leverages heuristic values to prioritize rule application, focusing on areas that are more likely to lead to successful outcomes. Additionally, in intermediate activities, since we cannot judge whether the synthesis process is correct based on whether complete programs are generated, guidance based on heuristic values and synthesis flow control using keywords are pivotal.

B. Neural Networks in COOL

COOL has an integrated machine learning system that automatically collects generated data and conducts training and prediction tasks for neural networks in the Domain-Specific Neural Network (DSNN).

B.1. Data Collection and Combination for Training

The neural networks leverage the transformation pairs (p, r) in the synthesis paths to train various heads.

To train the neural networks in DSNN bound with a DSL for program synthesis tasks of type T, COOL builds the dataset as follows:

Task Detection Head (TDH): This head distinguishes whether the input partial program belongs to type T. This is a binary classification task. The partial programs from type T program synthesis paths are collected as positive examples (proportion: 67%), while partial programs from other synthesis paths and built-in function calls are collected as negative examples (proportion: 33%).

Search Space Prune Head (SSPH): After determining that the program is of type T, this head identifies whether the input partial program is feasible to synthesize into a complete program. This is also a binary classification task. Programs from feasible synthesis paths are collected as positive examples (proportion: 67%), while programs from infeasible synthesis paths are collected as negative examples (proportion: 33%).

Search Guidance Head (SGH): After determining that the input is a feasible type T partial program, this head generates rule features to guide the DSL solver in applying rules to the partial program. This includes a series of classification and regression tasks.

B.2. Neural Network Input

As shown in Figure 5, there are three neural networks in a DSNN. Each network (labeled A, B, and C in their sequential order) takes a partial program as input. The input partial program is represented at the intermediate representation (IR) level in the form of Three-Address Code (TAC) (see Appendix P), allowing program synthesis to be conducted without the constraints of specific DSL syntax or the machine code format of the execution platform (Sujeeth et al., 2014). The TAC is then transformed into a graph representation for input to neural networks.

In the serial coupling structure of DSNN, network B is the downstream neural network of A and uses the output of the SGH head from A as part of its input. Similarly, network C is the downstream neural network of B and uses the output of the SGH head from B as part of its input. This serial coupling enables each downstream network to accumulate the error produced by the upstream network, making incorrect predictions more obvious.

The specific input features are shown in Table 4.

B.3. Neural Network Output

The corresponding relationships between the output features and TDH, SSPH, GSH are shown in Table 5, and all neural networks produce similar outputs to allow for comparison.

B.4. Neural Network Structure

As TAC embodies both the graphical properties of a syntax tree and the sequential properties of execution, the design of the neural network must be capable of capturing these dual characteristics.

The detailed layer architecture of neural networks in DSNN is illustrated in Figure 7. The processing flow consists of the following steps:

Table 4. Input features of neural networks in DSNN. Each entry specifies the feature, its size, and the neural networks it pertains to, along with a description of its role. These features contribute to the neural network's understanding of the syntax tree's structure and semantics, aiding in the accurate synthesis of programs.

Feature	Feature Size	Neural Network	Signification
grounded	2	A, B, C	The node is in a fully specified expression.
domain	1	A, B, C	Domain of the subtask represented by the subtree where the node is located.
root	2	A, B, C	The tree representing the subtask is rooted at this node.
non-terminal	2	A, B, C	The node is a non-terminal.
type	1	A, B, C	Type of the node.
identifier	1	A, B, C	Identifier of the node.
string	1	A, B, C	The node contains a string as the immediate value.
number	1	A, B, C	The node contains a number as the immediate value.
operator	1	A, B, C	The node is an operator.
current stage	1	A, B, C	Current CoL stage (valid when this node is grounded).
operand position	3	A, B, C	Placement of nodes in a binary operation tree (left operand node, right operand node, operation node).
applied (SGH)	1	B, C	A rule is applied to the subtree rooted at this node (derived from the output feature "**jumps**" of the previous neural network).
next stage (SGH)	1	C	The CoL stage to advance to after applying the rule (derived from the output feature "**next stage**" of the previous neural network).

Table 5. Output features of neural networks in DSNN. These features provide comprehensive optimizations for CoL DSL during program synthesis, including task detection, search space pruning, and search guidance.

Feature	Feature Size	Neural Network	Signification
domain (TDH)	2	A, B, C	Relevance of task domains to DSNN.
feasibility (SSPH)	2	A, B, C	Feasibility of synthesizing the complete program.
jumps (SGH)	max_tree_depth*3	A, B, C	The path from the tree's root to the subtree's root where the rule is applied (jump left, right, or stop in each step).
next stage (SGH)	1	A, B, C	The CoL stage to advance to after applying the rule.
heuristic sign (SGH)	2	A, B, C	Sign of the rule's heuristic value.
heuristic value (SGH)	1	A, B, C	Rule's heuristic value.
expression (SGH)	2	A, B, C	Type of rule's head (expression or terminal).

1. **Embedding Node Features:** We start by employing embedding layers with learning capabilities. These layers convert categorical inputs into dense, continuous vectors, which enhances the stability and efficiency of subsequent processing layers (Hrinchuk et al., 2019).

2. **Graph Feature Extraction:** Next, we use a Graph Neural Network (GNN) to extract graph features from each line of TAC code (Drori et al., 2022; Wu et al., 2022). To adaptively extract intricate details such as node types, graph attention (GAT) layers are applied after the embedding layers (Velickovic et al., 2017).

