A Solver-in-the-Loop Framework for Improving LLMs on Answer Set Programming for Logic Puzzle Solving

Timo Pierre Schrader[1,2], **Lukas Lange**[1], **Tobias Kaminski**[1], **Simon Razniewski**[3], **Annemarie Friedrich**[2]
[1]Bosch Center for AI, Renningen, Germany
[2]University of Augsburg, Germany
[3]ScaDS.AI & TU Dresden, Germany
timo.schrader@de.bosch.com

Abstract

The rise of large language models (LLMs) has sparked interest in coding assistants. While general-purpose programming languages are well supported, generating code for domain-specific languages remains a challenging problem for LLMs. In this paper, we focus on the LLM-based generation of code for Answer Set Programming (ASP), a particularly effective approach for finding solutions to combinatorial search problems. The effectiveness of LLMs in ASP code generation is currently hindered by the limited number of examples seen during their initial pre-training phase.

In this paper, we introduce a novel ASP-solver-in-the-loop approach for solver-guided instruction-tuning of LLMs to addressing the highly complex semantic parsing task inherent in ASP code generation. Our method only requires problem specifications in natural language and their solutions. Specifically, we sample ASP statements for program continuations from LLMs for unriddling logic puzzles. Leveraging the special property of declarative ASP programming that partial encodings increasingly narrow down the solution space, we categorize them into chosen and rejected instances based on solver feedback. We then apply supervised fine-tuning to train LLMs on the curated data and further improve robustness using a solver-guided search that includes best-of-N sampling. Our experiments demonstrate consistent improvements in two distinct prompting settings on two datasets.

Code — https://bos.ch/1qzx1jj

Introduction

One of the primary productive applications of large language models (LLMs) is currently their use as coding assistants (Jiang et al. 2024; Gu 2023; Ugare et al. 2024), supporting software developers by taking over tedious and repetitive programming workflows. However, current systems typically target popular and general-purpose programming languages such as JavaScript, C++, and Python.

Many real-life problems, however, require the use of domain-specific programming languages tailored towards specific problem types. Various studies on LLM-based code generation have been conducted on programming languages such as PROLOG or PDDL (*planning domain definition language*, McDermott et al. (1998)). Existing approaches rely on prompt engineering or create complex planning and reasoning systems (Yang, Chen, and Tam 2024; Stein et al. 2023; Vyas et al. 2024). Yet, for many programming languages, LLM-based generations remain understudied.

In this work, we focus on *Answer Set Programming* (ASP, Gelfond and Lifschitz 1988; Marek and Truszczyński 1999), which can be used for finding solutions to combinatorial search problems. ASP is a well-known and powerful approach for declarative problem solving. It offers an expressive input language, provides well-optimized solvers, and has been applied to a wide range of industrial problems such as generating software test cases, robotics, and configuration problems (Falkner et al. 2018).

ASP is highly suited for solving many real-life problems, such as scheduling in factory plants, configuring electrical circuits (ECUs), or assignment problems as shown in Figure 1. These problems and their constraints can often be described by domain experts in natural language. However, domain specialists often lack the necessary ASP programming skills. Out-of-the-box LLMs frequently struggle to solve these problems using chain-of-thought reasoning, particularly as problem size and complexity grow. Neuro-symbolic solutions that combine LLMs and symbolic solvers are a promising alternative (Olausson et al. 2023; Schrader et al. 2024). These systems translate fine-grained natural language statements into executable and semantically correct code for which a solver computes the solution.

First explorations addressing ASP code generation include the LLASP dataset (Coppolillo et al. 2024) that provides ASP statements only in isolation without a larger contextual problem. Ishay, Yang, and Lee (2023) engineer a highly specific prompt pipeline for a particular dataset for commercial GPT-based models only. For processing sensitive data on-premise, the use of open-weight LLMs is an alternative. **Yet, as we show in this paper, state-of-the-art open-weight LLMs still mostly fall short on producing correct ASP encodings, even when being provided with strong prompt guidance.** Our goal is to improve trainable state-of-the-art LLMs on the task of ASP code generation.

In this paper, we present a novel and efficient method that uses an ASP solver in the loop both for creating high-quality training data and for guiding the generation process with a solver-based reward. Our ultimate goal is to develop LLMs that excel at ASP coding for assignment problems. We focus

