Counterexample Guided Inductive Synthesis Using Large Language Models and Satisfiability Solving

Sumit Kumar Jha
Computer Science Department
Florida International University
sumit.jha@fiu.edu

Susmit Jha and Patrick Lincoln
Computer Science Laboratory
SRI International
susmit.jha@sri.com

Nathaniel D. Bastian
Army Cyber Institute
United States Military Academy
nathaniel.bastian@westpoint.edu

Alvaro Velasquez
Department of Computer Science
University of Colorado Boulder
alvaro.velasquez@colorado.edu

Rickard Ewetz
Electrical and Computer Engineering
University of Central Florida
rickard.ewetz@ucf.edu

Sandeep Neema
Electrical and Computer Engineering
Vanderbilt University
sandeep.neema@vanderbilt.edu

***Abstract*—Generative large language models (LLMs) can follow human-provided instruction prompts and generate human-like responses. Apart from natural language responses, they have been found to be effective at generating formal artifacts such as code, plans, and logical specifications. Despite their remarkably improved accuracy, these models are still known to produce factually incorrect or contextually inappropriate results despite their syntactic coherence – a phenomenon often referred to as *hallucinations*. This limitation makes it difficult to use these models to synthesize formal artifacts used in safety-critical applications. Unlike tasks such as text summarization and question-answering, bugs in code, plan, and other formal artifacts produced by LLMs can be catastrophic. We posit that we can use the satisfiability modulo theory (SMT) solvers as deductive reasoning engines to analyze the generated solutions from the LLMs, produce counterexamples when the solutions are incorrect, and provide that feedback to the LLMs exploiting the dialog capability of LLMs. This interaction between inductive LLMs and deductive SMT solvers can iteratively steer the LLM to generate the correct response. In our experiments, we use planning over the domain of blocks as our synthesis task for evaluating our approach. We use GPT-4, GPT3.5 Turbo, Davinci, Curie, Babbage, and Ada as the LLMs and Z3 as the SMT solver. Our method allows the user to communicate the planning problem in natural language; even the formulation of queries to SMT solvers is automatically generated from natural language. Thus, the proposed technique can enable non-expert users to describe their problems in natural language, and the combination of LLMs and SMT solvers can produce provably correct solutions.**

The authors acknowledge support from the National Science Foundation awards 2319401, 2113307, and 1908471, the DARPA cooperative agreement HR00112020002, DARPA FA8750-23-2-0501, ONR grant N000142112332, DOE grant DE-SC0023494, DE-SC0024428, DE-SC0024576, and the U.S. Army Research Laboratory Cooperative Research Agreement W911NF-17-2-0196 and under Support Agreement No. USMA21050. The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

I. Introduction

Many safety-critical applications such as communication protocols, adaptive software-defined radio, autonomous vehicles, and assistive robots need to operate in the presence of uncertainty and have the capability to adapt to an evolving non-stationary environment. This need for generalization and adaptation has fueled the adoption of artificial intelligence (AI) for such applications. We can categorize these AI components into two categories - inductive learning-based methods that generalize well but do not provide any guarantees on their output, and deductive automated reasoning methods that require significant user expertise but provide guarantees on their outputs. Examples of inductive learning methods include decision trees, support vector machines, deep learning models, the more recent class of large language models (LLMs), and, more generally, foundation models that generalize across tasks and domains. Examples of deductive symbolic reasoning methods include propositional satisfiability (SAT) solvers, satisfiability modulo theory (SMT) solvers, program synthesizers, and proof assistants. While inductive learning techniques have been typically used for tasks such as perception requiring decisions over very high-dimensional inputs, the deductive AI methods have been successfully used to solve complex tasks such as automated planning, program reverse engineering, code repair, and protocol design. The remarkable success of LLMs has made it feasible to use them even for these complex tasks, and several recent results have demonstrated that LLMs can be used for program synthesis and planning [1], [2]. More generally, the LLMs exhibit limited symbolic reasoning capability making them applicable to problems that traditionally required symbolic AI reasoning methods such as SAT and SMT solving.

