Solver-Guided Optimization of Large Language Models for Logic Puzzle Reasoning with Answer Set Programming

Anonymous ACL submission

Abstract

The rise of large language models (LLMs) has sparked interest in neuro-symbolic systems that leverage logic reasoners to overcome LLM shortcomings. Answer set programming (ASP) is a particularly effective approach to finding solutions to combinatorial search problems, which LLMs often fail to solve. However, the effectiveness of LLMs in ASP code generation is hindered by the limited number of examples seen during their initial pre-training phase.

In this paper, we introduce a novel approach for solver-guided instruction-tuning of LLMs for addressing the highly complex semantic parsing task inherent in ASP code generation. We sample ASP statements for program continuations proposed by LLMs for unriddling logic puzzles and 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 best-of-N test-time sampling. Our experiments demonstrate consistent improvements across four datasets.

1 Introduction

Current large language models (LLMs) often fail at solving problems that require extensive reasoning capabilities (Huang and Chang, 2023; Yang et al., 2023; Tyagi et al., 2024; Ahn et al., 2024). An increasingly popular countermeasure is test-time compute, e.g., by generating longer reasoning chains or sampling multiple outputs for the same input as done in reasoning language models (RLMs, Muennighoff et al., 2025; DeepSeek-AI, 2025; Cobbe et al., 2021). However, these RLMs are slower during inference due to long reasoning chains and suffer from increased hallucinations (Hashemi et al., 2025).

Alternatively, combining LLMs with symbolic reasoning engines has demonstrated promising results for problems such as logical reasoning (Olausson et al., 2023), mathematical reasoning (Gao et al., 2023) and Bayesian reasoning (Schrader et al., 2024). The neuro-symbolic approaches fuse LLMs with logic programming (Robinson, 1983; Gochet et al., 1988), which represents domain knowledge and problem specifications using formal logic and applies a solver to obtain solutions.

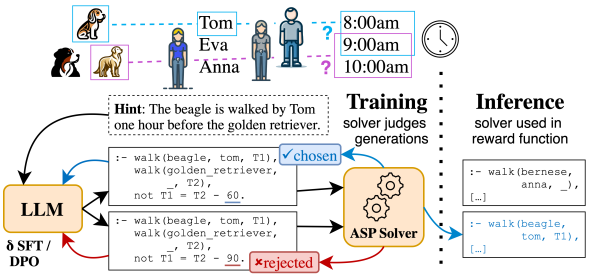

Figure 1: We use the feedback from an ASP solver to assess LLM-generated ASP statements. From that, we derive preference pairs that we can use both for SFT and DPO fine-tuning. Furthermore, we use the feedback to filter out erroneous generations during inference.

Many real-life problems, such as scheduling or assignment problems as shown in Figure 1, are solvable using answer set programming (ASP, Gelfond and Lifschitz, 1988; Marek and Truszczyński, 1999). Translating problems specified in natural language into ASP requires extensive training even for humans. With the rise of LLMs, research on automatic code generation has recently gained momentum (Jiang et al., 2024; Gu, 2023; Ugare et al., 2024). However, generating ASP code with LLMs remains understudied and suffers from limited presence of ASP code in pre-training data. First explorations find mixed results for question answering tasks (Yang et al., 2023), provide ASP statements only in isolation without a problem context (Coppolillo et al., 2024), or over-engineer a prompt pipeline for a particular dataset (Ishay et al., 2023).

In this paper, we combine neuro-symbolic methods with test-time compute. First, we introduce a novel solver-guided method for generating ASP preference pairs for instruction tuning of LLMs on logic puzzles (Mitra and Baral, 2015). The

LLM-generated ASP encodings are automatically evaluated by the solver and classified as chosen and rejected. Then, we build preference pairs to train open-weight LLMs from two families using direct preference optimization (DPO, Rafailov et al., 2024) and supervised fine-tuning (SFT) via causal language modeling.

We improve our parsing models during inference by leveraging best-of-N sampling with a novel solver-based reward function that is able to select the best ASP encodings from a set of alternatives.

Our experiments demonstrate the effectiveness of our novel training and inference methods: We observe large and consistent improvements on three logic puzzle datasets and one math dataset. Instruction-tuning increases accuracy by up to almost 30pp. when using a single prompt to generate the code. Our solver-based test-time compute boosts the performance further by up to 15pp. compared to the already trained models.

