Prompt2DeModel: Declarative Neuro-Symbolic Modeling with Natural Language

Hossein Rajaby Faghihi[1], **Aliakbar Nafar**[1], **Andrzej Uszok**[2], **Hamid Karimian**[1], and **Parisa Kordjamshidi**[1]

[1] Michigan State University
[2] Florida Institute for Human and Machine Cognition
{rajabyfa, nafarali, karimian, kordjams}@msu.edu, auszok@ihmc.org

Abstract. This paper presents a conversational pipeline for crafting domain knowledge for complex neuro-symbolic models through natural language prompts. It leverages large language models to generate declarative programs in the DomiKnowS framework. The programs in this framework express concepts and their relationships as a graph in addition to logical constraints between them. The graph, later, can be connected to trainable neural models according to those specifications. Our proposed pipeline utilizes techniques like dynamic in-context demonstration retrieval, model refinement based on feedback from a symbolic parser, visualization, and user interaction to generate the tasks' structure and formal knowledge representation. This approach empowers domain experts, even those not well-versed in ML/AI, to formally declare their knowledge to be incorporated in customized neural models in the DomiKnowS framework.

Keywords: Neuro-Symbolic AI · Code Generation · Constraint Integeration · Declarative Programming · Language to Code · Generative AI · Large Language Models.

1 Introduction

With the rise of large language models (LLM), the community is getting closer to the dream of natural language programming interfaces [30, 36]. Nonetheless, developing systems capable of seamlessly and accurately interpreting and executing complex programming tasks using natural language remains a challenging goal. Conversely, declarative programming has endeavored to express programs at the high-level problem specifications [24]. Within this programming paradigm, emphasis on the logic of what needs to be accomplished, as opposed to the intricacies of implementation, reduces the gap between natural and declarative languages. Inspired by this idea, our research envisions a natural language interface for a *declarative learning-based programming* framework [35, 21, 20]. Within the interface, developers articulate their learning tasks using natural language, where the underlying system translates them into code in the declarative language of DomiKnowS [9]. DomiKnowS further establishes connections between symbolic

knowledge and neural learning components (currently designed in PyTorch), facilitating the integration of domain knowledge into deep learning models.

Given the rise of LLMs and their ability to be universal computational models, the need for customized models can be debatable. However, we follow two main arguments that drive the need for developing customized learning-based models: 1) Despite their impressive performance across various tasks [48], Large Language Models (LLMS) trail behind smaller, domain-specific models [43, 42, 19, 26], and 2) incorporating domain knowledge into models, apart from the performance gains, improves the interpretability and reliability of the models by adhering to domain constraints [31, 34, 45]. Imposing domain knowledge has the potential to leverage explicit alignment of the models with safety, fairness, and ethical constraints [14].

In this work, we leverage the progress on specialized frameworks for knowledge integration [9, 1, 25, 16], particularly those with declarative interfaces [22] and the advancements in tailoring neural model architectures using LLMs [42].

Prior research on tailoring deep learning models with natural language prompts has either focused on fine-tuning simple text-to-text models [42] or composing complex architectures during inference [38]. However, our proposal facilitates the development of complex deep-learning models and enables further fine-tuning them on the task-specific data within one framework.

Adapting LLMs to generate DomiKnowS declarative programs presents several challenges. These include limited resources for fine-tuning (There are few training samples available in the declarative language of DomiKnowS, unlike popular programming languages such as Python or frameworks such as PyTorch.), the requirement to adjust LLMs to a new coding style specific to DomiKnowS, the complexity of translating domain knowledge into First-Order Logic (FOL) statements [13], let alone the specific DomiKnowS' Python-based FOL language, and the potential lack of user familiarity with the DomiKnowS language, hindering meaningful feedback in a human-in-the-loop process.

To tackle these issues, we propose an interactive pipeline where we utilize an underlying LLM with techniques such as prompt templates for user interactions, dynamic few-shot in-context learning, intermediary mapping of natural language (NL) to FOL statements, and iterative refinement based on feedback from symbolic semantic/syntactic verification functions to generate the final declarative modeling code. The pipeline's interface [3] can be accessed online.