3. **Sequential Feature Processing:** We adopt Long Short-Term Memory (LSTM) networks to capture the sequential features inherent in TAC (Chen et al., 2021; Nye et al., 2020). Recognizing the equal importance of each TAC line,

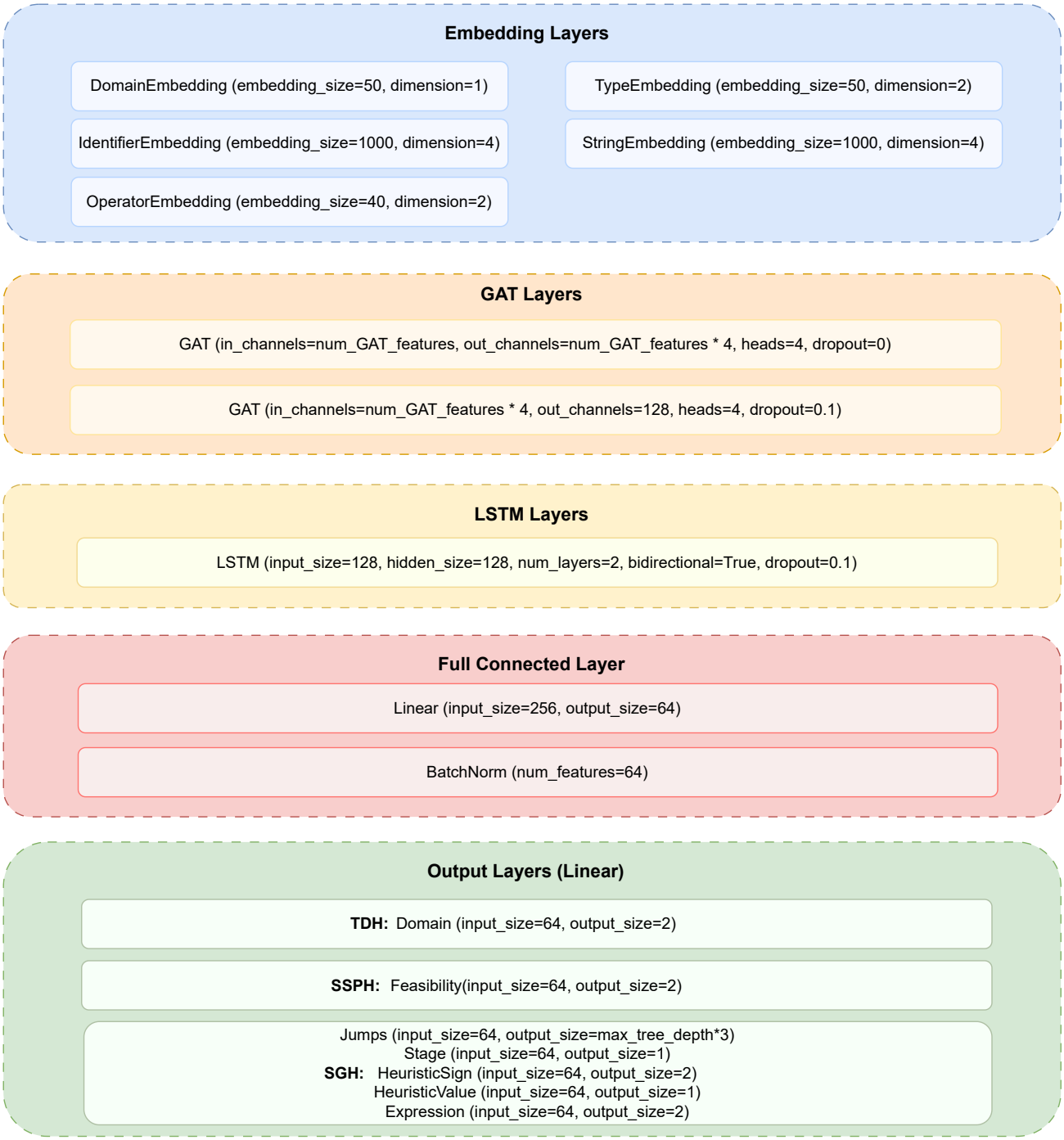

Figure 7. Layer architecture of neural networks in DSNN. Each neural network consists of embedding layers for domains, types, identifiers, strings, and operators, followed by GAT layers for tree feature extraction. LSTM layers provide sequential modeling for programs, with fully connected layers combining the outputs. Various output layers handle domain identification for task detection, feasibility judgment for search space pruning, tree jumps, stage prediction, heuristic constraint (sign and value), and constraint on the type of rule's head (expression or terminal) for search guidance.

bidirectional LSTM layers are employed following the GAT layers to enrich the contextual understanding (Huang et al.,

2015).

4. **Multi-Head Output:** Finally, the processed data is channeled through multiple output layers to prevent task interference and ensure clarity in results.

Figure 5 (right) illustrates using three neural network units arranged in series to construct the internal coupling structure of DSNN. Labeling these neural networks with A, B, and C in order of their sequence, Table 4 details the specific input features for each network: Neural network B receives its input feature "applied" from network A's output feature "jumps," while network C's input features "applied" and "next stage" are derived from the output features "jumps" and "next stage" of network B. The output features of three neural network units are consistent and comparable, Table 5 presents the output features of the neural networks.

B.5. Prediction Filtering

By comparing the output differences of the heads, we can determine whether there are possible prediction errors and filter out the prediction results. For classification tasks, we directly compare whether the outputs are the same. For regression tasks, we set a tolerance threshold (10%) for the difference.

B.6. Acting on the Synthesis Process

B.6.1. Single DSL Program Synthesis

As shown in Figure 3, the heuristic value of a rule affects its application, and the prediction results of the neural networks affect the synthesis process by correcting the heuristic value of the rules in the sub-DSL based on the output of heads:

Task Detection Head (TDH): If the output indicates that the partial program does not belong to the synthesis task that the DSL can handle, any rule application on this partial program will receive an additional negative bonus on the heuristic value. For example, $\mathbf{h}[i] = \mathbf{h}[i] - |\mathbf{h}[i]| - 10$.

Search Space Prune Head (SSPH): If the TDH output indicates that the partial program falls within the DSL and the SSPH output considers the partial program infeasible, any rule application on this partial program will receive an additional negative bonus on the heuristic value. For example, $\mathbf{h}[i] = \mathbf{h}[i] - |\mathbf{h}[i]| - 10$.