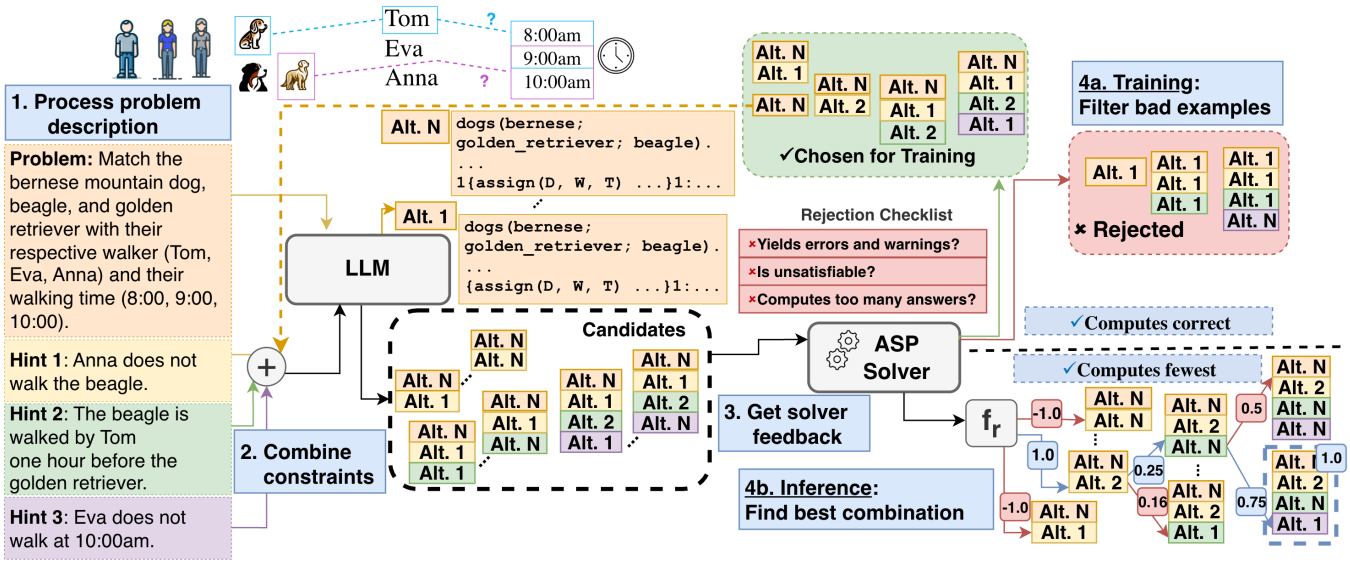

Figure 1: We use the feedback from an ASP solver to assess LLM-generated ASP statements. **(1)** We take the problem description and let the LLM output N alternatives for it (**generate** part of an ASP encoding). We then judge the correctness of these encodings using an ASP solver which checks for errors as well as for the expected answer size. **(2)** We incrementally combine the constraints of the problem into larger inputs to the LLM and obtain longer partial encodings (**check** part of an ASP encoding). **(3)** The ASP solver judges the correctness of each partial encoding containing the base encoding and at least one constraint. **(4a)** For training, we select instances from a chosen set that corresponds to instances with positive solver feedback. **(4b)** During inference, where the ground truth is not known, we score every partial encoding using a reward function that captures errors as well as the number of produced answer sets with preference for a lower number of returned answers. Additionally, we implement fallback mechanisms that regenerate encodings or backtrack in the graph if all alternatives were judged with a negative score.

on solving grid-based puzzles as a proxy task since they reflect assignment problems that are one of the main use cases of ASP. The LLM-generated ASP encodings are automatically evaluated by an ASP solver and classified as either *chosen* or *rejected*. Our solver-in-the-loop setup leverages the specific properties of ASP as a declarative approach, i.e., that the order in which rules and constraints are added does not alter the semantics of the final encoding. Hence, **it allows for fine-grained evaluation of partial encodings regardless of the order in which they are added by checking if the ground truth solution can still be derived from the partial encoding.** We also make use of the fact that ASP encodings are usually divided into clearly defined *generate* and *check* parts. This enables us to (re-)combine partial ASP encodings and to verify intermediate generation steps by checking for both syntax errors as well as for a reduction of the answer space after each added constraint. This specific ASP property allows for a substantially more fine-grained evaluation than evaluating code of other programming languages, e.g., in Python, semantic feedback from the interpreter can only be obtained for a full block without being able to check intermediate steps (Peng et al. 2024).

Using the chosen continuations, we apply supervised fine-tuning (SFT) via causal language modeling. Additionally, we leverage the same solver-in-the-loop setup to further improve the ASP code generation during test-time. We introduce a novel reward function that leverages solver feedback to rank a set of N alternatives for the same input and triggers fallback mechanisms in case of errors.

We evaluate our trained models with and without reward-based inference for solving logic puzzles in two distinct settings: First, in a **few-shot setting**, we only provide basic instructions as well as a few examples in the input prompt. This reflects a realistic setting in which the type of problem and dataset format is not clearly defined beforehand. Second, a **prompt-engineered setting** (Ishay, Yang, and Lee 2023) assumes that everything about the input problem and how it could be generally modeled in ASP is already fully known.

Our contributions are as follows: (1) We demonstrate that state-of-the-art open-weight LLMs struggle with ASP code generation, even under highly prompt-engineered conditions. (2) We create silver-standard training data by sampling partial ASP encodings from an LLM for given combinatorial problems and filtering them based on feedback from an ASP solver that includes whether the encoding is erroneous or whether the ground truth answer can be derived from it. (3) We show that applying SFT on the chosen instances improves the performance of open-weight LLMs of various sizes in generating ASP encodings for grid-based puzzles, especially for problems of higher complexity in both prompt settings. (4) We demonstrate that the solver can steer the ASP generation process successfully during inference.

Answer Set Programming

ASP, a form of declarative programming focusing on difficult, primarily NP-hard search problems, is based on the stable model semantics proposed by Gelfond and Lifschitz (1988). Our short introduction follows Lifschitz (2008). The purpose of ASP is to find answer sets consisting of sets of instantiated first-order atoms (e.g., `walk(beagle, eva, 8)`) that satisfy all given rules and constraints. We exemplify ASP with the puzzle of matching dogs with their walkers and times for walking (cf. Figure 1).

Rules define if-then statements written in PROLOG-style syntax using n-ary predicates. The right-hand side (*rule body*) is the premise and the left-hand side the conclusion (*rule head*). "If the beagle is walked by Tom, then the golden retriever will be walked by Eva at 9 am." is encoded as:

```
walk(golden, eva, 9) :-
              walk(beagle, tom, _).
```

Constraints eliminate undesired solutions by disallowing certain combinations of atoms to be true simultaneously. They only have a rule body, i.e., no head. A constraint requires at least 1 atom to evaluate to `false`. For example, "Anna walks her dog later than Eva." is encoded as:

```
:- walk(_, anna, T1), walk(_, eva, T2),
   not T1 > T2.
```

Unlike constraints, **choice rules** generate answer set candidates instead of filtering them. Assuming that the ASP program already contains atoms specifying a set of entities (here: dogs, persons, and times), the following rule states that "For each time T, some dog D is walked by some person P." The statement `dog(beagle;golden;bernese)` is an ASP shorthand for `dog(beagle). dog(golden). dog(bernese).`

```
person(tom;eva;anna).
dog(beagle;golden;bernese).
time(8;9;10).

1 { walk(D, P, T) : dog(D), person(P) } 1
   :- time(T).
```

Solutions in ASP. An ASP solver calculates all answer sets, corresponding to minimal sets of derivable atoms that are valid interpretations of all rules. More technically, a solution of an ASP program Π is a special truth assignment to all atoms that occur in the grounding of the program, i.e., a stable model, and the effect of adding a constraint to a program can lead to its elimination, i.e., constraints reduce the set of solutions monotonically.

Instruction Tuning for ASP

We present a novel instruction-tuning method for ASP code generation in LLMs. We prompt an LLM $\mathcal{M}_S$ to generate ASP code and classify the results into "chosen" and "rejected" samples using an ASP solver. The chosen data can be used with causal language modeling to train an ASP-specific model $\mathcal{M}_{ASP}$ based on the reference model $\mathcal{M}_S$.