2 Related Research

We aim to facilitate the development of neuro-symbolic models leveraging background knowledge expressed via logical constraints. Using logical knowledge, explored both during inference [11, 37, 12, 33] and training [15, 28, 44] of neural networks, has been encapsulated in other libraries like DeepProbLog [25], PyLon [1], and Scallop [16]. DomiKnowS [9] provides a declarative interface for

---

[3] To access the demo, use `https://hlr-demo.egr.msu.edu/` (Please email authors for login access.). For demonstration, see `https://youtu.be/9q9PvH3dJKE`

defining knowledge and computational units, allowing seamless integration of knowledge using various underlying techniques. While DomiKnowS represents a significant advancement, our proposal seeks to enhance the declarative interface by enabling the direct use of task descriptions in natural language form.

The existing endeavors in code generation [6, 41] deploy large models in few-shot or zero-shot settings [47, 4] or fine-tune them for specific tasks [30, 36]. These approaches demand data resources similar to the target task in pre-training/fine-tuning, which are inaccessible to us. Our work emphasizes using natural language interfaces to generate code that supports the development of neuro-symbolic models instead of conventional code in popular programming languages.

Recent research on generating neural architecture from natural language prompts has focused on either composing models during inference [38], generating training data [46], or training straightforward LLM architectures without extensive modeling efforts [42]. In contrast, we map natural language prompts to formal representations to create complex declarative neuro-symbolic models.

Our proposal aligns with recent research on the trajectory of deriving formal representations from language models, where recent research has delved into such translation to formalism suitable for underlying engines for logical inference and constraint optimization domains [31], as well as guided generation for consistent reasoning [32] or causal [18] and probabilistic reasoning [27]. However, we propose an interactive pipeline for extracting representations, facilitating the development of neural architectures intricately connected to symbolic concepts. Our task includes an additional complexity, as the target formalism is specific to DomiKnows and not seen during LLMs' pretraining.

Lastly, our work employs techniques for sampling LLMs' output, emphasizing that correct response can be obtained with a sufficient number of samples [29]. Techniques like majority voting, post-pruning, and filtering by test cases have enhanced accuracy in large-scale sampling [5, 2, 7]. Due to the dynamic nature of tasks and neural models, defining test cases is impossible for our use case. Instead, we employ an iterative feedback system that provides execution errors to the language model to reflect on. In contrast to self-refinement research [17], our work introduces an external symbolic parser for the semantic evaluation of expected output structures, offering insights for iterative error correction.

3   DomiKnowS Interface

Our tool aims to translate natural language descriptions of structured prediction tasks into corresponding representations within the DomiKnowS framework. DomiKnowS programs include three components: knowledge declaration (domain/ task structure and constraints), model declaration (computational units), and program execution (knowledge integration, learning, and inference). This paper focuses on the knowledge declaration, as it often requires input from domain experts who may lack familiarity with neural architectures or programming languages and benefit significantly from a natural language interface. The knowledge declaration stage results in a concept graph that represents the structure of

```python
with Graph('NLI_Graph') as graph:
    pair = Concept(name='pair')
    premise = Concept(name='premise')
    hypothesis = Concept(name='hypothesis')
    (arg_p, arg_h) = pair.has_a(premise, hypothesis)
    # 3 classes in a multi-class setting
    nli_lb = pair(
        name="nli_class", values=[
            "entailment", "neutral", "contradiction"])
    sym = Concept(name="symmetric")
    s_p1, s_p2 = sym.has_a(arg1=pair, arg2=pair)
    ### Symmetric constraint
    #### Ent(X1, X2) => !CON(X2, X1)
    ifL(andL(nli_lb.entailment('x'),
            existsL(sym('s', path=('x', sym.reversed)))),
        andL(notL(nli_lb.contradiction(
                    path=('x', sym.reversed, s_p2)))))
```