Search Guidance Head (SGH): If the TDH output indicates that the partial program falls within the DSL and the SSPH output indicates that the partial program is promising for synthesis into a complete program, then if the features of the output rule match certain rules (logical values must be equal, and numerical values must fall within a $\pm 10\%$ range), the heuristic value when applying these rules will receive a positive bonus. For example, $\mathbf{h}[i] = \mathbf{h}[i] + |\mathbf{h}[i]|$. Otherwise, it will receive a negative bonus: $\mathbf{h}[i] = \mathbf{h}[i] - |\mathbf{h}[i]| - 10$.

B.6.2. Multi-DSL Program Synthesis

The situation when multiple DSLs cooperate is similar to that of a single DSL. The difference is that for the same partial program, if at least one DSNN determines that the partial program belongs to the domain of its DSL, the DSNN bound to other DSLs, which believes that the partial program does not belong to its own domain, cannot interfere with the program synthesis at this step (as shown by signal c in Figure 5).

C. Signal Clipper

The Clipper, as illustrated in Figure 5 (left), caps signals that do not align with the DSNN guidance to zero:

$$u_2 = \begin{cases} 0 & \text{if } u_1 > 0 \text{ and current rule doesn't align with} \\ & \text{the guidance and there exists another rule in} \\ & \text{the search space that aligns with the guidance} \\ u_1 & \text{otherwise} \end{cases} \tag{4}$$

D. Experiment

The purpose of the experiment is to explore whether CoL+NNFC DSL has advantages over traditional DSL in terms of efficiency and reliability in program synthesis from the user's perspective.

"User" refers to the user of DSL, while the developer of DSL who is proficient in specific tasks is referred to as an "expert". The experiment does not study whether CoL makes the development process easier for DSL developers, as this requires a wide range of "experts" to use COOL to develop and provide feedback. At this stage, we cannot conduct this experiment.

D.1. User Input Example

The user input in the experiment is COOL code containing instructions for loading the DSL (packaged as a library) and representing the task specification.

D.1.1. Relational Tasks

Used to test the performance of program synthesis of a single DSL:

Code D.1: Relational Task User Input Example

```
//load DSL for family relationship reasoning
#load(family)

//Relational reasoning questions like (50 per batch):
(Wesley) is (James)s son & (Martha) is (Wesley)s daughter & (Hugh) is
(Martha)s uncle & (Hugh) is (James)s ($relation);
...
```

D.1.2. Symbolic Tasks

Used to test the performance of program synthesis of a single DSL:

Code D.2: Symbolic Task User Input Example

```
//load DSL for family relationship reasoning and symbolic reasoning
#load(quadratic)

//Symbolic reasoning questions like (50 per batch):
$x^2 + 4*$x == 3;
...
```

D.1.3. Multi-Domain Tasks

Used to test the performance of program synthesis when multiple DSLs are loaded at the same time:

Code D.3: Multi-Domain Task User Input Example

```
//load DSLs for relational reasoning and symbolic reasoning
#load(family)
#load(quadratic)

//Symbolic reasoning questions like (50 per batch):
$x^2 + 4*$x == 3;
...
//Relational reasoning questions like (50 per batch):  (Wesley) is (James)s
son & (Martha) is (Wesley)s daughter & (Hugh) is (Martha)s uncle & (Hugh) is
(James)s ($relation);
...
```

It should be noted that the execution of the code that does not contain the task specification is represented as a control variable in the experiment and is deducted from the final experimental results. (For example, the instructions for loading libraries)

D.2. Group Configuration

The specific experimental groups of dynamic and static experiments and their respective settings are shown in Table 6:

Table 6. Group configurations. Groups marked with ★ are the main experiments, those with ☆ are for ablation and extended experiments, and the unmarked group is the baseline.

Group	Experiment	Trained DSNN	NNFC	Inner Coupling Structure
DSL	static			
☆DSL (Heuristic)	static			
★CoL DSL	static, dynamic			
☆DSL+NN	static	✓		
☆DSL (Heuristic)+NN	static	✓		
☆ CoL DSL+NN	static	✓		
☆CoL DSL+NNFC	dynamic		✓	
☆DSL+NN (Cp)	static	✓		✓
☆DSL(Heuristic)+NN (Cp)	static	✓		✓
☆CoL DSL+NN (Cp)	static	✓		✓
☆CoL DSL+NN (Cp)	static	✓		✓
★CoL DSL+NNFC (Cp)	dynamic		✓	✓

D.3. DSL Configuration

Table 7 shows the CoL DSLs used in the two program synthesis tasks. The CoL length of the DSL in the control group is 1, indicating no activities are created. (Refer to the DSL code snippet in Appendix I and J for activity division and association rules.)

Table 7. CoL DSL configurations. The DSL for relational benchmarks has a limited search space and shorter CoL, facing challenges from numerous production rules leading to larger trees. Conversely, the DSL for symbolic benchmarks offers an unlimited search space with a longer CoL, but the many permutation rules increase the risk of cyclic rule applications.

