Improving Cross-Domain Low-Resource Text Generation through LLM Post-Editing: A Programmer-Interpreter Approach

Zhuang Li, Levon Haroutunian,
Raj Tumuluri, Philip Cohen, Gholamreza Haffari
Openstream.ai
{zhuang.li, levon, raj, phil.cohen, reza.haffari}@openstream.com

Abstract

Post-editing has proven effective in improving the quality of text generated by large language models (LLMs) such as GPT-3.5 or GPT-4, particularly when direct updating of their parameters to enhance text quality is infeasible or expensive. However, relying solely on smaller language models for post-editing can limit the LLMs' ability to generalize across domains. Moreover, the editing strategies in these methods are not optimally designed for text-generation tasks. To address these limitations, we propose a neural programmer-interpreter approach that preserves the domain generalization ability of LLMs when editing their output. The editing actions in this framework are specifically devised for text generation. Extensive experiments demonstrate that the programmer-interpreter significantly enhances GPT-3.5's performance in logical form-to-text conversion and low-resource machine translation, surpassing other state-of-the-art (SOTA) LLM post-editing methods in cross-domain settings.

1 Introduction

Large pre-trained language models like GPT-3.5[1] or GPT-4[2] have gained significant attention in natural language research. However, fine-tuning these models for specific tasks is challenging due to limited computational resources or inaccessible parameters. Consequently, many researchers resort to using web APIs for instructing LLMs, leveraging zero-shot or few-shot in-context learning, enabling the LLMs to tackle tasks they weren't explicitly trained for. Unfortunately, this approach falls short when tackling some low-resource sequence generation tasks in machine translation (MT), and logical form (LF)-to-text translation, as shown in Lai et al. (2023); Haroutunian et al. (2023). In such cases, minimal task-specific data was available during the LLMs' pre-training phase. The output quality of LLMs for such tasks is compromised due to the absence of task-specific knowledge.

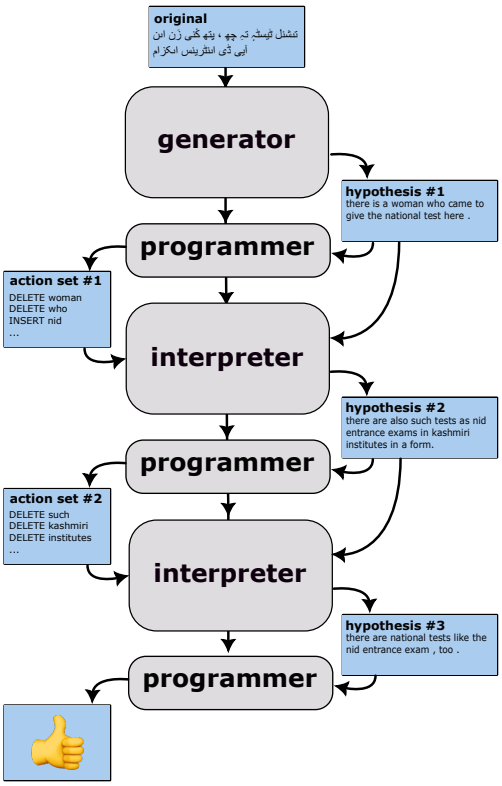

Figure 1: The diagram of our post-editing architecture.

To address this challenge, a promising set of solutions suggests integrating task-specific knowledge into language models through post-editing the generated text using a smaller model fine-tuned on task-specific data. Yet, these methods are not without their drawbacks. Our findings indicate that exclusive reliance on a smaller model for editing, e.g. Self-Correct (Welleck et al., 2022), results in suboptimal performance in domain generalization scenarios, likely due to the inherently limited domain knowledge within these smaller models.

As LLMs (i.e. GPT-3.5 or GPT-4) have shown superior domain generalization ability (Wang et al.,

[1]https://platform.openai.com/docs/models/gpt-3-5-turbo
[2]https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo

2023; Yang et al., 2023) over the fine-tuned model, we introduce an innovative approach based on the programmer-interpreter framework (Reed and de Freitas, 2016), which benefits from the domain generalization ability from LLMs. The programmer component - a smaller language model fine-tuned on task-specific data - delivers precise edit instructions to the larger language model, thus infusing the large model with task-specific knowledge. The interpreter, in turn, edits the large model's output given the provided instructions. Contrary to the Self-Correct (Welleck et al., 2022) approach that utilizes smaller, fine-tuned models for editing, our interpreter is also an LLM. The editing is accomplished through the use of prompts that include editing instructions, eliminating the need for any additional fine-tuning. This distinct framework guarantees the preservation of the LLM's domain generalization ability while simultaneously benefiting from the task-specific knowledge encoded by the programmer. Our method distinguishes itself from approaches like PiVe (Han et al., 2023), which also employ an LLM as the interpreter but focus on graph generation tasks. In contrast, our approach specifically designs word-level editing actions in the instructions, tailored to enhance text generation. This targeted strategy renders our method more effective for text-generation tasks.