Despite the reasoning capability of LLMs being extremely limited compared to symbolic reasoning methods, they overcome one key hindrance that has limited the widespread use of formal methods and automated reasoning: the need for deep expertise in formalizing and specifying problems to these tools. LLMs only require natural language specification of the problem and have been shown to be capable of automated formalization. Further, the output of the LLMs can be accompanied with easy to understand natural language explanations. This ease of interaction with LLMs as reasoning engines can enable non-expert users to pose planning or program synthesis prompts to them and obtain a solution that is easy to understand. However, LLMs are prone to generating factually incorrect or contextually inappropriate responses, also referred to as hallucinations [3], [4]. The lossy encoding of knowledge in LLMs and the memory distortion that accompanies knowledge generalization naturally lead to inaccurate retrieval of knowledge even from training data. Thus, simply scaling data and models is not sufficient to resolve this problem.

II. Related Work

The use of large language models for tasks such as planning and code generation has been an area of active research in recent years, and numerous techniques have been developed to exploit the capabilities of these models in these domains. Large language models have been used to generate plans for complex tasks [2], [5] and for code generation [1], [6]. Several efforts to improve the performance of large language models in code generation and planning have been recently made through the use of specialized training techniques such as a combination of contrastive learning and fine-tuning [7], incorporation of external feedback in the form of success detection and human interaction through internal monologue [8], and decomposition of plans to sub-plans using LLMs followed by mapping to admissible actions using demonstrations [5]. Despite these advancements, several challenges remain in the application of large language models in planning and code generation. Notably, the model's ability to handle edge cases, produce consistent outputs, and understand the precise semantics of the generated code or plan is very limited [9], [10]. The lack of world knowledge and contextual understanding is another limitation of large language models in planning. These models struggle to consider real-world constraints and commonsense reasoning in their planning process, which often leads to the generation of unrealistic or unfeasible plans [11]. Further, the opacity of the decision-making process in large language models is a crucial limitation for the use of the LLM-produced programs or plans.

In order to make the LLM outputs more consistent and robust, and consequently less prone to errors and hallucinations, fine-tuning has been proposed to adapt to specific tasks or domains [12]. But for many domains, it is not obvious what a sufficient size of the domain-specific dataset needs to be used for fine-tuning, and continuously monitoring and fine-tuning LLMs is not practical. Failure of fine-tuning to improve LLM performance has been reported in literature [13]. Further, fine-tuning [14] adversely impacts the model's fluency, conversational capability, and in-context learning ability, which is critical to its response to prompts.

Another approach is to connect LLMs to knowledge graphs [15], which represent knowledge as a graph of interconnected entities and relationships. Knowledge graphs can encode a wide range of structured and unstructured knowledge, including facts, concepts, and relationships. Methods have been developed to infuse structured knowledge into LLMs by training models on factual triples of knowledge graphs (KGs), and such models pre-trained on knowledge graphs have been shown to outperform baselines [16]. But this requires a well-curated and complete knowledge base, building which is a time-consuming and expensive endeavor.

Yet another possibility is to encode external relevant knowledge into a key-value memory that exploits the fast maximum inner product search for memory querying. These memory slots can then be integrated with language models [17] for relatively smaller models, such as T5. Such a memory augmentation has been shown to improve the performance of the model on knowledge-intensive tasks. More recently, recurrent memory transformer (RMT) has been shown to be computationally efficient for large prompts with a million or more tokens [18].

But for tasks such as programming or plan generation, the space of possible queries is very large and can have many syntactic variations. Some of these simple syntactic variations could be superfluous with respect to the actual task, and not be relevant to the planning problem. It has been shown that LLMs are sensitive to irrelevant variations [19]. Hence, improving accuracy via expensive fine-tuning or explicit curation of knowledge graphs in such a context would be unrealistic as it would require not just creating variations relevant to the domain but also considering changes irrelevant to the core search problem. We also aim to avoid making significant changes to the deep learning model's architecture, such as memory augmentation, which can cause a significant decline in the scalability and accuracy of these models.