Task Definition

The problems addressed in this paper are grid-based puzzle instances of the form $\mathcal{I} = \{\mathcal{D}, \mathcal{E}, \mathcal{H}, \mathcal{S}\}$. $\mathcal{D}$ refers to the natural language problem description, $\mathcal{E}$ to a set of entities and their types (e.g., dogs, persons, and times), $\mathcal{H}$ to a set of natural language hints (or clues) that constrain the answer space ("Clue: The beagle is walked one hour before the poodle."), and $\mathcal{S}$ refers to the correct solution assignment. To solve this, the LLM $\mathcal{M}_{\text{ASP}}$ has to parse the input problem $\{\mathcal{D}, \mathcal{E}, \mathcal{H}\}$ specified as natural language text into a valid ASP encoding Π that contains ASP encodings for entities, (choice) rules, and constraints. A solver will then compute the solution $\mathcal{S}$ given Π. In the following, we stick to the notations of Lifschitz (2008), using Γ to denote a partial ASP encoding that contains a proper subset of the ASP code in Π, i.e., $solution(\Pi = (\Gamma_1 \cup \cdots \cup \Gamma_n)) = \mathcal{S}$.

Sampling Trajectories

We define a *trajectory* $\mathcal{T}$ to be an alternating sequence of natural language inputs and ASP statements. A trajectory $\mathcal{T}$ starts with a prompt $P_{\mathcal{D},\mathcal{E}}$ comprising general information on ASP,[1] the textual problem description $\mathcal{D}$, and the set of entities $\mathcal{E}$. We feed $\mathcal{P}_{\mathcal{D},\mathcal{E}}$ into the LLM $\mathcal{M}_S$ to obtain ASP code $\Gamma_{\mathcal{C},\mathcal{E}}$, where $\mathcal{C}$ refers to the choice rule that initially generates all potential solutions representing the problem instance, which is then appended to the trajectory. Next, a natural language hint $h_i \in \mathcal{H}$ is appended to this trajectory and $\mathcal{M}_S$ is prompted again to obtain a partial encoding Γ_i. This results in trajectories of form $\mathcal{T} = \{\mathcal{P}_{\mathcal{D},\mathcal{E}}, \Gamma_{\mathcal{C},\mathcal{E}}, h_1, \Gamma_1, h_2, \Gamma_2, \ldots, h_n, \Gamma_n\}$ with n being the number of hints of the problem.

Classification of ASP Encodings

After processing k of the n hints, we obtain a trajectory $\mathcal{T}_k$ that contains k+1 ASP encodings (one for entities and choice rule and k for hints). We now combine all k+1 ASP encodings into the partial encoding $\Gamma_{\mathcal{C},\mathcal{E}} \cup \Gamma_1 \cup \ldots \Gamma_k$ and use the solver to compute its solution. We sample 5 completions from $\mathcal{M}_S$ for each step k simultaneously to get a diverse set of ASP encodings for the same input. Preliminary experiments showed that a temperature $t = 0.8$ for sampling provides a good balance between chosen and rejected responses. We then use an ASP solver to evaluate if the partial encoding Γ_k generates a solution that comprises the ground truth answer set $\mathcal{S}$. We consider Γ_k as a *chosen* response to input $\mathcal{T}_k = \{\mathcal{P}_{\mathcal{D},\mathcal{E}}, \Gamma_{\mathcal{C},\mathcal{E}}, h_1, \Gamma_1, h_2, \Gamma_2, \ldots, h_k\}$ if the solution of the partial program $\Gamma_{\mathcal{C},\mathcal{E}} \cup \Gamma_1 \cup \ldots \Gamma_k$ comprises the ground truth solution. If it produces only wrong answer sets, becomes unsatisfiable, or errors and warnings are returned by the solver, we consider Γ_k as *rejected*.

[1] This includes an instruction of how different variants of logical OR works in ASP, a hint to introduce helper predicates to perform arithmetics (e.g., on dates), a high-level explanation of the steps required to solve a grid-based puzzle in ASP as well as a basic choice rule.

Training Data Generation

We generate trajectories using depth-first search. At each step k, we consider at most two chosen responses. For each of them, we continue at step k+1 recursively until there are no hints left. To form pairs from both chosen and rejected responses at each step k, we take the Cartesian product between both sets. To put emphasis on instances that are difficult for current LLMs, we keep only instances for which there are at least one chosen and one rejected alternative. If we have a mix of chosen and rejected responses, we obtain either 1×4 or 2×3 preference pairs in each step. If there are only chosen or only rejected responses, the trajectory generation is continued at the next step without creating preference pairs. If all responses are rejected, the input to the next step becomes $\{\mathcal{P}_{\mathcal{D},\mathcal{E}}, \Gamma_{\mathcal{C},\mathcal{E}}, h_1, \Gamma_1, \ldots, h_{k-1}, \Gamma_{k-1}, h_{k+1}\}$, i.e., the rejected response and the hint prompt causing it are removed. This is possible because the hints in grid-based puzzles do not refer to each other.

Model Training

The chosen instances can be used to train LLMs using SFT with causal language modeling. Alternative training methods, such as DPO, could also be used to perform preference alignment using pairs of chosen and rejected instances. However, our initial experiments showed that SFT works slightly better than DPO in this context.

Test-Time Sampling for ASP

In this section, we explain our best-of-N sampling method that can be used during test-time on trained and untrained LLMs. It is based on a novel solver-grounded reward function for LLMs that helps to generate correct ASP encodings more reliably.

Reward Function

The reward function f_r maps an encoding Γ and the number $M \in \mathbb{N}$ of answer sets produced by Γ to a floating point reward $r \in R$ that aims to judge the quality of Γ:

$$f_r(\Gamma, M) = \frac{1}{M} - \mathbb{1}_E(\Gamma) - \mathbb{1}_U(\Gamma) - \mathbb{1}_{NE}(\Gamma)$$

Usually, with every hint in a logic puzzle, the number of possible answers is either reduced or stays the same. Therefore, f_r rewards stricter generations. $\mathbb{1}$ are binary indicator variables checking Γ for undesired properties. All three contribute negatively to the reward:

- $\mathbb{1}_E$ indicates whether there are any errors or warnings when trying to solve Γ.
- $\mathbb{1}_U$ refers to unsatifiability, i.e., there is no warning or error, but also no answer set.
- $\mathbb{1}_{NE}$ indicates that a manually specified maximum number of answer sets is exceeded. This avoids out-of-memory issues in cases where broken encodings lead to combinatorial explosions.