Overall, our key contributions are as follows:

- We introduce a novel programmer-interpreter method that enhances LLM in low-resource cross-domain text generation tasks. This approach capitalizes on the programmer's ability to encode task-specific knowledge and the interpreter's prowess in domain generalization.
- We design editing operations optimized for text generation tasks, leading to substantial text quality improvements by simply prompting the LLMs with action instructions.
- In scenarios where training and test data span different domains, our comprehensive empirical studies confirm that the method outperforms all existing LLM post-editing baselines in low-resource MT and LF-to-Text.

2 Programmer-Interpreter Approach

The objective in LF-to-text and MT tasks using LLMs is to generate a high-quality output text $\boldsymbol{y}$, denoted as $\boldsymbol{y}' = \arg\max_{\boldsymbol{y}\in\mathcal{Y}} P(\boldsymbol{y}|\boldsymbol{x}, \mathcal{C})$, given an input $\boldsymbol{x}$ (e.g., LF, source-language utterance) and an exemplar pool $\mathcal{C} = \{(\boldsymbol{x}_j, \boldsymbol{y}_j, \boldsymbol{y}_j^*, \boldsymbol{a}_j^*)\}_{j=1}^{|\mathcal{C}|}$. Here, $\boldsymbol{x}_i$ and $\boldsymbol{y}_j$ are the ground truth input-output pairs, $\boldsymbol{y}_j^*$ is the imperfect translation of $\boldsymbol{x}_i$, and $\boldsymbol{a}_j^*$ represents the Oracle edit actions that can modify $\boldsymbol{y}_j^*$ into $\boldsymbol{y}_j$. Our approach focuses on achieving high-quality generation through iterative refinement of the initial output text produced by an LLM. Specifically, the iterative refinement framework includes three-parameterized modules: a Generator, a Programmer, and an Interpreter,[3]

$$P(\boldsymbol{y}^t|\boldsymbol{x}, \mathcal{C}) = \overbrace{P(\boldsymbol{y}^0|\boldsymbol{x}, M(\cdot))}^{Generator} \times \tag{1}$$

$$\sum_{\{\boldsymbol{a},\boldsymbol{y}\}}^{t-1} \prod_{i=0}^{t-1} (\overbrace{P(\boldsymbol{y}^{i+1}|\boldsymbol{a}^i, \boldsymbol{y}^i, \boldsymbol{x}, A(\cdot))}^{Interpreter} \times \overbrace{P(\boldsymbol{a}^i|\boldsymbol{y}^i, \boldsymbol{x}))}^{Programmer} \tag{2}$$

The Generator corresponds to the LLM (e.g. GPT-3.5, GPT-4). It produces the initial output text, $\boldsymbol{y}^0$, given the input $\boldsymbol{x}$, a set of examples retrieved by the function $M(\boldsymbol{x}, \mathcal{C})$ when performing in-context learning. The Programmer, a module that creates editing actions $\boldsymbol{a}^i$ given $\boldsymbol{x}$ and the current imperfect output $\boldsymbol{y}^i$, is a pre-trained Sequence-to-Sequence (Sutskever et al., 2014) language model, such as mT5 (Xue et al., 2021) or flan-T5 (Chung et al., 2022), fine-tuned on a synthetic dataset. The Interpreter, essentially also an LLM, refines the imperfect intermediate output $\boldsymbol{y}^i$ by processing instructions that incorporate predicted editing actions and few-shot editing examples, retrieved via the function $A(\boldsymbol{x}, \mathcal{C})$. Please note that the Programmer has much fewer parameters than the LLM used by the Generator and Interpreter. After several iterative refinements, we arrive at the final output $\boldsymbol{y}^t$ generated by the LLM. During generation, we assume no access to the parameters of the LLMs but only obtain the output text by providing prompting instructions. The implementation details of each module are as follows:

Generator. To generate the initial output, we supply a prompt composed of a few-shot set of exemplar pairs, denoted as $M(\boldsymbol{x}, \mathcal{C}) = \{(\boldsymbol{x}_j, \boldsymbol{y}_j)\}_{j=1}^{m}$, selected from a pool of reference pairs $\mathcal{C}$. This is accompanied by an instruction prompting the LLM to produce output $\boldsymbol{y}^0$ based on the input $\boldsymbol{x}$. The retrieval function identifies the closest pairs by calculating the cosine similarity of TF-IDF features between $\boldsymbol{x}$ and other instances of $\boldsymbol{x}$ in $\mathcal{C}$.

[3]To save space, we simplify the marginalization notation.

Programmer. After obtaining the initial or intermediate output $\boldsymbol{y}^i$ from either the Generator or the Interpreter, we combine the input $\boldsymbol{x}$ and $\boldsymbol{y}^i$ into a single sequence and feed it to the Programmer to generate a sequence of edit actions $\boldsymbol{a}^i$. We create a synthetic training set $\mathcal{T}$, extracted from the example pool $\mathcal{C}$, for fine-tuning the Programmer. Each pair in $\mathcal{T}$ is defined as $(\boldsymbol{x}_{concat}, \boldsymbol{a}^*)$, where $\boldsymbol{x}_{concat}$ is the concatenated sequence of $\boldsymbol{x}$ and $\boldsymbol{y}^*$, serving as the input for the Programmer. The output $\boldsymbol{a}^*$ is the sequence of Oracle edit actions, synthetically generated based on the reference pairs in $\mathcal{C}$. For each reference $\boldsymbol{y} \in \mathcal{C}$, we calculate the word-level edit distance to the imperfect translation $\boldsymbol{y}^*$, generating intermediate edit actions. Only *INSERT*-word and *DELETE*-word actions are retained in the sequence, forming the final training sequence $\boldsymbol{a}^*$ for the Programmer. If $\boldsymbol{y}^*$ is identical to the reference $\boldsymbol{y}$, the action is labeled as "*NoAction*", indicating that no refinement is needed for that instance. Unlike PiVe, which generates the imperfect translation $\boldsymbol{y}^*$ by scrambling the original $\boldsymbol{y}$, we directly use the initial output $\boldsymbol{y}^0$ from the Generator as $\boldsymbol{y}^*$ in both $\mathcal{C}$ and $\mathcal{T}$. This approach enables the Programmer to learn an action distribution that more effectively corrects translation errors from LLMs.

Interpreter. To edit the intermediate output $\boldsymbol{y}^i$, we engage the LLM in the Interpreter role by providing it with prompting instructions. Given the edit instructions $\boldsymbol{a}^i$ and a pair $(\boldsymbol{y}^i, \boldsymbol{x})$, the LLM can *INSERT* or *DELETE* words *in order* to generate the modified text $\boldsymbol{y}^{i+1}$. We also incorporate a few-shot examples that demonstrate editing procedures, extracted from $\mathcal{C}$ and denoted as $A(\boldsymbol{x}, \mathcal{C}) = \{(\boldsymbol{x}_j, \boldsymbol{y}_j, \boldsymbol{y}_j^*, \boldsymbol{a}_j^*)\}_{j=1}^n$. These examples are selected based on the cosine similarity between the TF-IDF features of $\boldsymbol{x}$ and those in $\mathcal{C}$. Furthermore, to mimic action prediction errors from the Programmer, we adopt an *adversarial in-context learning* strategy, similar to the approach in Zhuo et al. (2023). This involves corrupting the action sequence by deleting Oracle actions with a certain probability $d\%$. If an action is not deleted, we swap it with other actions from $\mathcal{C}$ at the same probability $d\%$. Through this manipulation, we have discovered that the LLM's exceptional text generalization ability enables it to effectively comprehend the editing instructions. As a result, it can generate high-quality text after performing the necessary edits if the predicted actions from the Programmer are sufficiently accurate. See Figures 2 and 3 in the Appendix for zero/few-shot instruction examples.