Fig. 1. Parts of the concept graph generated for the Natural Language Inference task.
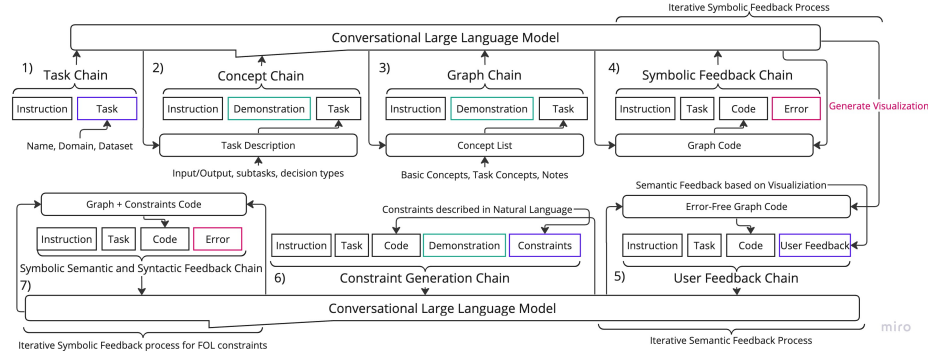

Fig. 2. Overview of the pipeline: The purple, green, and red parts represent direct user inputs, dynamic in-context examples, and execution/parser feedback, respectively.

tasks and incorporates domain-specific logical constraints. A snippet of code in DomiKnowS, illustrating the representation of the Natural Language Inference (NLI) task graph structure and logical constraints, is provided in Figure 1.

3.1   Graph Structure

The graph illustrates the input-output structure and dependencies of the task. It includes nodes representing different concepts and edges denoting their relationships. In NLP tasks, input concepts consist of sentences, paragraphs, phrases, words, and tokens. Vision tasks involve concepts related to images and bounding boxes. In structured prediction, the output concepts are task labels. Each decision concept needs to be anchored to an input concept. For example, the concept of *sentence_ class* is connected to the *sentence* concept, and *named entity* tags are associated with the *phrase* concept. DomiKnowS uses "is_a" to represent connections between a concept and its parent, "contains" for relations between a concept and its same-type children, and "has_a" for many-to-many relations.

3.2   Logical Constraints

DomiKnowS introduces a domain-specific Python-based language that employs concepts, edges, variables, and logical operators to declare dependencies in first-order logic between concepts, which defines the tasks' domain knowledge in the form of constraints (See Fig 1 for a sample code.). Significantly, the syntax for specifying first-order logic constraints in DomiKnowS is not observed during the pre-training of large language models, presenting a difficulty in generating outputs that adhere to this format in a zero or few-shot setting.

4   Natural Language to Knowledge Declaration

Our interactive interface employs prompt templates from the LangChain framework[4] and GPT-3.5-turbo. At each stage, user input fills in a predefined template that the model consumes to generate the information needed for subsequent steps. We adjust the conversation history to prevent excessive context accumulation, especially where we have an iterative process.

Figure 2 provides an overview of the pipeline. The model can consume a series of underlying natural language instructions, in-context demonstrations, and execution feedback, in addition to the user input and information from prior interactions at each step. The process begins by gathering basic information, such as the task's name, domain, and dataset name (see step 1 in the figure). Subsequently, the model formulates a clear task description, including the task's input/output and decision types. Users have the flexibility to modify this description. Next, the model compiles a list of concepts (nodes in the graph) representing the task's structure (the input/output concepts, decision types, and more; See step 2). Users can modify items on this list, which is the foundation for generating an initial concept graph (in Python; See step 3).
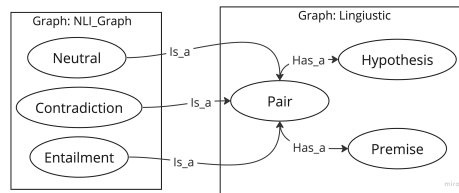

Fig. 3. The visualized graph for the NLI task.

