Table 1: Generalization errors of SED and LGRL with the same number of expanded programs per sample.

Debugger beam size	Debugger edit steps	# of expanded programs	SED	LGRL
32	25	141	**16.64%**	18.52%
64	25	229	**15.72%**	17.64%
32	100	393	**15.40%**	17.00%
64	100	687	**14.60%**	15.92%

We thank reviewers for constructive comments! We first address the common confusion, then the individual questions.

Differences with [20]. We would like to clarify that in [20], they didn't study how to incorporate a debugger component to improve the program synthesis results. Instead, they focus on repairing programs that are generated by randomly mutating the ground truth programs, thus they assume that the wrong programs are already syntactically similar to the ground truth. On the other hand, we demonstrate that the debugger component is not only helpful for the program repair task itself, but also improves the program synthesis performance. In particular, even if the program synthesizer generates programs that are syntactically far from the ground truth, the debugger may still be able to predict alternative programs that satisfy the specification. We will add a more detailed discussion in our revision.

Comparison of SED and the synthesizer with the same execution budget. Based on R1 and R3's questions, we compute the number of programs expanded by SED, and run the synthesizer with the same beam size. As shown in Table 1, SED still achieves better results than the synthesizer in this case. We will discuss more details in our revision.

R1. See the common response about the comparison with the synthesizer with the same execution budget, and the comparison with [20]. We will discuss the related work in our revision. Specifically, S3 focuses on designing search heuristics to improve the efficiency and generalizability of enumerative search over potential bug fixes, without any learning techniques. SED instead incorporates a neural debugger into the neural program synthesis framework.

R2. To show that the effect of TraceEmbed is not just due to the increase of the model size, as suggested by R2, we added experiments to increase the hidden size of IOEmbed and program encoder without TraceEmbed, so that the numbers of parameters for models with/without TraceEmbed are similar. However, the results are even worse than the smaller counterpart; e.g., with EGNPS, the generalization error is 13%, compared to 11.52% in the submission.

See the common response for the comparison with [20]. For L26, according to Table 2 in [3], the statements in a While loop are only runned once during the While body prediction, and all remaining iterations are executed after finishing the While loop generation, thus the partial execution results of a While body are essentially the same as an If-statement. In Figure 2, the embedded IOs are inputs to both TraceEmbed and EditDecoder, and we use the same colors to denote the same embeddings. $ep_i||q_i$ means the concatenation. About Lines 196–197, we mean that comparing the results of all 4 models, the lowest error is achieved by the model with TraceEmbed and trained on similar mutations. In Lines 197-198, we discussed that a potential reason why TraceEmbed harming the model when it's tested out of distribution may be the larger model sizes, which could lead to overfitting. We will clarify these points in our revision.

R3. Note that EGNPS achieves 91.68% generalization accuracy with an ensemble of 15 models, instead of a single model. We added experiments to build upon EGNPS with a higher accuracy, and SED also further improves the results. For example, using a single EGNPS synthesizer, the generalization accuracy is 89.52%, higher than 86.04% in the EGNPS paper. With the ensemble as the synthesizer, SED still improves the accuracy by 0.52%.

An "if" conditional guard is also aligned to its two branches. About different forms of pooling and using a GNN, we tried learning an attention map to aggregate the embeddings of actions, adding more types of graph edges, etc. However, these variants are significantly slower to train and do not perform well, so we did not include them in our submission.

Studying more advanced search techniques is not the main focus of this work, but we agree that this is a promising future direction. The fact that SED works even with a simple search justifies the overall design of our framework. We will provide more discussion about work on generate-and-test program synthesis. Specifically, Schkufza et al. performed MCMC sampling for superoptimization, while we train a neural network for program synthesis from input-output specifications. Laich et. al. and Solar Lezama et. al. propose counter-example generation approaches that require symbolic solvers, while we focus on program synthesis with a limited number of input-output examples as specification.

R4. See the common response about the differences with [20]. We would like to clarify that our main focus is not the program repair task itself. Instead, we propose to integrate a neural program repair component to improve the performance of a neural program synthesizer. The reviewer questions whether evaluation on Karel is sufficient to demonstrate the effectiveness of our approach. We would like to note that most existing work on input-output program synthesis was evaluated on domain-specific languages (DSLs), such as Karel (e.g., [2, 3, 4, 18]) and FlashFill (e.g., [5, 10, 16, 23]), and Karel is already among the most full-fledged DSLs with conditional and loop semantics. Therefore, while we agree that evaluating on more popular programming languages could strengthen this work, we believe our work is pushing forward along one direction that the neural program synthesis community is working on.