3 Experiments

Setup. In our experiments, we default to using GPT-3.5-turbo-0301 as the LLM for the Generator in both the zero-shot and few-shot settings. For the Interpreter, we use GPT-3.5-turbo-0301 in the zero-shot setting and GPT-3.5-turbo-16k[4] in the few-shot setting. For the Generator, we use 0 and 5 shots, respectively, for MT and LF-to-Text. For the Interpreter, we apply 10 shots for MT and 5 shots for LF-to-Text, with a 50% action corruption probability. For the MT and LF-to-Text tasks, we employ mT5-base and flan-T5-base as the backbones of the Programmers, respectively. These backbone choices are driven by our emphasis on a computationally efficient setup, ensuring the models fit within an Nvidia V100 with 16GB memory. We train our programmers with a development set to select the optimal model. Our search for the best learning rate includes [5e-5, 1e-4, 2e-4], while the range of epochs considered is [5, 10, 20], with batch sizes of 8. GPTs require no fine-tuning. Each generation of 1096 tokens costs approximately $0.0015. For Self-Correct and Self-Refine, we perform five editing iterations. Prog-Refine and Algo-Refine stop when more than 95% of action is 'NoAction'.

Datasets. To simulate low-data scenarios, in the context of **MT**, we utilize a Kashmiri-English dataset from IndicTrans2 (Gala et al., 2023). Since Kashmiri is a notably low-resource language, translating it poses a formidable challenge for LLMs. The dataset provides 26,016 training pairs, which we use to generate synthetic data for action generation. The development set consists of 997 pairs. The dataset includes two distinct test sets, GEN and CONV, with 1,024 and 1,503 pairs, respectively. Each of the training, development, and test sets originates from different domains. For **LF-to-Text**, we employ the AMR-LDC2.0[5] dataset, which contains 22,550 AMR-English pairs for training and 1,368 pairs for development. For testing, we turn to a separate dataset, Bio-AMR[6], which offers 500 pairs in a different domain. Likewise, the AMR-to-Text task poses a low-resource challenge for LLMs.

[4]https://platform.openai.com/docs/models/gpt-3-5-turbo
[5]https://catalog.ldc.upenn.edu/LDC2017T10
[6]https://amr.isi.edu/download.html

	MT (Kashmiri to English)						LF-to-Text (AMR to English)		
	GEN			CONV			Bio-AMR		
Method	BLEU	BERT	ChrF++	BLEU	BERT	ChrF++	BLEU	BERT	ChrF++
Fine-tuned mT5/flan-T5	**16.58**	89.32	41.77	13.19	88.83	33.03	9.27	87.90	41.06
GPT-3.5									
Initial	9.21	87.29	34.30	5.92	87.24	26.23	9.63	88.57	43.98
Self-Correct	13.11	89.02	38.98	12.73	89.61	33.76	11.64	**89.44**	46.05
Algo-Refine	8.40	86.92	39.66	6.29	87.31	32.21	7.72	86.64	43.39
Self-Refine	8.13	86.54	31.78	4.73	86.55	24.13	8.67	87.34	39.63
Prog-Refine (Zero-shot Act.)	13.81	88.58	39.00	12.09	89.41	33.41	11.43	89.30	45.44
Prog-Refine (Few-shot Act.)	16.32	**90.36**	**42.44**	**14.78**	**90.19**	**35.48**	**13.64**	89.27	**47.69**
Prog-Refine (ORACLE)	43.48	92.11	65.29	42.42	93.00	42.42	27.77	90.01	52.86

Table 1: The main results of MT on GEN and CONV test sets, and LF-to-Text on Bio-AMR test set.

Baselines. We evaluate our approach, Prog-Refine, which utilizes zero-shot action exemplars (Zero-shot Act.) and few-shot action exemplars (Few-shot Act.) for Interpreters, against five baseline methods and an ORACLE setting

i) **Fine-tuned Models** include mT5-base for MT and flan-T5-base for LF-to-Text generation, both of which are fine-tuned on the training set consisting of pairs $(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{C}$. These baseline models do not perform any refinement.

ii) **GPT-3.5 + Initial** simply applies the GPT-3.5 as the Generator to obtain the text without any further refinement.

iii) **GPT-3.5 + Self-Correct (Welleck et al., 2022)** fine-tunes smaller models to be the Interpreter, fixing the output errors of the large models given the feedback. Here, we supply the edit actions produced by our Programmer as feedback to the fine-tuned Interpreters. These Interpreters are also built upon mT5-base or flan-T5-base.

iv) **GPT-3.5 + Algo-Refine** directly 'Insert' or 'Delete' specific words in certain positions of the generated text instead of using an Interpreter to rewrite. Therefore, in this baseline, we also apply the Interpreter to predict the indices of words for actions. This method is prevalent in the MT literature; e.g. see Vu and Haffari (2018).

v) **GPT-3.5 + Self-Refine (Madaan et al., 2023)** leverages an LLM to provide feedback for its own output, enabling self-refinement without the need for additional fine-tuning.

vi) **GPT-3.5 + Prog-Refine (ORACLE)** applies the ORACLE actions generated by comparing the reference in the test set with the initial output of the Generator, allowing for optimal refinement.