The generated code might not be accurate/executable due to the model's lack of familiarity with DomiKnowS' language syntax. Therefore, we implement a loop for parsing the code and detecting the errors while providing the LLM with feedback for refinement. At each step, the model receives errors from the code's parse and execution, which evaluates the graph structure and uses this feedback

---

[4] https://www.langchain.com/

to refine the code (Step 4). We also seek user feedback once confident in the code's quality. As users need not be familiar with DomiKnowS' coding structure, they provide feedback based on the visualization of the code, representing the nodes and edges in the graph (Step 5). Figure 3 visualizes the graph for the NLI task. Subsequently, users provide the task constraints in natural language. The model aims to translate these expressions into DomiKnowS' Python-based FOL language, which differs somewhat from standard FOL syntax (see step 6). We employ techniques such as dynamic in-context learning, intermediary FOL mapping, and iterative refinement to assist the model in this translation (step 7). Here, we briefly discuss some of the important techniques used in the pipeline.

Sampling Strategy We draw multiple samples from the underlying LLM at each step and select the most accurate response through user feedback or automated metrics, such as error counts from the execution. Additionally, if some samples do not have any errors generated by the symbolic processing engine, we prune out the remaining erroneous samples after a certain amount of iterations.

Dynamic In-Context Retrieval Our pipeline involves in-context demonstrations for various tasks in the NLP and vision domains at each step. To enhance model execution speed and minimize noise from irrelevant examples, we selectively include a subset of in-context demonstrations relevant to the target task. We achieve this by creating a vector database of in-context demonstration representations using the OpenAI Embedding service and identifying the most relevant ones for inference based on cosine similarity. This approach reduces computational costs and noise in LLM's generation.

Iterative Symbolic Refinement We propose an iterative refinement process in response to LLMs' inability to execute code directly to ensure its sanity. Leveraging the representations of structure and constraints in DomiKnowS, we develop syntactic and semantic evaluation parsers to provide feedback. This symbolic parser generates graph structure and constraints, pinpointing semantic and syntactic errors. The parser is a custom-designed algorithm that processes the written code in DomiKnowS and finds errors and inconsistencies in its logic and syntax. The parser's feedback serves as instructions for the LLM to refine its initial answer, addressing the challenge of limited prior exposure to DomiKnowS syntax and ensuring accurate responses.

FOL Mapping To enhance constraint generation precision, we utilize an intermediary FOL mapping process. The model translates natural language descriptions into FOL statements and then converts them into DomiKnowS FOL syntax, leveraging the model's exposure to FOL statements during pre-training. Logical predicates/arguments are symbolically extracted and included in the input to help in the prediction. FOL mapping assists LLMs in capturing semantics through FOL representations, enabling users to validate constraints based on these statements, assuming perfect alignment between the two representations.

5 Evaluation and Analysis

We assessed our method for various tasks in NLP, Vision, and Constraint Satisfaction Problems (CSP) using both automatic (similarity-based) and human judgments.

5.1 Automatic Evaluation

We devise a set of metrics customized for each step of the process to evaluate the expected output at each stage. This evaluation covers a benchmark of 14 tasks, spanning Entity/Relation Extraction [40], Hierarchical Classification [23], Sudoku, Procedural Reasoning [10, 8], Causal Reasoning [39], Natural Language Inference [3], and more. At each stage, we compare with the ground truth input from the dataset for evaluation. A distinct evaluation approach for end-to-end mode, incorporating human intervention, will be discussed in the next section.

Table 1. Summary of automatic evaluation results for generating task components (description, concept list, initial graph, and post-fix graph). In full in-context, 4 demonstrations are used, while Dynamic has 1. E stands for Error. n/N indicates successful graph parsing of n out of N tasks without structural issues. x N, y E represents average x differences in nodes and y differences in edges compared to the ground truth. When using more than 1 sample, the result of the best sample is reported, mimicking human intervention in selecting the optimal answer. * the values are calculated by BertScore.