Sampling Procedure

Our test-time method is based on *greedy* search, i.e., it aims at maximizing the current reward and does not perform trade-offs in favor of future rewards (cf. Sutton, Barto et al. (1998)). Similar to creating ASP encodings for training, we first instruct $\mathcal{M}_{ASP}$ to generate N alternatives for the partial encoding $\Gamma_{\mathcal{C},\mathcal{E}}$ that encodes entities and the choice rule. We then select and keep the alternative receiving the highest reward according to f_r. We use a special version of f_r for evaluating the $\Gamma_{\mathcal{C},\mathcal{E}}$ encodings, by replacing the reciprocal factor $\frac{1}{M}$ with a strict check whether the output contains $(n!)^{m-1}$ answer sets for an $m \times n$ grid puzzle which is different from $\mathbb{1}_{NE}$ in that it is dynamically determined based on the size of the input problem. This corresponds to the number of all theoretically possible answer combinations when disregarding the constraints. Next, for every hint $h_j \in \mathcal{H}$ in sequential order, we generate N alternatives and append them to the partial encoding that contains all previously selected encodings and judge all N partial encodings again by f_r. At each step, we select the partial encoding with the highest score and continue until we arrive at the full encoding Π. We make use of two recovery mechanisms when all new generations have negative rewards: (1) *Re-generation:* We let $\mathcal{M}_{ASP}$ generate $2 \times N$ additional alternatives if all initial N alternatives for the current input were judged negative by f_r. (2) *Backtracking:* We jump back to the previous hint with maximum reward that had more than one alternative and continue with this as our new partial encoding. We then restart to generate all successive hints.

Experimental Setup

Evaluation Metrics and Datasets

We report the accuracy of the models based on how often their output exactly matches the correct answer. ASP encodings that produce more than one solution in the end are considered wrong in our strict evaluation setup as the considered problems only have unique solutions. One issue when converting the problem into ASP is the ambiguity of string representation in the answer sets and the evaluation with accuracy when comparing to the ground truth solution (e.g., spaces, underscores, and other ambiguities). To automatically evaluate the predicted answer sets, we implement a *Levenshtein heuristic* that acts as fallback for fuzzy matching if the answer set does not exactly match the ground truth representation.

We work with **LogicPuzzles** (Mitra and Baral 2015) and **GridPuzzles** (Tyagi et al. 2024). LogicPuzzles is a collection of 124 grid-based puzzles (50 train and 74 test).[2] All puzzles are of size 3×4, i.e., there are 3 entity types, each with 4 instances (e.g., 4 dogs, 4 owners, 4 countries) where each entity must be assigned once. This results in 4 triples representing a unique solution (e.g., each dog is assigned to a different owner from a different country). GridPuzzles introduces different puzzle sizes (3×4, 3×5, 4×4, 4×5, 4×6) and difficulty levels (easy, medium, hard). We use this dataset only for evaluation.

[2]We noticed that in the official dataset release, 26 puzzles from the train set are also part of the test set. In order to evaluate the trained models on unseen data only, we remove these 26 instances from our test set. For comparability with prior work, we also add the evaluation on the full data in the appendix of the arXiv version.

Models

We train and evaluate four open-weight models from two model families: Llama-3.3 70B and Llama-3.1 8B[3] (Grattafiori et al. 2024) as well as Qwen3 32B and Qwen3 8B (Qwen-Team 2025). We disable the thinking mode introduced in the Qwen3 model family. For the two bigger models, we sample distinct training data using our proposed method and fine-tune them separately on their own part. To train the 8B models, we use the SFT data drawn from Llama 3.3 70B as it yields a more diverse range of statements than Qwen3 32B. Additionally, since these smaller models initially possess almost no ASP coding skills, we supplement the training data by adding the same number of instances from the LLASP dataset (Coppolillo et al. 2024) on top of our data. The LLASP dataset is a collection of single-statement building blocks (e.g., transitive closures).

We further compare our ASP models to the distilled variants of DeepSeek-R1 (DeepSeek-AI 2025). They are prompted to perform reasoning without making use of ASP. We use the prompt setup of Tyagi et al. (2024) to perform reasoning without an ASP solver. While we mainly focus on ASP coding skills, we also investigate how our neuro-symbolic method performs compared to reasoning language models (RLMs) on these logic reasoning tasks. Finally, we test our new inference method on one of the latest closed-source models, GPT-4.1-mini (version *2025-04-14*) and show that our best-of-N sampling method also improves closed-source models without the need for prior training.

We train LoRA adapters (Hu et al. 2022) on top of the base LLMs. We use 2 to 4 Nvidia H200 GPUs using DeepSpeed stage 3 (Rasley et al. 2020). The batch size is set to 4 per GPU used during training and the learning rate is set to $5\mathrm{e}{-5}$ for all models, which is a commonly used value for SFT. The parameters are kept in bfloat16 format. LoRA rank r and α are both set to 128. All models are trained for at most 10 epochs to keep computation feasible.

For inference, we use both Nvidia and Intel Gaudi 2 acceletor cards in combination with vllm (Kwon et al. 2023). We use clingo v.5.7.1 (Gebser et al. 2017) as ASP solver. We set $N = 5$ with a temperature of $T = 1.0$ for the test-time experiments to get a higher variety of outputs. Due to the higher variety of outputs, we average test-time runs over 5 distinct runs in the case of LogicPuzzles. To ensure computational feasibility, we limit the number of backtracking steps to 5 for LogicPuzzles and to 1 for GridPuzzles.