To summarize, our contributions are as follows:
(1) We present a fully automated method for generating training data for self-supervised ASP preference-tuning that requires no manual annotation. Using this method, we generate thousands of pairs of chosen and rejected ASP instances.
(2) We show that self-supervised preference alignment on these ASP preference pairs greatly improves the performance of open-weight LLMs generating ASP encodings for grid-based puzzles, in particular, for problems of higher complexity.
(3) We demonstrate that feedback from an ASP solver can be incorporated during inference to effectively filter unreliable and erroneous ASP encodings generated by the LLM.

We publicly release code, preference pairs from three distinct LLMs, and all results on Github.[1]

2 Related Work

We review related work on reasoning tasks, neuro-symbolic systems, ASP code generation, preference data generation and test-time methods. Our use of the term *neuro-symbolic* refers to systems in which language models generate structured representations and solvers compute solutions (Kautz, 2022).

Reasoning tasks. Reasoning is a complex process with various facets (Qiao et al., 2023). Reasoning capabilities of natural language processing (NLP) models have commonly been tested using sentence-pair entailment tasks (Bos and Markert, 2005; Bowman et al., 2015). More recently, Han et al. (2024) predict whether logic statements entail or contradict a query statement. Most relevant for this work are grid-based logic puzzles like LogicPuzzles (Mitra and Baral, 2015), GridPuzzles (Tyagi et al., 2024), and ZebraLogic (Lin et al., 2025). Other datasets cover clue-based question answering (Zhong et al., 2021) or solving algebraic equations (He-Yueya et al., 2023).

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 et al. (2024) extend this idea to probabilistic reasoning with numeric probabilities and uncertainty. Pan et al. (2023) create symbolic representations and fine-tune LLMs based on error messages from the solver.

ASP code generation with LLMs. Early appraoches combining NLP methods with ASP propose ideas for solving question answering and reasoning tasks (Baral et al., 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 preference pairs for solving entire complex puzzles. Ishay et al. (2023) create a detailed prompting pipeline for GPT models for solving the LogicPuzzles dataset (Brown et al., 2020; OpenAI et al., 2024). Combining it with our ASP-tuned models outperforms prior work on GridPuzzles as well.

Preference data from feedback. Labeling preference datasets can be done manually or automated (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) utilize GPT-4 to automatically identify faulty steps in mathematical reasoning chains. We are not aware of any prior work generating ASP preference data.

[1] `will-be-added-after-review` For review purposes, we added samples in OpenReview.

Test-time methods. Recent test-time methods fall into three categories (Dong et al., 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 adaptions to prompts by enriching the input to the model, e.g., 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.

3 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). We now give a short introduction into ASP following Lifschitz (2008).[2] 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 now 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*), e.g., "If the beagle is walked by Tom, then the golden retriever will be walked by Eva at 9am." 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. Assuming that the ASP program already contains atoms specifying a set of entities (here dogs, persons, and times), the following choice rule states "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, the solution of an ASP program Π is a set of possible assignments, i.e., stable models, and the effect of adding a constraint to a program can lead to their elimination, i.e., constraints reduce the set of solutions monotonically.

4 Instruction Tuning for ASP

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

4.1 Task Definition

The problems addressed in this paper are 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 (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 $\mathcal{S}$ 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 true subset of the ASP code in Π, i.e., $solution(\Pi = (\Gamma_1 \dots \Gamma_n)) = \mathcal{S}$.

4.2 Sampling Trajectories

We define a *trajectory* $\mathcal{T}$ to be an alternating sequence of language inputs and ASP statements. A trajectory $\mathcal{T}$ starts with a prompt $P_{\mathcal{D},\mathcal{E}}$ com-

[2] We provide a step-by-step ASP example explaining all relevant concepts for a grid-based puzzle instance in Appendix A.

prising general information on ASP,[3] the textual problem description $\mathcal{D}$ and the set of entities $\mathcal{E}$ (provided within as text). 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 $\mathcal{H}_i \in \mathcal{H}$ is appended to this trajectory and $\mathcal{M}_S$ is prompted again. This results in trajectories of form $\mathcal{T} = \{\mathcal{P}_{\mathcal{D},\mathcal{E}}, \Gamma_{\mathcal{C},\mathcal{E}}, \mathcal{H}_1, \Gamma_1, \mathcal{H}_2, \Gamma_2, \ldots, \mathcal{H}_n, \Gamma_n\}$ with n being the number of hints of the problem.