Our approach to generating trustworthy formal artifacts using LLMs is based on formal synthesis [20]–[22]. Formal synthesis generates a program, a plan, or a protocol satisfying a high-level formal specification. In particular, we

use the paradigm of counterexample-guided inductive synthesis [22] – an instance of oracle-guided inductive synthesis [23], wherein the oracle is a verifier, and the feedback provided to the learning engine is in the form of counterexamples. These formal approaches to synthesis blend induction and deduction in the sense that even as they generalize from examples, deductive procedures are used in the process of generalization. Formal inductive synthesis and machine learning fields have the same high-level goal: to develop algorithmic techniques for synthesizing a concept from observations. However, there are also important differences in the problem formulations and techniques used in both fields. First, the concept classes in machine learning tend to use optimization-friendly representations such as deep neural networks, convex polytopes, or half-spaces. In the case of formal synthesis, the target of learning includes general programs, protocols, or automata. Secondly, the learning process in formal synthesis often incorporates combinatorial search methods such as satisfiability solving, and hence, are much less scalable compared to purely learning techniques. But these approaches provide formal guarantees on the produced artifacts, which is lacking in learning-based approaches. This paper combines counterexample-guided inductive synthesis with LLMs for the scalable yet trusted synthesis of plans and programs.

III. Technical Approach

We begin by defining some basic terms and notations motivated by the literature on formal synthesis [23]. Following standard terminology in the machine learning theory community [25], we define a concept c as a set of examples drawn from a domain of examples $\mathbf{E}$. In other words, $c \subseteq \mathbf{E}$. For example, when learning to produce plans, the $\mathbf{E}$ could be the sequence of all possible state evolutions of the world, and the plan could be the subset of evolutions that correspond to the correct action sequence. The set of all possible concepts is termed the *concept class*, denoted by $\mathcal{C}$. Thus, $\mathcal{C} \subseteq 2^{\mathbf{E}}$. Depending on the application domain, $\mathbf{E}$ can be finite or infinite. The concept class $\mathcal{C}$ can also be finite or infinite. Note that it is possible to place (syntactic) restrictions on concepts so that $\mathcal{C}$ is finite even when $\mathbf{E}$ is infinite. Unlike machine learning, one distinguishing characteristic of our problem formulation is the presence of an explicit formal specification. This is crucial for an unambiguous definition of correct output.

While the complete specification is infeasible, many domains, such as planning and program synthesis, admit properties that characterize the target plan or program requirements. Thus, a specification Φ can be thought to represent a set of "correct" concepts, i.e., $\Phi \subseteq \mathcal{C} \subseteq 2^{\mathbf{E}}$. Any example $x \in \mathbf{E}$ such that there is a concept $c \in \Phi$ where $x \in c$ is called a positive example or witness. Likewise, an example x that is not contained in any $c \in \Phi$ is a negative example or counterexample. Given a concept $c \in \mathcal{C}$ we say that c satisfies Φ iff $c \in \Phi$. If we have a complete specification, it means that Φ is a singleton set comprising only a single allowed concept. In general, Φ is likely to be a partial specification that allows for multiple correct concepts.

Since we use LLMs for learning, the subsets of examples can be provided as part of the prompts and used for in-context learning by the LLMs. Using only the witnesses is common in machine learning, but the presence of deductive reasoning engines and formal verification methods can enable a richer set of inputs, such as counterexamples obtained from verifying whether the artifact produced from the LLM satisfies the given specification. In general, we can model the availability of this richer set of inputs that can access the specification Φ as an *oracle interface*. An *oracle interface* $\mathcal{O}$ is a subset of $\mathcal{Q} \times \mathcal{R}$ where $\mathcal{Q}$ is a set of query types, $\mathcal{R}$ is a corresponding set of response types, and $\mathcal{O}$ defines which pairs of query and response types are semantically well-formed. A simple instance of an oracle interface is one with a single query type that returns positive examples from $\mathbf{E}$. Another instance is an oracle that has access to a verification engine and can return counterexamples. Implementations of the oracle interface can be nondeterministic algorithms that exhibit nondeterministic choices in the stream of queries and responses. Consider a *concept class* $\mathcal{C}$, a domain of examples $\mathbf{E}$, a specification Φ, and an oracle interface $\mathcal{O}$. Our goal is to find a *target concept* $c \in \mathcal{C}$ that satisfies Φ, given only $\mathcal{O}$ and $\mathcal{C}$. In other words, $\mathbf{E}$ and Φ can be accessed only through $\mathcal{O}$.