Training Data Statistics

We use the train split of LogicPuzzles to generate ASP training data. We request the LLMs to generate $N = 5$ alternatives per input. In the case of Llama-3.3 70B, we get 3.7 chosen responses and 1.3 rejected responses on average with a standard deviation of 1.82. Qwen3 32B yields 3.3 chosen and 1.7 rejected responses on average with a standard deviation of 2.04. This indicates that during sampling, Llama produces a slightly higher number of correct instances, whereas Qwen is not yet at the same level of ASP coding skills. Finally, we end up with 1.546 training instances from Llama and 1.060 from Qwen. Sampling ASP encodings from the 8B models did not yield sufficient results as they initially lack ASP knowledge and hence do not generate useful responses.

[3]At the time of publication, there is no Llama-3.3 8B model.

Settings

To test the impact of injecting ASP coding knowledge into LLMs, we compare two settings: First, we use a **single-prompt two-shot (2S)** prompt with limited engineering effort to generate the ASP program hint-wise as during training data generation. For this, we provide two dataset-specific examples and explain choice rules and different forms of logical OR constructs in ASP. We randomly select parts of two instances for GridPuzzles as there is no training split available. Second, we plug our models into an existing **PromptPipeline (PP)** (Ishay, Yang, and Lee 2023) consisting of 6 prompts specifically tailored to LogicPuzzles. This resembles a major prompt engineering effort for tailored ASP program generation to a particular dataset. Whereas our two-shot setting directly requests the LLM to generate ASP encodings for each natural language input, the pipeline uses the following six steps: (1) structured generation of constants, (2) formatting constants so that they are suited for ASP syntax, (3) generation of predicates that are used to form the relations between the entities, (4) formulation of the choice rule, (5) rewriting natural language constraints if they contain for example complex logical OR constructs, and (6) translating all natural language constraints into ASP at once. As only the sixth and last step produces one complete ASP encoding, a fine-grained intermediate analysis using the solver is not possible.

Results and Analysis

Two-Shot Parsing

The two upper rows of Table 1 display our main results on LogicPuzzles and GridPuzzles in the single-prompt parsing setup. All our SFT-trained models outperform their untrained counterparts on the task of ASP generation, emphasizing the necessity of fine-tuning current LLMs on underrepresented programming languages like ASP. Especially the 70B Llama heavily benefits from the training with two-digit improvements in all settings. For both 8B models, which do not possess usable ASP skills initially, we observe a notable increase in performance of approximately 20pp.-25pp. (percentage points) on LogicPuzzles and 10pp. on average on GridPuzzles after SFT-based training on data automatically drawn from Llama-3.3 70B. This indicates that our pipeline generates data with qualitative training signals such that small LLMs can learn the essence of ASP coding as well. Similar performance increases can also be observed for the much harder GridPuzzles dataset on which the models were not specifically trained, i.e., our method demonstrates strong transferability to more complex problem settings.

Effect of Test-time Search. Next, we apply our reward-based search function to all SFT-trained models with $N = 5$. We observe great improvements for all models over greedy sampling with $T = 0.0$ (and $N = 1$). Moreover, our test-time method leads to competitive performance compared to

Setup		Llama-3.3 70B				Llama-3.1 8B			Qwen3 32B			Qwen3 8B			DeepSeek Variants		
		w/o ASP	Base	SFT	+TT	Base	SFT	+TT	Base	SFT	+TT	Base	SFT	+TT	Llama 70B	Qwen3 32B	Llama 8B
2S	LogicP.	27.0	31.1	55.4	**66.8**	0.0	21.6	46.8	17.6	23.0	50.3	0.0	24.3	43.2	59.5	62.1	24.3
	GridP.	9.1	8.0	16.8	23.7	0.0	8.8	16.8	18.2	21.9	**33.9**	0.0	12.4	21.2	21.9	18.2	4.4
PP	LogicP.	-	56.8	**78.4**	-	0.0	0.0	-	51.4	67.6	-	31.1	24.3	-	-	-	-
	GridP.	-	33.9	**55.5**	-	0.0	0.0	-	27.7	37.2	-	9.1	16.1	-	-	-	-

Table 1: Accuracy of all models on both datasets in both the single-prompt parsing (2S) and prompt pipeline (PP) setting. All test-time methods (TT) are averaged across five runs. Underlined scores denote the best score within the category, bold scores the overall best results. Every row corresponds to one combination of prompt setting and dataset.

Model Variant	SFT	Seq	Reg	Back	N	LogicP.	Δ
Llama-3.3 70B					1	31.0	–
+SFT	✓				1	55.4	0.0
+Seq	✓	✓			5	62.7	+7.3
+Reg	✓	✓	✓		5	64.6	+9.2
+Back	✓	✓		✓	5	62.7	+7.3
+Both (TT)	✓	✓	✓	✓	5	66.8	+11.4
	✓	✓	✓	✓	10	65.1	+9.7
	✓	✓	✓	✓	25	**69.2**	+13.8
GPT-4.1-mini					1	39.2	0.0
+Seq		✓			5	55.4	+16.2

Table 2: Ablation study results on LogicPuzzles. Δ shows improvement over +SFT (Llama) or the base model (GPT). (**Reg** = regeneration, **Back** = backtracking, **Seq** = best-of-N w/o regeneration and backtracking)

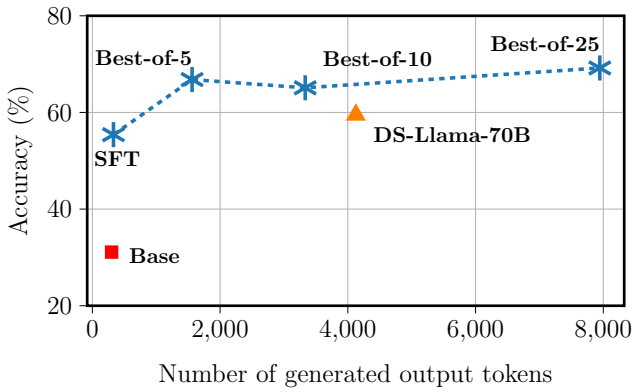

Figure 2: Comparison of generated output tokens in relation to accuracy on LogicPuzzles between the DeepSeek Llama-70B, the untrained Llama-3.3 70B as well as the SFT-tuned one in combination with reward-based inference.

the state-of-the-art RLMs: ASP-tuned Llama models show strong performance when compared to their Deepseek counterparts. We also observe similar tendencies for GridPuzzles.