Benchmark	**Rules**					**Length of CoL**
	Total	**Production Rules**	**Reduction Rules**	**Recursive Rules**	**Permutation Rules**	
relational	40	36	2	16	0	4
symbolic	55	17	26	3	11	7

D.4. Experiment Data

Static experiment results are shown in Figure 8:

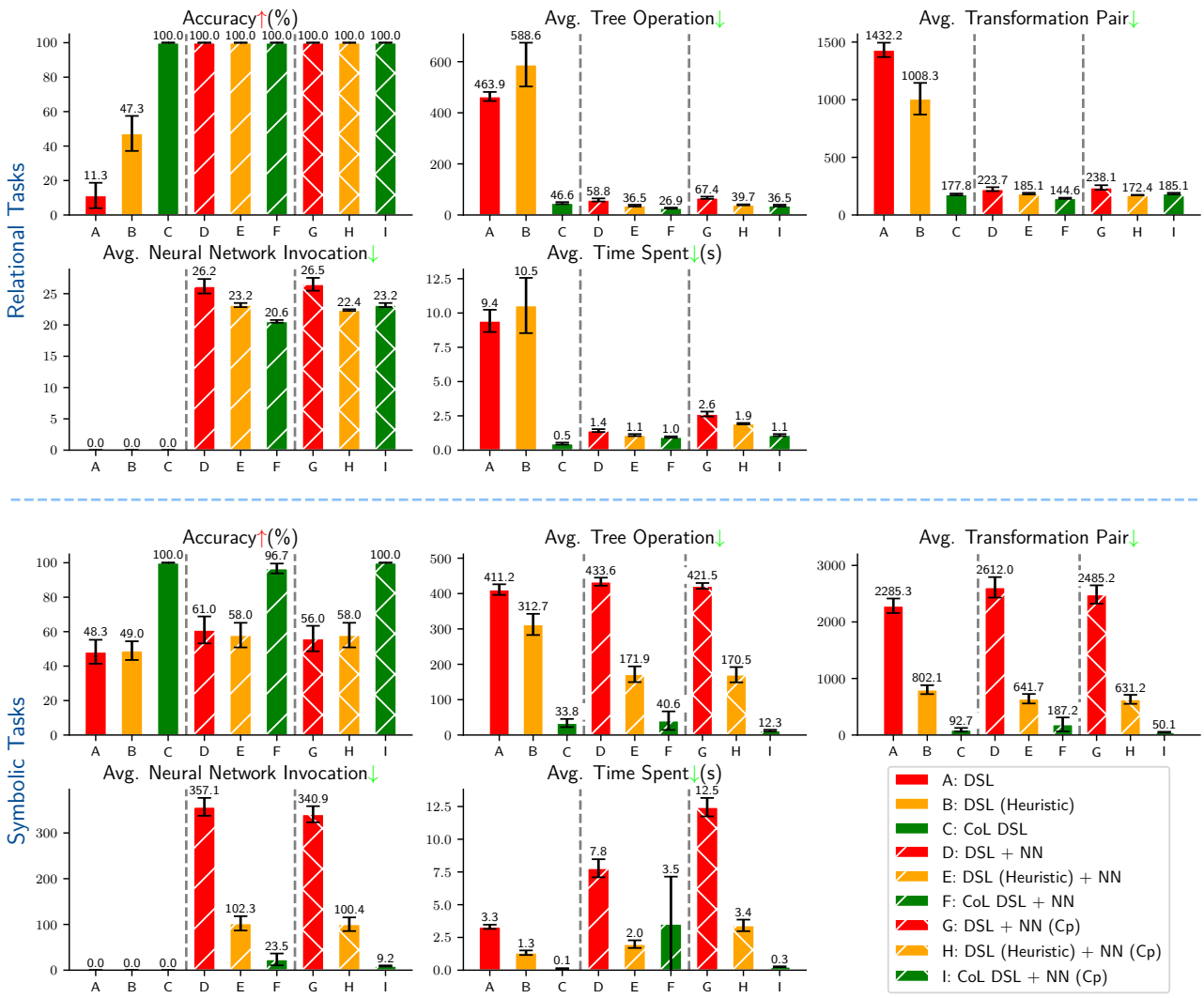

Figure 8. Static performance on relational and symbolic tasks at difficulty level A. CoL DSL-based groups outperform DSL (Heuristic) and DSL groups. Performance varies for DSNN-enhanced groups with the inner coupling structure. Error bars show 95% confidence intervals across 6 batches.

Singledomain dynamic experiment results are shown in Figure 9:

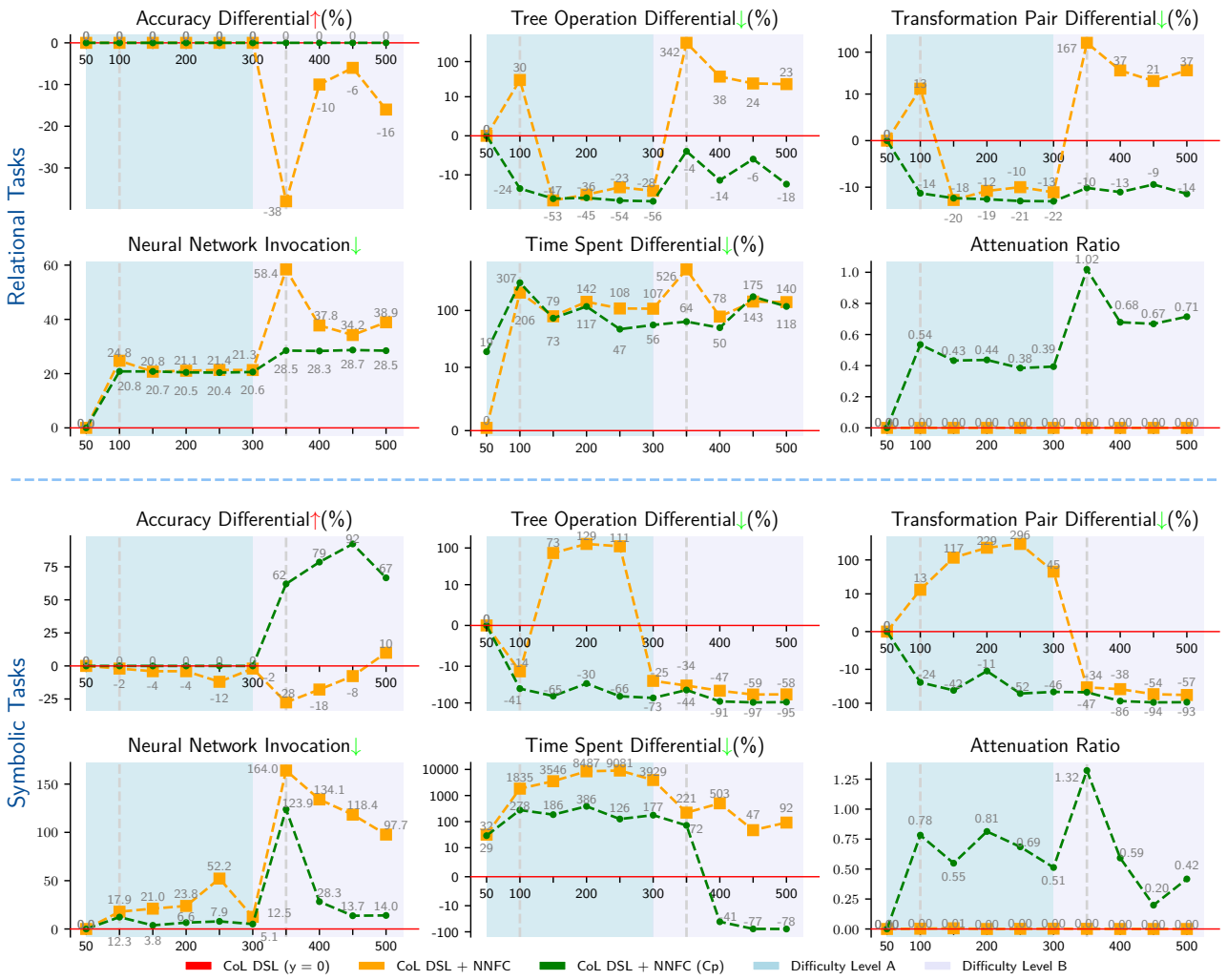

Figure 9. Dynamic performance differential to CoL DSL in singledomain tasks. The NNFC group without the inner coupling structure shows 12 accuracy declines across 20 batches, while the group with the structure shows none. Each batch consists of 50 tasks, and NNFC continuously trains DSNNs using generated data after each batch, starting from scratch.

Multidomain dynamic experiment results are shown in Figure 10:

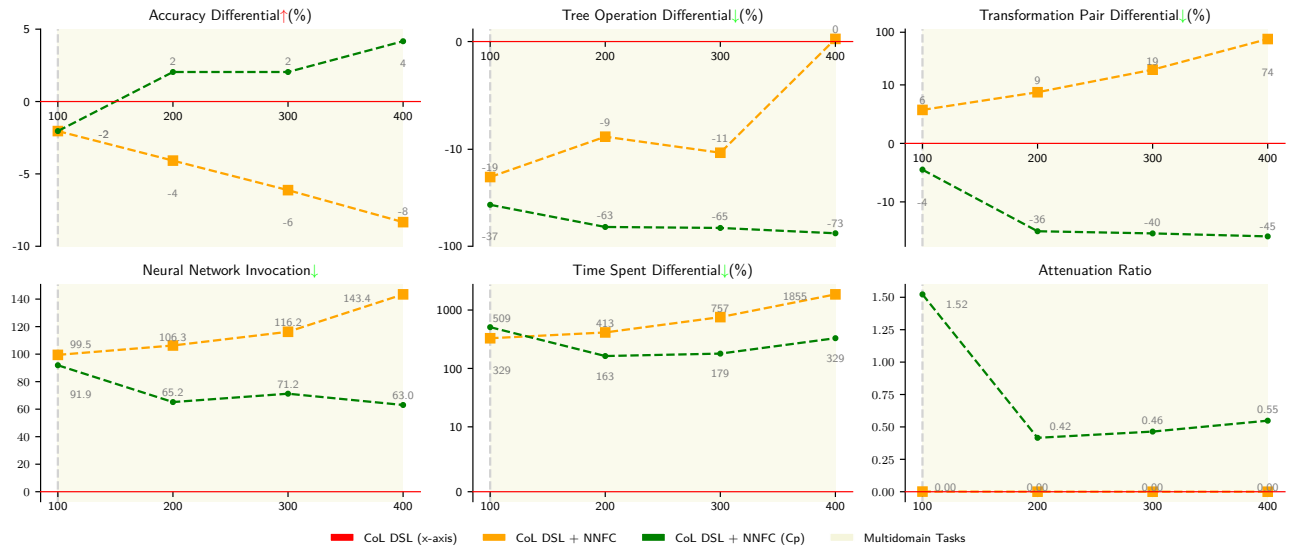

Figure 10. Dynamic performance differential to CoL DSL in multidomain tasks. The NNFC group without an inner coupling structure degrades across all 4 batches, while the group with the structure experiences degradation only in the first batch. Each batch includes 50 relational and 50 symbolic tasks, and DSNNs are continuously trained from those for tasks at difficulty level A in Figure 9.

E. Rule in CoL DSL

In addition to the heuristic vector and keywords, COOL extends the flexibility of the synthesis process by enhancing DSL rules. These enhancements are exemplified in Figure 11, which clarifies the rule introduced in Figure 2.