| In-Context | # Samples | Task Desc* | Concept Diff | Initial Graph | | Refined Graph |
				E-Free	Diff	E-Free
Full	1	73.9%	20.8%	12/14	0.8 N, 3.3 E	14/14
Full	3	**79.3%**	**20.6%**	**13/14**	**0.76 N, 2 E**	14/14
Dynamic	1	73.9%	26.8%	10/14	2.1N, 2.6 E	14/14
Dynamic	3	**79.3%**	26.8%	**13/14**	1.5 N, 2.1 E	14/14

Task, Concept, and Graph Generation Table 1 reports the results using automatic metrics that evaluate task description, concepts, and initially generated and corrected graphs. The outcomes demonstrate the success of our proposed tool in generating the information at each step. The large number of error-free tasks after refinement and the higher performance of the model using sampling indicate the effectiveness of both of these techniques in enhancing the model's performance. Notably, generating the concept list shows a higher error rate, especially in tasks like image classification in CIFAR-100 [23], where decision space is large, and the model often falls short by adding 'etc.' to the label set instead of listing all classes. The limited differences in predicted nodes and edges compared to the ground truth suggest that human intervention can effectively address graph errors by instructing the model to add or remove small components. Although dynamic retrieval can help reduce the cost and increase the speed of the process, it slightly hurts the model's performance at these phases.

Table 2. Results for constraint generation and the ratio of error types in resolved/unresolved errors during iterative feedback. Four in-context demonstrations are available in Full and one in Dynamic. E stands for Error. 'i D, j I' denotes i tasks are correct without the need for feedback and j are fixed after the iterative feedback. i% P, j% S indicates that i% of resolved/unresolved errors are Python errors, while j% are errors caught and reported in a custom prompt with our symbolic parser. In multi-sample settings, an error-free task has at least one sample with no errors.

Setting	In-Context	# Samples	E-Free Tasks	% E in Resolved E	% E in Unresolved E
Direct	Full	1	2 D, 6 I	53% P, 47% S	79% P, 21% S
		3	4 D, 9 I	48%P, 52% S	66% P, 34% S
	Dynamic	1	5 D, 5 I	70% P, 30% S	70% P, 30% S
		3	6 D, 7 I	41% P, 59% S	64%P, 36%S
FOL	Full	1	8 D, 0 I	90% P*, 10% S	80% P, 20%S
		3	13 D, 0 I	77%P, 23% S	82%P, 18%S
	Dynamic	1	5 D, 2 I	65%P, 35% S	80% P, 20% S
		3	10 D, 1 I	65%P, 35%S	75%P, 25% S

Constraints Generation Table 2 displays experimental results for evaluating the model's ability to generate semantically/syntactically correct logical constraints within the graph structure and Python syntax of DomiKnowS. As indicated by the high percentage of resolved errors (*%S* in *%E in Resolved E*) detected by our custom-designed parser, it plays a significant role in improving program accuracy. This is accomplished through the feedback loop, by providing clear descriptions of both syntactic and semantic errors. It is notable that a small sampling factor (3 samples) enhances model accuracy, and FOL mapping serves as a valuable intermediary layer, reducing the need for iterative constraint adjustments. Manual evaluation reveals that while most generated constraints (even in erroneous tasks) are semantically accurate, issues arise in adapting to the DomiKnowS language due to limited prior exposure of the LLM to its specific syntax. While dynamic retrieval aids in direct constraint generation, it hinders performance using FOL due to the lack of exposure to FOL-to-DomiKnowS mappings, which encourages the use of unsupported FOL operations in DomiKnowS.

5.2   Human Evaluation

The human evaluation assesses two key aspects of our demo. Firstly, it measures the comparative ease and utility of our proposed demo in contrast to the conventional method of navigating the documentation and manually crafting programs within the Python package of the DomiKnowS library. Secondly, the evaluation provides insights into the accuracy of each step in the interactive process, detailing the quantity and nature of user interventions required for satisfactory performance across diverse tasks. Additionally, we aim to assess whether the proposed tool facilitates users unfamiliar with the DomiKnowS programming language in validating the model's responses.