We restrict the oracle to be a verifier that can produce counterexamples, and thus, our approach becomes an instance of counterexample-guided inductive synthesis (CEGIS). It was originally proposed as an algorithmic method for program synthesis where the specification is given as a reference program and the concept class is defined using a partial program, also referred to as a "sketch" [22]. In CEGIS, the learner (synthesizer) interacts with a "verifier" that can take in a candidate plan or a program and a specification, and try to find a counterexample showing that the candidate program does not satisfy the specification. In CEGIS, the learner is typically implemented on top of a general-purpose decision procedure such as a satisfiability solver, satisfiability modulo theory solver, or model checker. The oracle (verifier) is also implemented similarly. In contrast, we use the LLMs as the learner and a formal verification engine as the oracle.

Thus, we adopt the CEGIS architecture and locate the LLM as a scalable learner within it. This enables us to scale synthesis and, at the same time, produce plans or programs that are formally verified against the given specifications.

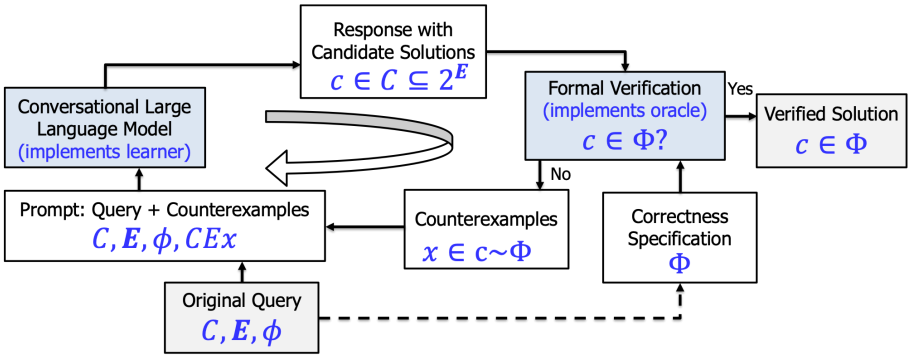

Fig. 1. The technical approach in this paper wraps the generative capability of the large language models in a formal synthesis loop inspired from counterexample-guided inductive synthesis [22] and more generally, oracle-guided inductive synthesis [23]. The learning oracle in our approach is the conversation large language model, and the verification engine serves as the oracle. In particular, we use SMT solvers [24] for verification. The initial query defines the concept class (plans over a given set of actions), the set of possible behaviors (state space and the evolution of the environment state), and the specification. The verifier complements this initial query by adding counterexamples — behaviors of the produced incorrect plans by the LLM that are not consistent with the specification ($\sim$ denotes symmetric set difference). The final output is the candidate produced by the LLM that is formally verified to be consistent with the given specification.

This proposed approach extends our recent work-in-progress report on improving the trust of LLMs [26], and exploits the following characteristics of the LLMs and the satisfiability solvers as verification engines. First, the LLMs can be prompted to generate well-structured outputs that can be easily parsed by a formal verification engine. The inclusion of a large corpus of code in their training data is perhaps responsible for this characteristic. We successfully use this to ensure the output of LLMs can be ingested by the verifier. We also use LLMs to derive the queries to the verifier, automatically checking for the correctness specification. Second, the formal verifiers are very fast at checking the correctness of a single solution for a task such as planning, as compared to solving the planning problem using combinatorial search and finding a valid plan. Formal verification techniques can also be used to localize the part of the solution that violates the specification and to provide informative feedback to the LLMs that rule out not just a single incorrect solution but a whole class of incorrect solutions. A large body of work on verification, fault isolation, and explanation generation can be leveraged using this architecture. This would expand the query types and response types $(\mathcal{Q}, \mathcal{R})$ in the formal synthesis framework. Further, LLMs demonstrate remarkable in-context learning, and adding counterexamples and explanations to the prompt steers them away from incorrect responses and eventually drives them to a correct solution that is accepted by the verifier. The context length used to train LLMs has been rapidly growing. Models such as GPT-4 can take a context length of 32K tokens, while other generative models with 100K tokens have also been developed. This larger context length enables LLMs to stay consistent with provided prompts. This increasing context length enables being able to support more number of counterexamples in the prompt. Thus, the current direction of improvement in LLMs supports more number of iterations of the presented architecture. Finally, the use of formal methods for tasks such as planning requires tedious modeling into existing solvers and verifiers. The responses of these formal engines also need to be translated back into more easily interpretable forms to be used by engineers and developers who are not experts in formal methods. The use of LLM in the above architecture also serves as a human-friendly front end and, thus, partially alleviates the challenge of making formal synthesis engines more accessible to non-expert users.