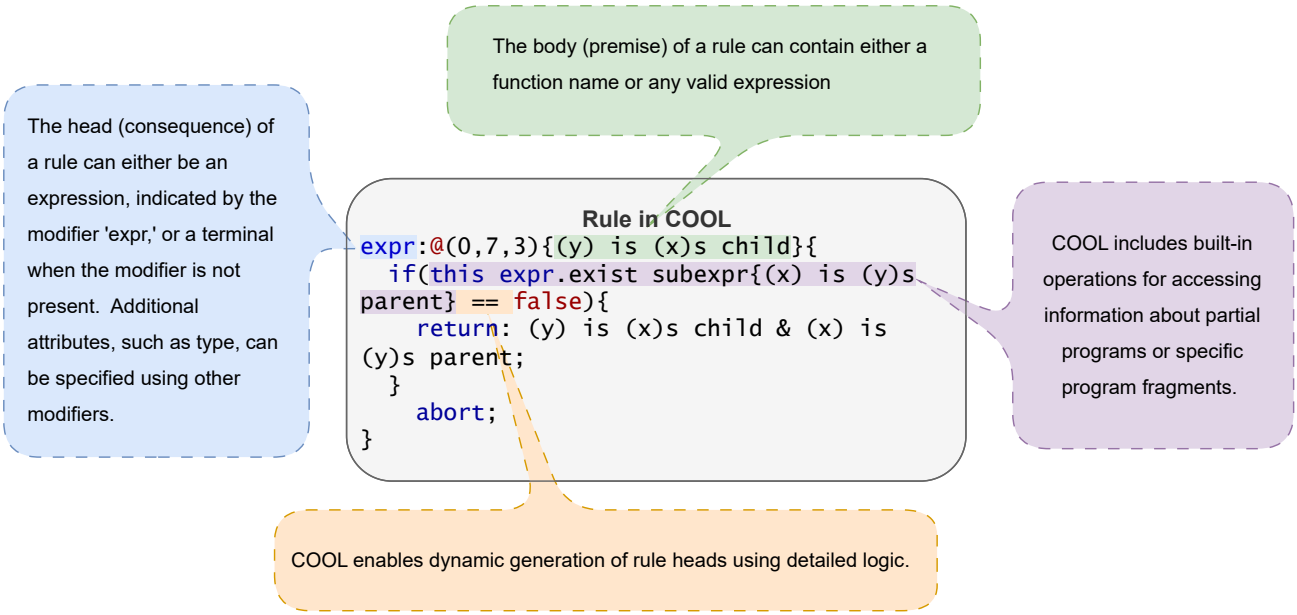

Figure 11. DSL rules in COOL. The framework allows for defining rule heads using expressions or terminals, which are enhanced with modifiers for additional attributes. Rule bodies can incorporate any valid expression or function name. Besides, COOL provides built-in operations for accessing program fragment information and facilitates dynamic rule head generation.

F. Implementation Toolchain

To fully implement the CoL DSL and adapt it to NNFC, we choose to build COOL from the ground up rather than extending existing DSL frameworks such as Xtext (Bettini, 2016) or Groovy (King, 2020). We use C++ as the primary language to meet the execution efficiency requirements for the numerous tree operations inherent in the DSL program synthesis process. For development efficiency, we utilize Lex (Lesk & Schmidt, 1975) and YACC (Johnson et al., 1975) for syntax and semantic parsing, respectively. The neural network components are implemented in Python, leveraging the PyTorch library (Imambi et al., 2021) to support machine learning tasks effectively. Table 8 shows the detailed code effort involved in developing the different components of COOL across various programming languages.

Table 8. Code Effort in COOL. Components of COOL are developed across different programming languages.

Language	Lines	Components
C++	60k	framework and CoL DSL solver
Python	3k	DSNN
Lex	1k	syntax parser
YACC	2k	semantic parsers

G. Optimization strategy

In practice, we observe that as the CoL length increases, the frequency of skipping stages rises. While skipping can lead to shorter synthesis paths and improved efficiency, it may cause task failures by omitting necessary stages. To manage this, we propose two strategies:

1. Gradient-Based Regulation: We employ gradient-based regulation, a widely used strategy in program synthesis (Cui & Zhu, 2021; Liang et al., 2018; Chaudhuri et al., 2021). By evaluating the slope or rate of change between consecutive stages, gradients help us make dynamic adjustments to synthesis paths. In our approach, we regulate skipping by applying a gradient to the heuristic values at each stage in the CoL. We encourage skipping when the heuristic gradient from one stage to the next is positive. Conversely, if the gradient is negative, we suppress skipping.

2. NNFC Regulation: Once we establish a feasible synthesis path, we can treat partial programs derived through skipping as infeasible. Then, we will utilize the feedback loop to suppress unwarranted skipping actions. However, since these partial programs might still contain feasible solutions, we need further investigation to understand and fully leverage the potential impact of this data.

In our experiments, we prioritize accuracy by suppressing skipping behavior, ensuring essential stages are included in synthesis paths.

H. Future Work

In future work, we aim to enhance the capability of the COOL framework by exploring the implementation of CoL and NNFC in more complex scenarios, such as managing dependencies among DSL libraries and object-oriented development. We plan to facilitate community collaboration by developing more DSL libraries to expand COOL's applications. Additionally, we are interested in integrating COOL with language models. As these models evolve, ensuring ethical and accurate reasoning becomes increasingly crucial (Jacovi & Goldberg, 2020; Chen et al., 2022; Li et al., 2022). The COOL framework, including CoL's constraints on rule application and NNFC's structured agent interactions, helps to enhance reasoning faithfulness, preventing harmful reasoning logic. We hope our work will serve as a reliable bridge for interaction and understanding between human cognitive processes and language model reasoning.

I. CoL DSL for Relational Tasks (Source Code)

We present only the specific code for the CoL DSL group, while the code for the DSL and DSL (Heuristic) groups, referenced in Table 6, is not displayed. This omission is because their differences from the CoL DSL group are confined to their heuristic vectors. In both the DSL and DSL (Heuristic) groups, the heuristic vectors have a dimension of 1. However,