Interface V.S. Coding We assess user experiences in crafting structured prediction tasks in DomiKnowS with two volunteer groups (5 people per group). Each group received a 20-minute introduction to structured prediction and a high-level overview of DomiKnowS. The first group used the demo interface, while the second group, with access to DomiKnowS documents and examples, manually composed concept graphs and logical constraints. Tasks included hierarchical image classification, sentiment analysis, and entity mention and relation extraction.

Groups' Sub-tasks: Group 1: Describe the task and interact with the interface. Group 2: Read the documentation and examples; Write the code.

Average Time of Implementation: Group 1: 20 minutes, Group 2: 1 hour

Time Splits: Group 1: 60% user input - 40% model output. Group 2: 35% documents - 65% coding

Findings: Human effort using the interface is reduced 5 times. Opting for the local LLM version instead of the API can further accelerate final code generation. Notably, the second group often sought additional guidance on task modeling, while the tools integrated into the demo interface effectively guided the first group. Additionally, we instructed the second group to revisit the task using the demo interface, allocating them a 10-15 minute timeframe. The consensus within this group was that the demo interface provided a significantly superior and more user-friendly experience.

Interactive Setting To evaluate the demo in an end-to-end interactive setting, we asked two volunteers to implement a total of **seven** tasks while recording their every interaction with the interface.

Duration: The average time for the process was 17 minutes. The most time-consuming part was the constraint generation, taking more time for both the user to write constraints and the model to generate them (averaging 150 seconds for model responses).

Task Descriptions: In 6 out of 7 tasks, the user removed only additional and unnecessary information. In one task, the user extended the decision set by replacing the word 'etc.' with actual labels.

Concept List: In 4 out of 7 tasks, one sample was correct, while in the remaining tasks, the user had to remove or add less than two concepts to the list.

Concept Graph: In 4 out of 7 tasks, the correct graph was generated without user intervention. In 2 out of 7 tasks, one interaction was needed. In 1 out of 7 tasks, the user interacted five times to remove a wrongly included relationship between concepts where multiple relationships existed. The visualization tool notably provided users with all the necessary information to evaluate the graph.

Constraints: Most of the constraints were both semantically and syntactically correct. In cases with erroneous results, the verification process could capture the error but not resolve it correctly. Errors were mainly due to using syntax close to FOL for operations not directly supported in DomiKnowS, like equality between multiple variables. Remarkably, in cases such as constraints in a sudoku table,

the model detected similar patterns and used for-loops to implement multiple constraints with similar logical structures but variant logical predicates.

6   Conclusion

We proposed a natural language interface for declarative programming to facilitate the design of neuro-symbolic models by domain experts. We exploit LLMs and prompt engineering techniques to convert natural language into declarative code, written in a specific syntax for the *DomiKnowS* neuro-symbolic modeling framework. It includes the structure of concepts and relationships related to the problem, as well as formal logical constraints expressing domain knowledge. Building on top of these declarations, *DomiKnowS* integrates domain knowledge with deep models of any kind designed in PyTorch. This framework is a step forward in making the design of deep models accessible to domain experts, allowing them to inject their domain knowledge into the model using natural language.

Acknowledgments

Bibliography

[1] K. Ahmed, T. Li, T. Ton, Q. Guo, K.-W. Chang, P. Kordjamshidi, V. Srikumar, G. Van den Broeck, and S. Singh. Pylon: A pytorch framework for learning with constraints. In *NeurIPS 2021 Competitions and Demonstrations Track*, pages 319–324. PMLR, 2022.

[2] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[3] S. Bowman, G. Angeli, C. Potts, and C. D. Manning. A large annotated corpus for learning natural language inference. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 632–642, 2015.

[4] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.

[5] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[6] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[7] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[8] H. R. Faghihi and P. Kordjamshidi. Time-stamped language model: Teaching language models to understand the flow of events. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 4560–4570, 2021.

[9] H. R. Faghihi, Q. Guo, A. Uszok, A. Nafar, and P. Kordjamshidi. DomiKnowS: A library for integration of symbolic domain knowledge in deep learning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 231–241, 2021.