Sampling multiple alternatives with a higher temperature and judging them independently using our reward function f_r greatly increases the robustness and reduces the number of semantically wrong (captured by the factor $\frac{1}{M}$) and erroneous partial ASP encodings as seen by the increased accuracy. This improved performance indicates that there is still insecurity within the model when it comes to ASP coding and that sampling multiple alternatives mitigates this issue.

A detailed study on the different components used in our test-time methods is shown in Table 2. Re-generation shows slight improvements of 2pp. over basic best-of-5 sampling. However, the combination of re-generation and backtracking demonstrates a robust improvement by over 3pp. This underlines the necessity of both fallback mechanisms: Sometimes regenerating $2 \times N$ alternatives is already sufficient to recover from errors, whereas backtracking allows for deeper changes. Table 2 also shows how the number of generations N influences our best-of-N sampling method. We observe bigger improvements, achieving up to 69% accuracy for $N = 25$, which shows that there is a valuable trade-off between inference runtime and accuracy on the problems. Finally, for the closed-source GPT-4.1-mini without any specific fine-tuning for ASP, we still observe a performance improvement of 16.2pp. when used with our reward-based search. This highlights the value of our solver-in-the-loop setup for a wider range of LLMs.

Cost Comparisons. Figure 2 further provides a calculation of the generated output tokens per instance on average in relation to the accuracy on LogicPuzzles for the RLM DeepSeek Llama 70B compared to the ASP-based setups.[4] First, we can see that our trained Llama 70B outperforms the untrained base version by 24.4pp. while keeping the same amount of produced output tokens. Furthermore, the number of generated output tokens scales linearly with N. We conclude that for increasing values of N, the number of produced output tokens is still in a feasible range. Thorough trade-off considerations for inference costs, however, must be taken into account for determining the best value for N. Finally, we can see that the number of produced output tokens of the RLM is in the range of our best-of-10 model while performing worse on LogicPuzzles. We can conclude that a value of $N = 5$ provides a good trade-off between cost efficiency and task performance in our setup.

[4]We use the number of generated output tokens as a general proxy for compute costs, as this greatly influences generation speed and costs - two of the main factors when deciding for LLMs. We do not count input tokens because this is highly problem-specific and depends mainly on the input prompt.

ASP-tuned Backbones for PromptPipeline

The bottom row of Table 1 shows our ASP models plugged into the 6-step PromptPipeline of Ishay, Yang, and Lee (2023) to test instruction-following capabilities. We find that fine-tuning the models also improves their instruction following capabilities w.r.t. producing proper ASP encodings. This shows that ASP-specific fine-tuning using few-shot prompts also generalizes to different settings that follow different approaches by instructing the model to first perform intermediate rewrite steps before finally producing the ASP encoding.

Discussion

One important strength of our method is that it requires no additional human annotations as it samples ASP statements from LLMs and automatically computes a reward or categorizes them into *chosen* and *rejected*. Yet, this also implies that even the chosen instances might not be perfect. We have shown that recent LLMs, despite possessing limited ASP coding skills, are able to produce initial ASP encodings that facilitate our successful self-supervised training data generation method. Furthermore, our experiments show that best-of-N sampling using a solver-in-the-loop setup mitigates stability issues of the ASP generation process with LLMs by filtering erroneous ASP encodings from a set of N alternatives during inference time.

Related Work

We review related work on neuro-symbolic systems with a focus on systems in which LLMs generate structured representations and solvers compute solutions (Kautz 2022), ASP code generation, data generation with feedback incorporation, and test-time methods.

Semantic parsing in neuro-symbolic systems. Inducing explicit representations of meaning from text relates to the task of *semantic parsing* (Hendrix et al. 1978; Delmonte 1990; Baud et al. 1998). Recently, LLMs have been used for semantic parsing into logic programming languages. LINC (Olausson et al. 2023) translates natural language premises and conclusions into symbolic representations for a theorem prover. Schrader et al. (2024) and Nafar, Venable, and Kordjamshidi (2024) extend this idea to probabilistic reasoning with numeric probabilities and uncertainty. Pan et al. (2023) create symbolic representations comparable to PROLOG and first-order logic and fine-tune LLMs based on error messages from the solver.

ASP code generation with LLMs. Early approaches combining NLP methods with ASP propose ideas for solving question answering and reasoning tasks (Baral, Gelfond, and Scherl 2004; Nouioua and Nicolas 2006). The LOGICIA system (Mitra and Baral 2015) combines a pairwise Markov network for entity extraction with a maximum entropy model for relation classification on the LogicPuzzles dataset. Coppolillo et al. (2024) introduce the LLASP dataset with single-line building blocks of ASP code. We focus on deriving ASP encodings for solving entire complex puzzles. Ishay, Yang, and Lee (2023) create a detailed prompting pipeline for GPT models for solving the LogicPuzzles dataset (Brown et al. 2020; OpenAI et al. 2024). Combining this pipeline with our ASP-tuned models outperforms prior work on GridPuzzles as well.

Generating data from feedback. Datasets can be labeled either manually or in an automated way (Xiao et al. 2024). The HelpSteer dataset (Wang et al. 2023; Dong et al. 2023) is an example for human annotations, while Li et al. (2023) employ GPT-4V to automatically judge the outputs of vision-language models. Lai et al. (2024) use GPT-4 to automatically identify faulty steps in mathematical reasoning chains. We are not aware of any prior work generating ASP training data from a solver-in-the-loop setup.

Test-time methods. Recent test-time methods fall into three categories (Dong, Teleki, and Caverlee 2024): *Independent self-improvement* refers to intervening in the generation process of frozen-parameter LLMs (Lu et al. 2022; Ning et al. 2024). *Context-aware self-improvement* describes adaptations to prompts by enriching the input to the model, e.g., via chain-of-thought prompting (Wei et al. 2022). Methods using feedback from additional expert (Zeng et al. 2024) or reward models (Deng and Raffel 2023) are called *model-aided self-improvement methods*. Our approach belongs to the third category.

Conclusion and Outlook

In this paper, we have presented a novel method for increasing the performance of LLMs on the task of ASP code generation. We have shown that an external ASP solver can be used to generate high-quality training data for instruction tuning in a fully automated fashion as well as to provide guidance during inference to filter out faulty ASP encodings. We observe consistent improvements for multiple models after training in two distinct prompting settings, demonstrating that our training method integrating solver feedback scales to a wider range of LLMs on assignment problems of different sizes and difficulties. Furthermore, we showed that solver feedback incorporated during test-time can further improve the robustness of the ASP generation process as it is used to find the best combination of partial encodings by ranking them against each other and allows to recover from generation errors.