the DSL group employs a fixed heuristic value of -1, whereas the DSL (Heuristic) group utilizes variable values. The experimental codes are presented concisely, showcasing only the framework. Please refer to the attached supplementary materials for the complete content.

```
//1 Separate Relations and Genders
expr:@(9){(a) is (b)s grandson}{
    return:(a) is male &  (a) is (b)s grandchild & (b) is (a)s grandparent;
}
...

//2 Reason Inverse Relations
expr:@(0,7,3){(a) is (b)s grandchild}{
    if(this expr.exist subexpr{(b) is (a)s grandparent} == false){
        return: (a) is (b)s grandchild & (b) is (a)s grandparent;
    }
    abort;
}
...

//3 Reason Indirect Relations
expr:@(0,0,5){(a) is (b)s sibling}{
    placeholder:p1;
    while(this expr.find subexpr{(p1) is (a)s sibling}){
        if(this expr.exist subexpr{(p1) is (b)s sibling} == false && p1 != b){
return: (a) is (b)s sibling & (p1) is (b)s sibling;
        }
        p1.reset();
    }
    p1.reset();
    while(this expr.find subexpr{(p1) is (a)s parent}){
        if(this expr.exist subexpr{(p1) is (b)s parent} == false){
return: (a) is (b)s sibling & (p1) is (b)s parent;
        }
        p1.reset();
    }
    p1.reset();
    while(this expr.find subexpr{(p1) is (a)s pibling}){
        if(this expr.exist subexpr{(p1) is (b)s pibling} == false){
return: (a) is (b)s sibling & (p1) is (b)s pibling;
        }
        p1.reset();
    }
    p1.reset();
    while(this expr.find subexpr{(p1) is (a)s grandparent}){
        if(this expr.exist subexpr{(p1) is (b)s grandparent} == false){
return: (a) is (b)s sibling & (p1) is (b)s grandparent;
        }
        p1.reset();
    }
    p1.reset();
    abort;
}
...
```

```
//4 Recombine Relations and Genders, Eliminate Irrelevant Relations
expr:@(0,0,0,8){(a) is (b)s ($relation)}{
   //immediate family
   placeholder:p1;
   while(this expr.find subexpr{(a) is (b)s grandchild}){
       if(this expr. exist subexpr{(a) is male}){
return: $relation == "grandson";
       }
       if(this expr.exist subexpr{(a) is female}){
return:$relation == "granddaughter";
       }
       p1.reset();
   }
   p1.reset();
   while(this expr.find subexpr{(a) is (b)s child}){
       if(this expr. exist subexpr{(a) is male}){
return: $relation == "son";
       }
       if(this expr.exist subexpr{(a) is female}){
return:$relation == "daughter";
       }
       p1.reset();
   }
   ...
   abort;
}
...
expr:@(0,0,0,10){a & ($b == c)}{
   return:b == c;
}
...
```

J. CoL DSL for Symbolic Tasks

```
// Common Transformations
expr:@(2,2,2,2,2){0+#a}{
   return:a;
}
expr:@(2,2,2,2,2){#a+0}{
   return:a;
}
...

// 1 Expand Square Terms
expr:@(5,0,0,0){(#?a + #?b)^2}{
   return:a^2+2*a*b+b^2;
}
expr:@(5,0,0,0){(#?a - #?b)^2}{
   return:a^2+(-2)*a*b+b^2;
}
expr:@(6,0,0,0){(#a*#b)^2}{
   return:a^2*b^2;
}
...
```

```
// 2 Expand Bracketed Terms
expr:@(0,4,0,0,0){#?a-(#?b+#?c)}{
   return:a-b-c;
}
expr:@(0,3.8,0,0,0){(#?b+#?c)*#?a}{
   return:b*a+c*a;
}
...

// 3 Extract Coefficients
expr:@(0,0,5,0){$x*a}{
   return:a*x;
}
expr:@(0,0,4.8,0){(immediate:a*$x)*(immediate:b*$x)}{
   new:tmp = a*b;
   return:tmp*x^2;
}
expr:@(0,0,4.6,0){$x*(a*$x)}{
   return:a*x^2;
}
...

// 4 Re-Express Negative Coefficients
expr:@(0,0,0,3.5,0){#a-$x}{
   placeholder:p1;
   placeholder:p2;
   if(x.exist subexpr{p1*p2}){
       abort;
   }
   return:a+(-1)*x;
}
expr:@(0,0,0,3.7,0){#a-immediate:b*$x}{
   new:tmp = 0 - b;
   return:a+tmp*x;
}
...

//5 Arrange Terms in Descending Order, Combine Like Terms
expr:@(0,0,0,0,3){immediate:a*$x+immediate:b*$x}{
   new:tmp = a+b;
   return:tmp*x;
}
expr:@(0,0,0,0,2.8){a1*$x+a2*$x^2}{
   return:a2*x^2+a1*x;
}
...

//6 Convert to Standard Form
expr:@(0,0,0,0,0,2.5){a*$x^2+b*x == #d}{
   return: a*$x^2+b*x + 0 == d;

}
expr:@(0,0,0,0,0,2.5){b*$x == $d}{
```

```
    if(d.exist subexpr{x^2}){
        return: 0*x^2 + b*x + 0 == d;
    }else {
        abort;
    }
}
expr:@(0,0,0,0,0,-4){$a==$b}{
    return:b==a;
}
...

//7 Apply Solution Formula
@(0,0,0,0,0,0,0,10){a*$x^2+b*x+c==0}{
    if(b^2-4*a*c<0){
        x="null";
    }
    else {
        new:x1=(-b+(b^2-4*a*c)^0.5)/(2*a);
        new:x2=(-b-(b^2-4*a*c)^0.5)/(2*a);
        x={x1,x2};
    }
};
```

K. Relational Tasks at Difficulty Level A (Source Code)

```
#load(family) // Load the CoL DSL library for Relational Tasks
new:relation = "";
// [Francisco]'s brother, [Wesley], recently got elected as a senator. [Lena] was
↪  unhappy with her son, [Charles], and his grades. She enlisted a tutor to help
↪  him. [Wesley] decided to give his son [Charles], for his birthday, the latest
↪  version of Apple watch.
// Ans: (Francisco) is (Lena)s brother
new:Lena = "Lena";
new:Charles = "Charles";
new:Wesley = "Wesley";
new:Francisco = "Francisco";
(Charles) is (Lena)s son & (Wesley) is (Charles)s father & (Francisco) is
↪  (Wesley)s brother & (Francisco) is (Lena)s ($relation);
relation-->"#FILE(SCREEN)";

// [Clarence] woke up and said hello to his wife, [Juanita]. [Lynn] went shopping
↪  with her daughter [Felicia]. [Felicia]'s sister [Juanita] was too busy to
↪  join them.
// Ans: (Lynn) is (Clarence)s mother-in-law
new:Clarence = "Clarence";
new:Juanita = "Juanita";
new:Felicia = "Felicia";
new:Lynn = "Lynn";
(Juanita) is (Clarence)s wife & (Felicia) is (Juanita)s sister & (Lynn) is
↪  (Felicia)s mother & (Lynn) is (Clarence)s ($relation);
relation-->"#FILE(SCREEN)";
...
```