In the rest of the paper, we evaluate the proposed approach using automated planning as the synthesis domain and using SMT solver [24] as the verification engine. We first describe a case study in detail and then present the quantitative results.

IV. Experiments

We use block-world planning [27] for our experiments. We specify the high-level predicates that capture the state of the world, such as `BlockOn`, `OnTable`, and `Holding`, and their evolution. We describe the permitted operations/actions and the initial state. We provide a final state and ask the LLM to produce a valid plan. The LLMs take a few iterations with the verifier to produce the correct result. We consider responses where the verifier appends the incorrect sequences to the prompt only stating that these sequences are not valid or provides a richer response as a prefix of the invalid plan such that any completion would be incorrect. This iteration continues till the response of the LLM is verified to be correct.

The final response from the LLM provides a human-readable justification in addition to the final plan. We

use one example of a block-world problem to demonstrate our proposed approach before presenting quantitative results that compare different LLMs, such as GPT4, GPT3.5 Turbo, Davinci, Curie, Babbage, and Ada, and also consider planning problems of varying scale.

Illustrative Example We consider the block world planning BLOCKWORLD problem and translate it into formal first-order constraints using the LLM. We provide the details in [28] and focus here on the iterative refinement of the prompt to steer LLM toward a correct response using the formal verifier as the guiding oracle. We provide two examples in our prompt to assist the LLMs to inductively reason about a new problem and respond with its plan. One of the two examples is listed below.

```
Given the block world problem OLDPROB1:
(define (problem BW-sample-0)
(:domain blocksworld-4ops)
(:objects b1 b2 b3 b4 b5 b6 )

(:init
(arm-empty)
(on b1 b2)
(on b2 b3)
(on b3 b4)
(on b4 b5)
(on b5 b6)
(on-table b6)
(clear b1)
(clear b2)
(clear b3)
(clear b4)
(clear b5)
(clear b6)
)

(:goal
(and
(on b6 b5)
(on b5 b4)
(on b4 b3)
(on b3 b2)
(on b2 b1)
(on-table b1)
)))
```

We first list the problem, as shown above, and then a plan for the planning problem.

```
The solution for the problem OLDPROB1 is:
START-PLAN
1. unstack b1 b2
2. put-down b1
3. unstack b2 b3
4. put-down b2
5. unstack b3 b4
6. put-down b3
7. unstack b4 b5
8. put-down b4
9. unstack b5 b6
10. put-down b5
11. pick-up b6
12. stack b6 b5
13. pick-up b5
14. stack b5 b4
15. pick-up b4
16. stack b4 b3
17. pick-up b3
18. stack b3 b2
19. pick-up b2
20. stack b2 b1
END-PLAN
```

Now, given a new block world problem NEWPROB, the LLM may produce an incorrect response:

```
(define (problem BW-rand-6)
(:domain blocksworld-4ops)
(:objects b1 b2 b3 b4 b5 b6 )
(:init
(arm-empty)
(on b1 b6)
(on b2 b3)
(on-table b3)
(on b4 b1)
(on-table b5)
(on-table b6)
(clear b2)
(clear b4)
(clear b5)
)
(:goal
(and
(on b1 b2)
(on b5 b3)
(on b6 b4))
)
)
```

The incorrect solution from the problem NEWPROB is:

```
START-PLAN
1. unstack b4 b1
2. put-down b4
3. unstack b1 b6
4. put-down b1
5. pick-up b2
6. stack b2 b3
7. pick-up b1
8. stack b1 b2
9. pick-up b5
10. stack b5 b3
11. pick-up b6
12. stack b6 b4
END-PLAN
```