Outlook. Potential next steps involve investigating further training setups. The usage of reinforcement learning-based (RL) training of LLMs has become popular due to its promising training results (Ouyang et al. 2022), including RL with our solver feedback as a training signal (Jha et al. 2024). We propose to extend our reward function f_r by introducing weights that rank each error type by its importance. It can be further extended to also include signals during training that indicate the correctness based on the ground truth solution (similar to our sampling procedure).

Acknowledgements

This work was partially supported by the EU Project SMARTY (GA 101140087).

References

Baral, C.; Gelfond, M.; and Scherl, R. 2004. Using answer set programming to answer complex queries. In *Proceedings of the Workshop on Pragmatics of Question Answering at HLT-NAACL 2004*, 17–22. Boston, Massachusetts, USA: Association for Computational Linguistics.

Baud, R. H.; Lovis, C.; Rassinoux, A.-M.; and Scherrer, J.-R. 1998. Morpho-semantic parsing of medical expressions. In *Proceedings of the AMIA Symposium*, 760. American Medical Informatics Association.

Brown, T. B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; Agarwal, S.; Herbert-Voss, A.; Krueger, G.; Henighan, T.; Child, R.; Ramesh, A.; Ziegler, D. M.; Wu, J.; Winter, C.; Hesse, C.; Chen, M.; Sigler, E.; Litwin, M.; Gray, S.; Chess, B.; Clark, J.; Berner, C.; McCandlish, S.; Radford, A.; Sutskever, I.; and Amodei, D. 2020. Language Models are Few-Shot Learners. arXiv:2005.14165.

Coppolillo, E.; Calimeri, F.; Manco, G.; Perri, S.; and Ricca, F. 2024. LLASP: Fine-tuning Large Language Models for Answer Set Programming. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, volume 21, 834–844.

DeepSeek-AI. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948.

Delmonte, R. 1990. Semantic parsing with LFG and conceptual representations. *Computers and the Humanities*, 24: 461–488.

Deng, H.; and Raffel, C. 2023. Reward-Augmented Decoding: Efficient Controlled Text Generation With a Unidirectional Reward Model. In Bouamor, H.; Pino, J.; and Bali, K., eds., *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 11781–11791. Singapore: Association for Computational Linguistics.

Dong, X.; Teleki, M.; and Caverlee, J. 2024. A Survey on LLM Inference-Time Self-Improvement. *arXiv preprint arXiv:2412.14352*.

Dong, Y.; Wang, Z.; Sreedhar, M. N.; Wu, X.; and Kuchaiev, O. 2023. SteerLM: Attribute Conditioned SFT as an (User-Steerable) Alternative to RLHF. arXiv:2310.05344.

Falkner, A.; Friedrich, G.; Schekotihin, K.; Taupe, R.; and Teppan, E. C. 2018. Industrial applications of answer set programming. *KI-Künstliche Intelligenz*, 32(2): 165–176.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2017. Multi-shot ASP solving with clingo. *CoRR*, abs/1705.09811.

Gelfond, M.; and Lifschitz, V. 1988. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, 1070–1080. Cambridge, MA.

Grattafiori, A.; et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783.

Gu, Q. 2023. Llm-based code generation method for golang compiler testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2201–2203.

Hendrix, G. G.; Sacerdoti, E. D.; Sagalowicz, D.; and Slocum, J. 1978. Developing a natural language interface to complex data. *ACM Transactions on Database Systems (TODS)*, 3(2): 105–147.

Hu, E. J.; yelong shen; Wallis, P.; Allen-Zhu, Z.; Li, Y.; Wang, S.; Wang, L.; and Chen, W. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *International Conference on Learning Representations*.

Ishay, A.; Yang, Z.; and Lee, J. 2023. Leveraging Large Language Models to Generate Answer Set Programs. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, volume 19, 374–383.

Jha, P.; Jana, P.; Suresh, P.; Arora, A.; and Ganesh, V. 2024. RLSF: Reinforcement Learning via Symbolic Feedback. *arXiv preprint arXiv:2405.16661*.

Jiang, J.; Wang, F.; Shen, J.; Kim, S.; and Kim, S. 2024. A Survey on Large Language Models for Code Generation. arXiv:2406.00515.

Kautz, H. 2022. The third ai summer: Aaai robert s. engelmore memorial lecture. *Ai magazine*, 43(1): 105–125.

Kwon, W.; Li, Z.; Zhuang, S.; Sheng, Y.; Zheng, L.; Yu, C. H.; Gonzalez, J. E.; Zhang, H.; and Stoica, I. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Lai, X.; Tian, Z.; Chen, Y.; Yang, S.; Peng, X.; and Jia, J. 2024. Step-DPO: Step-wise Preference Optimization for Long-chain Reasoning of LLMs. arXiv:2406.18629.

Li, L.; Xie, Z.; Li, M.; Chen, S.; Wang, P.; Chen, L.; Yang, Y.; Wang, B.; and Kong, L. 2023. Silkie: Preference Distillation for Large Visual Language Models.

Lifschitz, V. 2008. What is answer set programming? In *Proceedings of the 23rd national conference on Artificial intelligence-Volume 3*, 1594–1597.

Lu, X.; Welleck, S.; West, P.; Jiang, L.; Kasai, J.; Khashabi, D.; Le Bras, R.; Qin, L.; Yu, Y.; Zellers, R.; Smith, N. A.; and Choi, Y. 2022. NeuroLogic A*esque Decoding: Constrained Text Generation with Lookahead Heuristics. In Carpuat, M.; de Marneffe, M.-C.; and Meza Ruiz, I. V., eds., *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 780–799. Seattle, United States: Association for Computational Linguistics.

Marek, V. W.; and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The logic programming paradigm: A 25-year perspective*, 375–398. Springer.