[10] H. R. Faghihi, P. Kordjamshidi, C. M. Teng, and J. Allen. The role of semantic parsing in understanding procedural text. In *Findings of the Association for Computational Linguistics: EACL 2023*, pages 1792–1804, 2023.

[11] M. Freitag and Y. Al-Onaizan. Beam search strategies for neural machine translation. *ACL 2017*, page 56, 2017.

[12] Q. Guo, H. R. Faghihi, Y. Zhang, A. Uszok, and P. Kordjamshidi. Inference-masked loss for deep structured output learning. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 2754–2761, 2021.

[13] S. Han, H. Schoelkopf, Y. Zhao, Z. Qi, M. Riddell, L. Benson, L. Sun, E. Zubova, Y. Qiao, M. Burtell, et al. Folio: Natural language reasoning with first-order logic. *arXiv preprint arXiv:2209.00840*, 2022.

[14] N. Hoernle, R. M. Karampatsis, V. Belle, and K. Gal. Multiplexnet: Towards fully satisfied logical constraints in neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 36, pages 5700–5709, 2022.

[15] Z. Hu, X. Ma, Z. Liu, E. Hovy, and E. Xing. Harnessing deep neural networks with logic rules. In *54th ACL*, pages 2410–2420, 2016.

[16] J. Huang, Z. Li, B. Chen, K. Samel, M. Naik, L. Song, and X. Si. Scallop: From probabilistic deductive databases to scalable differentiable reasoning. *Advances in Neural Information Processing Systems*, 34:25134–25145, 2021.

[17] J. Huang, S. S. Gu, L. Hou, Y. Wu, X. Wang, H. Yu, and J. Han. Large language models can self-improve. *arXiv preprint arXiv:2210.11610*, 2022.

[18] Z. Jin, Y. Chen, F. Leeb, L. Gresele, O. Kamal, L. Zhiheng, K. Blin, F. G. Adauto, M. Kleiman-Weiner, M. Sachan, et al. Cladder: Assessing causal reasoning in language models. In *Thirty-seventh conference on neural information processing systems*.

[19] J. Kocoń, I. Cichecki, O. Kaszyca, M. Kochanek, D. Szydło, J. Baran, J. Bielaniewicz, M. Gruza, A. Janz, K. Kanclerz, et al. Chatgpt: Jack of all trades, master of none. *Information Fusion*, page 101861, 2023.

[20] P. Kordjamshidi, D. Roth, and H. Wu. Saul: towards declarative learning based programming. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, page 1844–1851. AAAI Press, 2015. ISBN 9781577357384.

[21] P. Kordjamshidi, D. Roth, and K. Kersting. Systems ai: A declarative learning based programming perspective. In *International Joint Conference on Artificial Intelligence*, 2018. URL `https://api.semanticscholar.org/CorpusID:44080347`.

[22] P. Kordjamshidi, D. Roth, and K. Kersting. Declarative learning-based programming as an interface to ai systems. *Frontiers in artificial intelligence*, 5:755361, 2022.

[23] A. Krizhevsky, V. Nair, and G. Hinton. Cifar-100 (canadian institute for advanced research). URL `http://www.cs.toronto.edu/~kriz/cifar.html`.

[24] J. Lloyd. Practical advantages of declarative programming. In *Unknown*, pages 3 – 17, 1994. Conference Proceedings/Title of Journal: Joint Conference on Declarative Programming.

[25] R. Manhaeve, S. Dumancic, A. Kimmig, T. Demeester, and L. De Raedt. Deepproblog: Neural probabilistic logic programming. *Advances in neural information processing systems*, 31, 2018.

[26] R. Mirzaee and P. Kordjamshidi. Disentangling extraction and reasoning in multihop spatial reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 3379–3397, 2023.

[27] A. Nafar, K. B. Venable, and P. Kordjamshidi. Probabilistic reasoning in generative large language models. *arXiv preprint arXiv:2402.09614*, 2024.