Evaluation Metrics. For LF-to-Text and MT tasks, we utilize three evaluation metrics to assess the quality of the final output text generated by the Programmer-Interpreter framework: BLEU (Papineni et al., 2002), BERTScore (Zhang et al.) and Chrf++ (Popović, 2017).

3.1 Main Results and Analysis

Table 1 shows that *GPT-3.5 + Prog-Refine* notably boosts the Generator's performance (i.e., *GPT-3.5 + Initial*), underlining our method's effectiveness in cross-domain scenarios by enhancing initial GPT-3.5 outputs. Moreover, the few-shot setting (Few-shot Act.) significantly outperforms both the zero-shot (Zero-shot Act.) setting and all other refinement baselines. It's also noteworthy that applying ORACLE action to our method can lead to a roughly 30-point increase in BLEU score, suggesting substantial potential for improvement in our approach. In comparison, *Self-Refine* shows minimal improvement, possibly due to its limited integration of task-specific knowledge. *Algo-Refine* inconsistently improves the initial text, lacking the robustness seen in our method. We note that rewriting Interpreters, as in our approach and Self-Correct, can eliminate invalid actions, thus enhancing editing quality. However, *Algo-Refine* does not possess this capability and is susceptible to incorrect feedback actions. The *Self-Correct* method, using a fine-tuned Interpreter, along with fine-tuned mT5/flan-T5 models, demonstrates better performance than other baselines across various tasks. This underscores the importance of learning task-specific knowledge, especially in low-resource scenarios. Nonetheless, these methods face significant challenges in cross-domain applications, as further evidenced by our analysis in Table 4.

3.2 Ablation Study

Refinement Iterations. In Table 2, we observe that Prog-Refine significantly improves the initial output generated by the Generator. However, it only demonstrates marginal improvements in the subsequent outputs from the Interpreter, even after

MT (Kashmiri to English)

#Iter	BLEU	BERT	ChrF++	NoAct%
Iter 0	5.92	89.00	33.27	17.70
Iter 1	11.01	89.18	33.05	79.71
Iter 2	11.87	89.36	33.41	90.67
Iter 3	12.09	89.41	33.41	95.28
Iter 4	12.26	89.45	33.43	97.21
Iter 5	12.36	89.47	33.39	-

Table 2: The influence of multiple iterations on main results of MT using Prog-Refine (Zero-shot Act.) on CONV test set. NoAct%: The percentage of utterances requiring no refinement, as indicated by 'NoAction'.

MT (Kashmiri to English)

	BLEU	BERT	ChrF++
Initial	5.92	89.00	33.27
Edit: DEL, INS	12.36	89.47	33.39
Edit: DEL	12.27	89.42	33.21
Edit: INS	12.18	89.45	33.42
Unordered: DEL, INS	7.12	87.86	29.21
Unordered: DEL	6.52	87.51	26.46
Unordered: INS	7.14	88.04	30.38

Table 3: The results of MT on CONV test set with different types of actions. Edit: Actions are generated based on edit distance. Unordered: Actions without any specific order. INS: Insertion. DEL: Deletion.

four additional iterations. We hypothesize that this limited improvement may be attributed to training the model solely on synthetic data generated by the Generator, so the action distribution might be different to the ones for modifying the output of the Interpreter in the subsequent iterations.

Action Types. We further examine the impact of solely utilizing one type of action and the influences of disregarding the sequence of these actions. In the setting with unordered actions, oracle actions are generated by simply contrasting the differences within two sentences' unordered sets of words. As depicted in Table 3, the *Delete* and *Insert* actions, when used individually, can deliver performance metrics on par with when they are combined. However, ignoring the order of actions can lead to a substantial decline in the refinement performance. This highlights that LLM editing methods like PiVe, which utilize unordered insertions, are not optimally suited for our tasks. Further analysis is in Appendix A.5.

