Survey on Constrained LLM Decoding

Breandan Considine
McGill University
bre@ndan.co

Xujie Si
University of Toronto
six@cs.utoronto.ca

Abstract

Recent advances in language model (LLM) decoding have led to the publication of a lot of papers that claims to always be syntactically valid. However, the reliability and efficiency of these methods vary widely in practice. We survey the landscape of language model decoding methods, focusing on those which claim to generate only valid programs. We compare the soundness, completeness, naturalness, parallelism and tooling of each approach.

1 Introduction

Many methods to sample from LLMs have been proposed. Some of these guarantee that all samples are grammatically valid. Others guarantee that all grammatically valid samples are generable. Still others guarantee that all samples are unique, i.e., converge linearly to the language.

The trick is not just synthesizing valid functions, but doing so in a parallel communication-free manner, without compromising soundness or completeness. You want to massively scale up a discrete sampler without replacement. One very efficient way of doing this is by constructing an explicit bijection from the sample space to the integers, sampling integers, then decoding them into programs.While this technique enjoys certain advantages, i.e., it is embarrassingly parallelizable and guaranteed to enumerate distinct solutions with a bounded delay, it also somewhat unnatural. By flattening the distribution onto the integers a la Gödel numbering, it destroys locality, does not play well with incremental decoding methods (left-to-right is currently en vogue in generative language modeling), and will fail if the sample space is uncountable.

Maciej Bendkowski has some related work [1] on generating random lambda terms, but was unable to overcome what he calls the asymptotic sparsity problem:

"Sampling simply-typed terms seems notoriously more challenging than sampling closed ones. Even rejection sampling, whenever applicable, admits serious limitations due to the imminent asymptotic sparsity problem — asymptotically almost no term, be it either plain or closed, is at the same time (simply) typeable."

Victor Maia has done some interesting work on this topic, but it is not clear how scalable it is. Another key step is reducing symmetries in your sample space by quotienting it somehow (e.g., by α-equivalence). The author seems to be invoking some kind of equivalence relation by "superposition", but the technical details here are a little fuzzy.

This problem is also closely related to model counting in the CSP literature, so a practical speedup could lead to improvements on a lot of interesting downstream benchmarks.

In general, the problem of program induction from input-output examples is not well-posed, so specialized solvers that can make stronger assumptions will usually have an advantage on domain-specific benchmarks. Most existing program synthesizers do not satisfy all of these desiderata, e.g., soundness (**S**), completeness (**S**), naturalness (**N**), and parallelism (||). It depends on how you define ||, but basically, we want to decode in parallel. So an LLM that uses a GPU we do not consider to be "parallel" in the sense we mean here.

	S	**C**	**N**	**Theory**	**||**	**Tool**
Tidyparse [4]	✓	✓	✓	$\text{CFG}_\cap$	✓	IDE-ready
Seq2Parse [8]	✓†	✗	✓	CFG	✗	Python
BIFI [11]	✗	✗	✓	Σ^*	✗	Python
OrdinalFix [12]	✓	✗	✗	CFG+	✗	Rust
Aho/Peterson [1]	✓	✗	✗	CFG	✗	None

Now, we consider just LLM-based SyGuS synthesizers.

	S	**C**	**N**	**Theory**	**||**	**Tool**
Outlines [10]	✓†	✓†	✓	CFG	✗	Python
SynCode [9]	✓	?	✓	CFG	✗	Python
GAD [6]	✓	?	✓	CFG	✗	Python
CodeGuard+ [5]	✓	?	✓	CFG	✗	Python
FLAP [7]	✓	?	✓	CFG	✗	Python
DOMINO [3]	✓	✗	✓	CFG	✗	Python

Also, we consider discrete program search and enumerative search techniques that do not use an LLM, but allow some semantic constraints on the generated program.

	S	**C**	**N**	**Theory**	**||**	**Tool**
Boltzmann Brain [2]	✓	✓	✓	UTλC	✓	Python
OrdinalFix [12]	✓	?	✗	N/A	✓	Rust
Bend/DPS	✓	?	?	IntCalc	✓	CUDA

Another important feature is incrementality, which none of the existing synthesizers really implement properly. Basically, you want the ability to compute quotients, i.e.:

$$L_1/L_2 = \{w \in \Sigma^* \mid \exists x \in L_2 \colon\ wx \in L_1\} \tag{1}$$

$$L_2 \backslash L_1 = \{w \in \Sigma^* \mid \exists x \in L_2 \colon\ xw \in L_1\} \tag{2}$$

References

[1] Alfred V Aho and Thomas G Peterson. 1972. A minimum distance error-correcting parser for context-free languages. *SIAM J. Comput.* 1, 4 (1972), 305–312.

[2] Maciej Bendkowski. 2022. Automatic compile-time synthesis of entropy-optimal Boltzmann samplers. *arXiv preprint arXiv:2206.06668* (2022).

[3] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2024. Guiding LLMs The Right Way: Fast, Non-Invasive Constrained Generation. *arXiv preprint arXiv:2403.06988* (2024).

[4] Breandan Considine. 2023. A Pragmatic Approach to Syntax Repair. In *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. 19–21.

[5] Yanjun Fu, Ethan Baker, and Yizheng Chen. 2024. Constrained Decoding for Secure Code Generation. *arXiv preprint arXiv:2405.00218* (2024).

[6] Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova, and Loris D'Antoni. 2024. Grammar-Aligned Decoding. *arXiv preprint arXiv:2405.21047* (2024).

[7] Shamik Roy, Sailik Sengupta, Daniele Bonadiman, Saab Mansour, and Arshit Gupta. 2024. Flap: Flow adhering planning with constrained decoding in llms. *arXiv preprint arXiv:2403.05766* (2024).

[8] Georgios Sakkas, Madeline Endres, Philip J Guo, Westley Weimer, and Ranjit Jhala. 2022. Seq2Parse: neurosymbolic parse error repair. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 1180–1206.

[9] Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2024. Improving llm code generation with grammar augmentation. *arXiv preprint arXiv:2403.01632* (2024).

[10] Brandon T Willard and Rémi Louf. 2023. Efficient guided generation for llms. *arXiv preprint arXiv:2307.09702* (2023).

[11] Michihiro Yasunaga and Percy Liang. 2021. Break-it-fix-it: Unsupervised learning for program repair. In *International conference on machine learning*. PMLR, 11941–11952.

[12] Wenjie Zhang, Guancheng Wang, Junjie Chen, Yingfei Xiong, Yong Liu, and Lu Zhang. 2023. OrdinalFix: Fixing Compilation Errors via Shortest-Path CFL Reachability. *arXiv preprint arXiv:2309.06771* (2023).