The SMT solver identifies the following as an infeasible subset of the plan and generates a new prompt including the statement that "Any plan with the following prefix is not correct":

```
START-PLAN
1. unstack b4 b1
2. put-down b4
3. unstack b1 b6
4. put-down b1
5. pick-up b2
```

Using this information, the LLM produces a new response that is verified to be correct by the decision procedure.

```
START-PLAN
1. unstack b4 b1
2. put-down b4
3. unstack b1 b6
4. put-down b1
5. unstack b2 b3
6. put-down b2
7. pick-up b5
8. stack b5 b3
9. pick-up b6
10. stack b6 b4
11. pick-up b1
12. stack b1 b2
END-PLAN
```

Quantitative Results We analyze the performance of different GPT on 20 random problems with 3 blocks in Fig. 2 (left). GPT-4 is the most successful model and we analyze its performance with with 3, 4, 5 and 6 blocks. In Fig. 2 (right), we observe that it is only able to solve 2 out of the 20 problems with 10 blocks even though it could solve 19 out of the 20 problem with 3 blocks. Additional results are available in our accompanying technical report [28].

We study how LLMs are improved by adding verification in the loop and report the number of iterations taken by our approach to produce the correct result. In Fig. 3, we analyze the number of iterations of counterexamples need by GPT-4 and GPT-3.5 on 20 random problems involving 3 and 4 blocks.

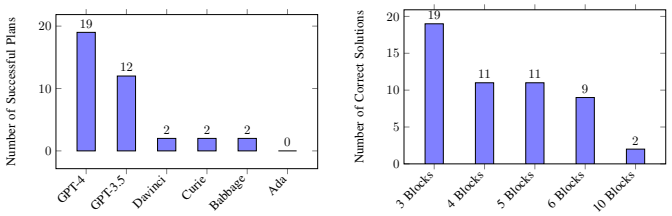

Fig. 2. (left) The performance of different GPT models on problems with 3 blocks. (right) The performance of GPT-4 on problems with multiple blocks.

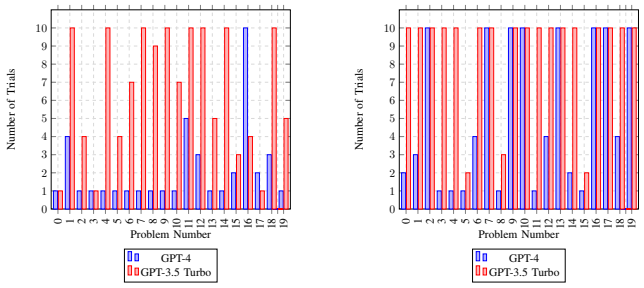

Fig. 3. Number of iterations for GPT-4 and GPT-3.5 Turbo for 3 and 4 blocks.

V. Conclusions

We present a novel approach to addressing the challenge of the trustworthy generation of formal artifacts such as plans and programs from LLMs. Our technique uses the paradigm of counterexample-guided inductive synthesis wherein the learner is implemented using the LLM and the verifier uses SMT solver. Our approach can be viewed as an adversarial variant of the in-context learning approach, wherein we use formal verification to detect incorrect responses and include the generated counterexamples as a part of the prompt in the dialog with the LLM. The initial experiments reported over the planning task in this paper are encouraging and indicate that the proposed combination of LLMs and formal verifiers can be an effective approach to using LLMs in applications where the generated artifact must be verified.

References

[1] 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:2107.03374*, 2021.

[2] C. H. Song, J. Wu, C. Washington, B. M. Sadler, W.-L. Chao, and Y. Su, "Llm-planner: Few-shot grounded planning for embodied agents with large language models," *arXiv preprint arXiv:2212.04088*, 2022.

[3] M. Cao, Y. Dong, and J. C. K. Cheung, "Hallucinated but factual! inspecting the factuality of hallucinations in abstractive summarization," in *Proceedings of the ACL (Volume 1: Long Papers)*, 2022.

[4] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," *ACM Computing Surveys*, 2022.

[5] W. Huang, P. Abbeel, D. Pathak, and I. Mordatch, "Language models as zero-shot planners: Extracting actionable knowledge for embodied agents," in *International Conference on Machine Learning*. PMLR, 2022, pp. 9118–9147.