Domain Discrepancy. As shown in Table 4, a domain shift dramatically impacts the performance of flan-T5 and Self-Correct. While both baseline models show markedly superior performance on the in-domain test set relative to our model, ours

LF-to-Text (AMR to English)

Method	BLEU	BERT	ChrF++
Fine-tuned flan-T5	**34.63**	**95.05**	**66.97**
GPT-3.5			
Initial	19.67	92.10	55.98
Self-Correct	34.49	94.68	66.81
Self-Refine	16.16	91.08	52.78
Prog-Refine	29.12	94.01	64.85

Table 4: LF-to-Text results on the in-domain LDC test.

LF-to-Text (AMR to English)

Rate	BLEU	BERT	ChrF++
0.0	12.06	89.31	46.23
0.2	12.35	**89.36**	46.49
0.5	**13.64**	89.27	**47.69**
1.0	11.97	89.32	46.13

Table 5: LF-to-Text results vary with different corruption probabilities for the action sequence in the adversarial in-context examples used for the Interpreter.

either surpasses or equals their performance in the cross-domain MT and AMR-to-Text test sets. This disparity in performance is likely due to the smaller models' limited cross-domain generalization. Similarly, in MT tasks, our preliminary experiments show that fine-tuned models achieve 30 BLEU in in-domain tests but only 16 and 13 in out-of-domain tests. For further details on domain discrepancies, see Appendix A.3.

Adversarial In-context Learning. Table 5 indicates 0.0 for no corruption and 1.0 for complete discarding of exemplar actions, leaving only $(\boldsymbol{x}_j, \boldsymbol{y}^* j, \boldsymbol{y} j)_{j=1}^{n}$. Rates between 0.0 and 1.0 represent partial corruption of Oracle actions. The results suggest that neither full application nor total corruption of Oracle actions is optimal. However, partial corruption leads to improved performance. Additionally, across all corruption rates, few-shot settings consistently outperform zero-shot settings.

4 Conclusions

We present a programmer-interpreter method that iteratively refines LLM outputs using edit actions from a fine-tuned programmer and an LLM interpreter. Our approach combines the task-specific encoding capacity of a fine-tuned model with the domain generalization strength of the LLM, incorporating specifically designed actions for text generation. The experiments confirm its efficacy, showing significant improvements in LLM-generated text quality for low-resource MT and LF-to-Text tasks. Moreover, our approach outperforms established baselines in cross-domain scenarios.

5 Limitations

This work has two primary limitations. First, in in-domain tests, our approach does not outperform smaller models, such as mT5 and flan-T5. Considering the performance improvements we observed when using ORACLE actions, we believe there is substantial potential to further enhance our method for text generation in the in-domain evaluation setting. Second, our approach requires internet transmission of prompt instructions to the servers of ChatGPT. This could potentially lead to a risk of privacy leakage, which is a critical concern in data-sensitive applications.

References

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. 2022. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*.

Jay Gala, Pranjal A Chitale, Raghavan AK, Sumanth Doddapaneni, Varun Gumma, Aswanth Kumar, Janki Nawale, Anupama Sujatha, Ratish Puduppully, Vivek Raghavan, et al. 2023. Indictrans2: Towards high-quality and accessible machine translation models for all 22 scheduled indian languages. *arXiv preprint arXiv:2305.16307*.

Jiuzhou Han, Nigel Collier, Wray Buntine, and Ehsan Shareghi. 2023. Pive: Prompting with iterative verification improving graph-based generative capability of llms. *arXiv preprint arXiv:2305.12392*.

Levon Haroutunian, Zhuang Li, Lucian Galescu, Philip Cohen, Raj Tumuluri, and Gholamreza Haffari. 2023. Reranking for natural language generation from logical forms: A study based on large language models. *arXiv preprint arXiv:2309.12294*.

Viet Dac Lai, Nghia Trung Ngo, Amir Pouran Ben Veyseh, Hieu Man, Franck Dernoncourt, Trung Bui, and Thien Huu Nguyen. 2023. Chatgpt beyond english: Towards a comprehensive evaluation of large language models in multilingual learning. *arXiv preprint arXiv:2304.05613*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Krishna Pillutla, Swabha Swayamdipta, Rowan Zellers, John Thickstun, Sean Welleck, Yejin Choi, and Zaïd Harchaoui. 2021. MAUVE: measuring the gap between neural text and human text using divergence frontiers. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 4816–4828.