[28] Y. Nandwani, A. Pathak, P. Singla, et al. A primal dual formulation for deep learning with constraints. In *Advances in Neural Information Processing Systems*, pages 12157–12168, 2019.

[29] A. Ni, S. Iyer, D. Radev, V. Stoyanov, W.-t. Yih, S. Wang, and X. V. Lin. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pages 26106–26128. PMLR, 2023.

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

[31] L. Pan, A. Albalak, X. Wang, and W. Y. Wang. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. *arXiv preprint arXiv:2305.12295*, 2023.

[32] G. Poesia, K. Gandhi, E. Zelikman, and N. D. Goodman. Certified reasoning with language models. *arXiv preprint arXiv:2306.04031*, 2023.

[33] H. Rajaby Faghihi and P. Kordjamshidi. Consistent joint decision-making with heterogeneous learning models. In Y. Graham and M. Purver, editors, *Findings of the Association for Computational Linguistics: EACL 2024*, pages 803–813, St. Julian's, Malta, Mar. 2024. Association for Computational Linguistics. URL `https://aclanthology.org/2024.findings-eacl.53`.

[34] H. Rajaby Faghihi, A. Nafar, C. Zheng, R. Mirzaee, Y. Zhang, A. Uszok, A. Wan, T. Premsri, D. Roth, and P. Kordjamshidi. Gluecons: A generic benchmark for learning under constraints. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(8):9552–9561, Jun. 2023. `https://doi.org/10.1609/aaai.v37i8.26143`.

[35] D. Roth. Learning Based Programming. Technical Report UIUCDCS-R-99-2127, 10 1999. URL `http://cogcomp.org/papers/Roth99c.pdf`.

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

[37] T. Scholak, N. Schucher, and D. Bahdanau. Picard: Parsing incrementally for constrained auto-regressive decoding from language models. In *EMNLP*, pages 9895–9901, 2021.

[38] Y. Shen, K. Song, X. Tan, D. Li, W. Lu, and Y. Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface. *arXiv preprint arXiv:2303.17580*, 2023.

[39] N. Tandon, B. Dalvi, K. Sakaguchi, P. Clark, and A. Bosselut. WIQA: A dataset for "what if..." reasoning over procedural text. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 6076–6085, Hong Kong, China, Nov. 2019. Association for Computational Linguistics. `https://doi.org/10.18653/v1/D19-1629`. URL `https://aclanthology.org/D19-1629`.

[40] E. F. Tjong Kim Sang and F. De Meulder. Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. In *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003*, pages 142–147, 2003. URL `https://aclanthology.org/W03-0419`.

[41] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[42] V. Viswanathan, C. Zhao, A. Bertsch, T. Wu, and G. Neubig. Prompt2model: Generating deployable models from natural language instructions. *arXiv preprint arXiv:2308.12261*, 2023.

[43] Z. Wu, L. Zhang, C. Cao, X. Yu, H. Dai, C. Ma, Z. Liu, L. Zhao, G. Li, W. Liu, et al. Exploring the trade-offs: Unified large language models vs local fine-tuned models for highly-specific radiology nli task. *arXiv preprint arXiv:2304.09138*, 2023.

[44] J. Xu, Z. Zhang, T. Friedman, Y. Liang, and G. Broeck. A semantic loss function for deep learning with symbolic knowledge. In *ICML*, pages 5502–5511. PMLR, 2018.

[45] Z. Yang, A. Ishay, and J. Lee. Coupling large language models with logic programming for robust and general reasoning from text. *arXiv preprint arXiv:2307.07696*, 2023.

[46] P. Yu and S. Bach. Alfred: A system for prompted weak supervision. *arXiv preprint arXiv:2305.18623*, 2023.

[47] T. Zhang, T. Yu, T. Hashimoto, M. Lewis, W.-t. Yih, D. Fried, and S. Wang. Coder reviewer reranking for code generation. In *International Conference on Machine Learning*, pages 41832–41846. PMLR, 2023.

[48] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.