[6] 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.

[7] S. Liu, B. Wu, X. Xie, G. Meng, and Y. Liu, "Contrabert: Enhancing code pre-trained models via contrastive learning," *arXiv preprint arXiv:2301.09072*, 2023.

[8] W. Huang, F. Xia, T. Xiao, H. Chan, J. Liang, P. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar *et al.*, "Inner monologue: Embodied reasoning through planning with language models," *arXiv preprint arXiv:2207.05608*, 2022.

[9] K. Valmeekam, S. Sreedharan, M. Marquez, A. Olmo, and S. Kambhampati, "On the planning abilities of large language models (a critical investigation with a proposed benchmark)," *arXiv:2302.06706*, 2023.

[10] K. Valmeekam, A. Olmo, S. Sreedharan, and S. Kambhampati, "Large language models still can't plan (a benchmark for llms on planning and reasoning about change)," *arXiv:2206.10498*, 2022.

[11] G. Marcus, "The next decade in ai: four steps towards robust artificial intelligence," *arXiv preprint arXiv:2002.06177*, 2020.

[12] C. Lee, K. Cho, and W. Kang, "Mixout: Effective regularization to finetune large-scale pretrained language models," *arXiv preprint arXiv:1909.11299*, 2019.

[13] M. Bommarito II and D. M. Katz, "GPT takes the bar exam," *arXiv preprint arXiv:2212.14402*, 2022.

[14] Y. Wang, S. Si, D. Li, M. Lukasik, F. Yu, C.-J. Hsieh, I. S. Dhillon, and S. Kumar, "Preserving in-context learning ability in large language model fine-tuning," *arXiv preprint arXiv:2211.00635*, 2022.

[15] Q. Wang, Z. Mao, B. Wang, and L. Guo, "Knowledge graph embedding: A survey of approaches and applications," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 12, pp. 2724–2743, 2017.

[16] F. Moiseev, Z. Dong, E. Alfonseca, and M. Jaggi, "Skill: Structured knowledge infusion for large language models," *arXiv preprint arXiv:2205.08184*, 2022.

[17] Y. Wu, Y. Zhao, B. Hu, P. Minervini, P. Stenetorp, and S. Riedel, "An efficient memory-augmented transformer for knowledge-intensive nlp tasks," *arXiv preprint arXiv:2210.16773*, 2022.

[18] A. Bulatov, Y. Kuratov, and M. S. Burtsev, "Scaling transformer to 1m tokens and beyond with rmt," *arXiv preprint arXiv:2304.11062*, 2023.

[19] S. K. Jha, R. Ewetz, A. Velasquez, and S. Jha, "Responsible reasoning with large language models and the impact of proper nouns," in *Workshop on Trustworthy and Socially Responsible ML, NeurIPS*, 2022.

[20] S. K. Jha, *Towards automated system synthesis using sciduction*. University of California, Berkeley, 2011.

[21] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2010.

[22] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *ASPLOS*, 2006.

[23] S. Jha and S. A. Seshia, "A theory of formal synthesis via inductive learning," *Acta Informatica*, vol. 54, pp. 693–726, 2017.

[24] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*. Springer, 2008, pp. 337–340.

[25] D. Angluin, "Queries and concept learning," *Machine learning*, vol. 2, pp. 319–342, 1988.

[26] S. Jha, S. K. Jha, P. Lincoln, N. D. Bastian, A. Velasquez, and S. Neema, "Dehallucinating large language models using formal methods guided iterative prompting," in *IEEE International Conference on Assured Autonomy*, 2023.

[27] N. Gupta and D. S. Nau, "On the complexity of blocks-world planning," *Artificial Intelligence*, vol. 56, no. 2-3, pp. 223–254, 1992.

[28] S. K. Jha, S. Jha, P. Lincoln, N. D. Bastian, A. Velasquez, R. Ewetz, and S. Neema, "Neuro symbolic reasoning for planning: Counterexample guided inductive synthesis using large language models and satisfiability solving," 2023. [Online]. Available: https://arxiv.org/abs/2309.16436