Maja Popović. 2017. chrf++: words helping character n-grams. In *Proceedings of the second conference on machine translation*, pages 612–618.

Scott E. Reed and Nando de Freitas. 2016. Neural programmer-interpreters. In *International Conference on Learning Representations (ICLR)*.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27.

Thuy Vu and Gholamreza Haffari. 2018. Automatic post-editing of machine translation: A neural programmer-interpreter approach. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, pages 3048–3053.

Jindong Wang, HU Xixu, Wenxin Hou, Hao Chen, Runkai Zheng, Yidong Wang, Linyi Yang, Wei Ye, Haojun Huang, Xiubo Geng, et al. 2023. On the robustness of chatgpt: An adversarial and out-of-distribution perspective. In *ICLR 2023 Workshop on Trustworthy and Reliable Large-Scale Machine Learning Models*.

Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. 2022. Generating sequences by learning to self-correct. *arXiv preprint arXiv:2211.00053*.

Linting Xue, Noah Constant, Adam Roberts, Mihir Kale, Rami Al-Rfou, Aditya Siddhant, Aditya Barua, and Colin Raffel. 2021. mt5: A massively multilingual pre-trained text-to-text transformer. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 483–498.

Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Bing Yin, and Xia Hu. 2023. Harnessing the power of llms in practice: A survey on chatgpt and beyond. *arXiv preprint arXiv:2304.13712*.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with bert. In *International Conference on Learning Representations*.

Terry Yue Zhuo, Zhuang Li, Yujin Huang, Fatemeh Shiri, Weiqing Wang, Gholamreza Haffari, and Yuan-Fang Li. 2023. On robustness of prompt-based semantic parsing with large pre-trained language model: An empirical study on codex. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pages 1090–1102.

A Appendix

A.1 Prompt Example for Editing Text

Figures 2 and 3 depict the exemplary zero/few-shot prompt employed in LF-to-Text.

```
You are a AMR translator and you are proficient with both AMR and English.

You are given the following AMR logical form:

( q / quote-01 : arg0 ( r / report ) : arg2 ( a2 / and : op1 ( g / government-organization :
arg0-of ( g4 / govern-01 : arg1 ( c / country : wiki `` greece '' : name ( n2 / name : op1 ``
greece '' ) ) ) ) : op2 ( g2 / government-organization : arg0-of ( g5 / govern-01 : arg1 ( c2 /
country : wiki `` turkey '' : name ( n4 / name : op1 `` turkey '' ) ) ) ) : op3 ( g3 /
government-organization : arg0-of ( g6 / govern-01 : arg1 ( c3 / country : wiki `` belarus '' :
name ( n6 / name : op1 `` belarus '' ) ) ) ) ) : arg3 ( a / acknowledge-01 : arg0 a2 : arg1 (
m / miss-02 : arg0 a2 : arg1 ( d / deadline ) ) ) )

You are given the following English translation:

the report quotes the governments of greece , turkey and belarus acknowledging that
they missed the deadline .

Please improve the above English translation using the following edit rewriting actions:

DELETE : quotes
INSERT : quoted
INSERT : as
DELETE : they
...
INSERT : missed

Please only show the English sentence:
```

Figure 2: The zero-shot exemplary prompt for LF-to-Text.

```
Here are the edit rewriting examples:

### Example 1:

You are a AMR translator and you are proficient with both AMR and English.
You are given the following AMR source logical form:

( i / increase-01 : arg1 ( e / express-03 : arg2 ( p / protein ) ) : arg2 ( p2 / product-of : op1 10 )
: arg1-of ( s / statistical-test-91 : arg2 ( l / less-than : op1 0.05 ) ) )

You are given the following English translation:

there was a statistically significant increase in protein expression ( 10 fold , p < 0.05 ) .

Please improve the above English translation using the following edit rewriting actions:

DELETE "was" from the translation
DELETE "significant" from the translation
DELETE "fold" from the translation
DELETE "increase" from the translation

Please provide a fluent English sentence that is semantically equivalent to the AMR logical
form after editing its corresponding English translation.

Improved English sentence:

protein expression increased 10-fold ( p < 0.05 ) .

### Example 2:
  ...
### Example 3:
  ...
### Example 4:
  ...
```

Figure 3: The few-shot exemplary prompt for LF-to-Text.