L. Relational Tasks at Difficulty Level B (Source Code)

```
#load(family) // Load the CoL DSL library for Relational Tasks
new:relation = "";
// [Antonio] was happy that his son [Bernardo] was doing well in college.
↪  [Dorothy] is a woman with a sister named [Tracy]. [Dorothy] and her son
↪  [Roberto] went to the zoo and then out to dinner yesterday. [Tracy] and her
↪  son [Bernardo] had lunch together at a local Chinese restaurant.
// Ans: (Roberto) is (Antonio)s nephew
new:Antonio = "Antonio";
new:Bernardo = "Bernardo";
new:Tracy = "Tracy";
new:Dorothy = "Dorothy";
new:Roberto = "Roberto";
(Bernardo) is (Antonio)s son & (Tracy) is (Bernardo)s mother & (Dorothy) is
↪  (Tracy)s sister & (Roberto) is (Dorothy)s son & (Roberto) is (Antonio)s
↪  ($relation);
relation-->"#FILE(SCREEN)";

// [Bernardo] and his brother [Bobby] were rough-housing. [Tracy], [Bobby]'s
↪  mother, called from the other room and told them to play nice. [Aaron] took
↪  his brother [Bernardo] out to get drinks after a long work week. [Tracy] has
↪  a son called [Bobby]. Each day they go to the park after school. ans: (Bobby)
↪  is (Aaron)s brother
new:Aaron = "Aaron";
new:Bernardo = "Bernardo";
new:Bobby = "Bobby";
new:Tracy = "Tracy";
(Bernardo) is (Aaron)s brother & (Bobby) is (Bernardo)s brother & (Tracy) is
↪  (Bobby)s mother & (Bobby) is (Tracy)s son & (Bobby) is (Aaron)s ($relation);
relation-->"#FILE(SCREEN)";
...
```

M. Symbolic Tasks at Difficulty Level A (Source Code)

```
#load(quadratic) // Load the CoL DSL library for Symbolic Tasks
new:x = 1;
6*$x^2 == 3*x - 7;
x-->"#FILE(SCREEN)";
($x - 6)*(x + 3) == x;
x-->"#FILE(SCREEN)";
...
```

N. Symbolic Tasks at Difficulty Level B (Source Code)

```
#load(quadratic) // Load the CoL DSL library for Symbolic Tasks
new:x = 1;
$x*($x + 11) == 16*($x + 22);
x-->"#FILE(SCREEN)";
$x*(36*$x + 50) - 11*(19 - 30*$x) == $x^2;
x-->"#FILE(SCREEN)";
...
```

O. Multidomain Tasks (Source Code)

```
#load(quadratic) // Load the CoL DSL library for Symbolic Tasks
#load(family) // Load the CoL DSL library for Relational Tasks
new:x = 1;
$x^2 - 4*$x == 6;
x --> "#FILE(SCREEN)";
...
new:relation = "";
// [Dolores] and her husband [Don] went on a trip to the Netherlands last year.
↪  [Joshua] has been a lovely father of [Don] and has a wife named [Lynn] who is
↪  always there for him.
// Ans: (Dolores) is (Lynn)s daughter-in-law
new:Lynn = "Lynn";
new:Joshua = "Joshua";
new:Don = "Don";
new:Dolores = "Dolores";
(Joshua) is (Lynn)s husband & (Don) is (Joshua)s son & (Dolores) is (Don)s wife &
↪  (Dolores) is (Lynn)s ($relation);
relation-->"#FILE(SCREEN)";
...
```

P. COOL Intermediate Representation


```
"codeTable": [
    {
        "boundtfdomain": "",
        "grounded": false,
        "operand1": {
            "argName": "x",
            "argType": "identifier",
            "changeable": 1,
            "className": "",
            "isClass": 0
        },
        "operand2": {
            "argName": "2",
            "argType": "number",
            "changeable": 0,
            "className": "",
            "isClass": 0
        },
        "operator": {
            "argName": "^",
            "argType": "other"
        },
        "result": {
            "argName": "1418.4",
            "argType": "identifier",
            "changeable": 1,
            "className": "",
            "isClass": 0
        },
        "root": false
    },
```

```
    ...
]
```

Q. Related Work

Neural Search Optimization: Neural networks are key for optimizing search in program synthesis. Projects like (Kalyan et al., 2018; Zhang et al., 2023) and (Li et al., 2024) use neural networks to provide oracle-like guidance, while Neo (Feng et al., 2018), Flashmeta (Polozov & Gulwani, 2015), and Concord (Chen et al., 2020) prune search spaces with infeasible partial programs. COOL employs both strategies to enhance efficiency.

Multi-step Program Synthesis: Chain-of-Thought (CoT) (Wei et al., 2022) enhances LLMs by breaking tasks into subtasks. Projects like (Zhou et al., 2022; Shi et al., 2023b) and (Zheng et al., 2023) use this in program synthesis. Compared to CoT, which directly decomposes tasks, CoL does so indirectly by constraining rule applications.

Reinforcement Learning: Reinforcement learning improves neural agents in program synthesis through feedback, as seen in (Eberhardinger et al., 2023; Liu et al., 2024; Bunel et al., 2018), Concord (Chen et al., 2020), and Quiet-STaR (Zelikman et al., 2024). NNFC similarly refines control flow but serves an auxiliary role for programmer strategies in synthesis rather than dominating it.