McDermott, D.; Ghallab, M.; Howe, A. E.; Knoblock, C. A.; Ram, A.; Veloso, M. M.; Weld, D. S.; and Wilkins, D. E. 1998. PDDL-the planning domain definition language. In *Proceedings of the Artificial Intelligence Planning Systems Conference (AIPS-98)*.

Mitra, A.; and Baral, C. 2015. Learning to Automatically Solve Logic Grid Puzzles. In Màrquez, L.; Callison-Burch, C.; and Su, J., eds., *Proceedings of the 2015 Conference on*

Empirical Methods in Natural Language Processing, 1023–1033. Lisbon, Portugal: Association for Computational Linguistics.

Nafar, A.; Venable, K. B.; and Kordjamshidi, P. 2024. Probabilistic Reasoning in Generative Large Language Models. arXiv:2402.09614.

Ning, X.; Lin, Z.; Zhou, Z.; Wang, Z.; Yang, H.; and Wang, Y. 2024. Skeleton-of-thought: Prompting LLMs for efficient parallel generation. In *Proceedings 12th international conference on learning representations-ICLR 2024*.

Nouioua, F.; and Nicolas, P. 2006. Using Answer Set Programming in an inference-based approach to Natural Language Semantics. In Bos, J.; and Koller, A., eds., *Proceedings of the Fifth International Workshop on Inference in Computational Semantics (ICoS-5)*.

Olausson, T.; Gu, A.; Lipkin, B.; Zhang, C.; Solar-Lezama, A.; Tenenbaum, J.; and Levy, R. 2023. LINC: A Neurosymbolic Approach for Logical Reasoning by Combining Language Models with First-Order Logic Provers. In Bouamor, H.; Pino, J.; and Bali, K., eds., *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 5153–5176. Singapore: Association for Computational Linguistics.

OpenAI; et al. 2024. GPT-4 Technical Report. arXiv:2303.08774.

Ouyang, L.; Wu, J.; Jiang, X.; Almeida, D.; Wainwright, C. L.; Mishkin, P.; Zhang, C.; Agarwal, S.; Slama, K.; Ray, A.; Schulman, J.; Hilton, J.; Kelton, F.; Miller, L.; Simens, M.; Askell, A.; Welinder, P.; Christiano, P.; Leike, J.; and Lowe, R. 2022. Training language models to follow instructions with human feedback. arXiv:2203.02155.

Pan, L.; Albalak, A.; Wang, X.; and Wang, W. 2023. Logic-LM: Empowering Large Language Models with Symbolic Solvers for Faithful Logical Reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, 3806–3824.

Peng, Y.; Gotmare, A. D.; Lyu, M.; Xiong, C.; Savarese, S.; and Sahoo, D. 2024. PerfCodeGen: Improving Performance of LLM Generated Code with Execution Feedback. arXiv:2412.03578.

Qwen-Team. 2025. Qwen3 Technical Report. arXiv:2505.09388.

Rasley, J.; Rajbhandari, S.; Ruwase, O.; and He, Y. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, 3505–3506. New York, NY, USA: Association for Computing Machinery. ISBN 9781450379984.

Schrader, T. P.; Lange, L.; Razniewski, S.; and Friedrich, A. 2024. QUITE: Quantifying Uncertainty in Natural Language Text in Bayesian Reasoning Scenarios. In Al-Onaizan, Y.; Bansal, M.; and Chen, Y.-N., eds., *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2634–2652. Miami, Florida, USA: Association for Computational Linguistics.

Stein, K.; Fišer, D.; Hoffmann, J.; and Koller, A. 2023. Autoplanbench: Automatically generating benchmarks for llm planners from pddl. *arXiv preprint arXiv:2311.09830*.

Sutton, R. S.; Barto, A. G.; et al. 1998. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.

Tyagi, N.; Parmar, M.; Kulkarni, M.; Rrv, A.; Patel, N.; Nakamura, M.; Mitra, A.; and Baral, C. 2024. Step-by-Step Reasoning to Solve Grid Puzzles: Where do LLMs Falter? In Al-Onaizan, Y.; Bansal, M.; and Chen, Y.-N., eds., *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 19898–19915. Miami, Florida, USA: Association for Computational Linguistics.

Ugare, S.; Suresh, T.; Kang, H.; Misailovic, S.; and Singh, G. 2024. Improving llm code generation with grammar augmentation. *arXiv preprint arXiv:2403.01632*.

Vyas, K.; Graux, D.; Yang, Y.; Montella, S.; Diao, C.; Zhou, W.; Vougiouklis, P.; Lai, R.; Ren, Y.; Li, K.; et al. 2024. From an LLM Swarm to a PDDL-empowered Hive: Planning Self-executed Instructions in a Multi-modal Jungle. *arXiv preprint arXiv:2412.12839*.

Wang, Z.; Dong, Y.; Zeng, J.; Adams, V.; Sreedhar, M. N.; Egert, D.; Delalleau, O.; Scowcroft, J. P.; Kant, N.; Swope, A.; and Kuchaiev, O. 2023. HelpSteer: Multi-attribute Helpfulness Dataset for SteerLM. arXiv:2311.09528.

Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; ichter, b.; Xia, F.; Chi, E.; Le, Q. V.; and Zhou, D. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In Koyejo, S.; Mohamed, S.; Agarwal, A.; Belgrave, D.; Cho, K.; and Oh, A., eds., *Advances in Neural Information Processing Systems*, volume 35, 24824–24837. Curran Associates, Inc.

Xiao, W.; Wang, Z.; Gan, L.; Zhao, S.; He, W.; Tuan, L. A.; Chen, L.; Jiang, H.; Zhao, Z.; and Wu, F. 2024. A Comprehensive Survey of Direct Preference Optimization: Datasets, Theories, Variants, and Applications. arXiv:2410.15595.

Yang, X.; Chen, B.; and Tam, Y.-C. 2024. Arithmetic reasoning with llm: Prolog generation & permutation. *arXiv preprint arXiv:2405.17893*.

Zeng, J.; Meng, F.; Yin, Y.; and Zhou, J. 2024. Improving Machine Translation with Large Language Models: A Preliminary Study with Cooperative Decoding. In Ku, L.-W.; Martins, A.; and Srikumar, V., eds., *Findings of the Association for Computational Linguistics: ACL 2024*, 13275–13288. Bangkok, Thailand: Association for Computational Linguistics.