A.2 Adaption of Self-Corrector

In our experiment, we adapted the implementation of the Self-Corrector to better suit our specific requirements. To customize it for our context, we constructed the training set for the Self-Corrector's Interpreter as follows: the input consists of a concatenation of Kashrimi/AMR, text produced by the Generator, and edit actions. The output, on the other hand, is the ground truth text. For a fair comparison with our approach and to minimize training and data collection expenses, models are trained only during the first iteration. Additionally, the generation of the training set solely utilizes text from the Generator in the initial iteration, without using text from the Interpreter in subsequent refinement iterations.

splits compared	KL-div ↓	MAUVE ↑
train, dev	2.23	0.006
dev, test$_{gen}$	1.97	0.231
train, test$_{gen}$	1.94	0.005
dev, test$_{conv}$	2.97	0.040
train, test$_{conv}$	2.98	0.007

Table 6: Measures of domain difference across different splits of the machine translation datasets. KL-divergence scores are calculated for the English sentences in each data split, with additive smoothing ($\alpha = 1 \times 10^{-4}$). For MAUVE, 5000 sentences are sampled from the training set.

splits compared	KL-div ↓	MAUVE ↑
train, dev	2.00	0.512
dev, test$_{i.d.}$	2.39	0.327
train, test$_{i.d.}$	1.97	0.342
dev, test$_{bio}$	6.01	0.004
train, test$_{bio}$	5.48	0.004

Table 7: Measures of domain difference across different splits of the AMR dataset. KL-divergence scores are calculated for the English sentences in each data split, with additive smoothing ($\alpha = 1 \times 10^{-4}$). For MAUVE, 5000 sentences are sampled from the training set.

A.3 Measures of Domain Discrepancy

Tables 6 and 7 present domain discrepancies for the training/development/testing sets for the MT and LF-to-text generation tasks. The domain discrepancy measures include the KL-divergence (based on the unigram distributions) and MAUVE (Pillutla et al., 2021). KL-divergence scores are higher when two distributions are more different from each other. MAUVE scores, which have a range (0,1), are lower when two distributions are more different from each other.

Based on Table 6, we observe that the domain of test-gen is closer to the training set compared to that of the test-conv. This is pronounced in higher KL-divergence and lower MAUVE numbers for the test-conv compared to test-gen, with respect to

	INSERT	DELETE	Total
MT	33.64	83.73	62.57
NLG	24.52	60.48	44.90

Table 8: The F1 scores of comparing the predicted actions with the ORACLE actions in the GEN test set.

LF-to-Text (AMR to English)

	BLEU	BERT	ChrF++
GPT-3.5-turbo-16k	11.43	89.30	45.44
GPT-4-turbo	11.72	89.36	45.58

Table 9: LF-to-Text results of Prog-Refine (Zero-shot Act.) in zero-shot setting with different LLMs as Interpreters.

the training set.

Based on Table 7 , we observe a higher difference for the domain of the biology-AMR test compared to the LDC2.0-AMR test set, with respect to the training/development sets of the LDC2.0-AMR dataset. This is pronounced in larger KL divergence and lower MAUVE numbers compared to those for the LDC2.0-AMR test set.

A.4 F1 Definition for Action Prediction

$$F1 = 2 \times \frac{P_{act} \times R_{act}}{P_{act} + R_{act}} \tag{3}$$

Here, P_{act} represents action precision, defined as the ratio of predicted actions present in the reference action sequence to the total number of predicted actions. R_{act} denotes action recall, which is the ratio of predicted actions that appear in the reference action sequence to the total number of actions in the reference sequence. The F1 score, thus, provides a harmonious mean of these two metrics.

A.5 F1 for Action Prediction

Table 8 reveals that predicting INSERT actions is a relatively easier task compared to predicting DELETE actions. This observation is reasonable since the Programmer only needs to learn how to DELETE words from the text with a fixed vocabulary, whereas, for INSERT actions, the Programmer must learn to INSERT arbitrary words.

A.6 Comparing GPT-4 and GPT-3.5 as Interpreters

Table 9 illustrates the performance differences in the LF-to-Text task when using GPT-4 and GPT-3.5 as Interpreters for Prog-Refine (Zero-shot Act.). While GPT-4 offers a slight performance boost, the improvement is not substantial, amounting to only a 0.3 increase in BLEU score. Moreover, this comes at a higher cost of 0.06 per 1000 characters, compared to 0.0015 for GPT-3.5.