4.3 Classification of ASP Encodings

After processing k of the n hints, we obtain a trajectory $\mathcal{T}_k$ which contains k+1 ASP encodings (one for entities + choice rule and k for hints). We now combine all k+1 ASP encodings into the partial encoding $\Gamma_{\mathcal{C},\mathcal{E}}\Gamma_1, ..\Gamma_k$ and use the solver to compute its solution. Since DPO relies on preference pairs, i.e., a chosen and a rejected response to an input prompt, we sample 5 completions from $\mathcal{M}_{\text{ASP}}$ for each step k.[4] We then use an ASP solver to evaluate if the partial encoding Π_k generates a solution that comprises the ground truth answer set $\mathcal{S}$. Recall that when adding constraints, possible solutions are removed (see Section 3), i.e., if the partial program $\Gamma_{\mathcal{C},\mathcal{E}}\Gamma_1, ..\Gamma_k$ is correct, the ground truth answer should be one of the solutions that can be derived from it. We consider Γ_k as a *chosen* response to input $\mathcal{T}_k = \{\mathcal{P}_{\mathcal{D},\mathcal{E}}, \Gamma_{\mathcal{C},\mathcal{E}}, \mathcal{H}_1, \Gamma_1, \mathcal{H}_2, \Gamma_2, \ldots, \mathcal{H}_k\}$ if the solution of the partial program $\Gamma_{\mathcal{C},\mathcal{E}}\Gamma_1, ..\Gamma_k$ comprises the ground truth solution. If it produces only wrong answer sets, no answer sets (i.e., it is unsatisfiable), or errors and warnings are returned by the solver, we consider Γ_k as *rejected*.

4.4 Preference Pair 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. 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 preferences pairs. If all responses are rejected, the input to the next step becomes $\{\mathcal{P}_{\mathcal{D},\mathcal{E}}, \Gamma_{\mathcal{C},\mathcal{E}}, \mathcal{H}_1, \Gamma_1, \ldots, \mathcal{H}_{k-1}, \Gamma_{k-1}, \mathcal{H}_{k+1}\}$, i.e., the rejected response and the hint prompt causing it are removed. This is possible because the hints in LogicPuzzles do not refer to each other.

4.5 Model Preference Alignment

The preference pairs can be used to instruction-tune LLMs for the task of generating ASP code using either DPO or SFT. DPO requires preference pairs to be sample from the original model's distribution, hence, our algorithm for generating preference pairs must be performed for each LLM in our study separately. As an alternative to DPO, we use SFT by training using causal language modeling. To allow for a fair comparison, we train on the same number of instances as in the DPO setting by taking all preference pairs and only train on the *chosen* subpart. DPO and SFT can be combined by first performing SFT on the base LLM and then sampling new preference data from $\mathcal{M}_{\text{SFT}}$ (since the DPO data should always come from exactly the model to be trained). We then apply DPO using these additional preference pairs.

5 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.

5.1 Reward Function

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

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

M is the number of produced answer sets. Usually, every hint in logic puzzles reduces the number of possible answers or keep it as it is. 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.

[3] This includes an instruction of how XOR works in ASP, a high-level explanation of the steps required to solve a grid-based puzzle in ASP as well as a basic choice rule

[4] Preliminary experiments showed that a temperature $t = 0.8$ for sampling provides a good balance between chosen and rejected responses.

$\mathbb{1}_U$ 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.

5.2 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 et al. (1998)).

Similar to creating preference pairs (cf. Section 4), we first instruct $\mathcal{M}_{ASP}$ to generate N alternatives for the partial encoding $\Gamma_{\mathcal{C},\mathcal{E}}$ that encodes entities and 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 check whether the output contains $(n!)^{m-1}$ answer sets for an $m \times n$ grid puzzle. This corresponds to the number of all theoretically possibly answer combinations when disregarding the constraints (see Appendix H).

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.

Furthermore, we add two recovery mechanisms when all new generations have negative rewards:

Regeneration. 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.

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. To keep it computationally feasible, we limit the amount of backtracking steps to six for LogicPuzzles and two on the other datasets.

6 Experimental Setup

We now describe the datasets and models used in our experiments.

6.1 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 provide more than one solution in the end are considered wrong in our strict evaluation setup. [5]

We mainly work with **LogicPuzzles** (Mitra and Baral, 2015) and **GridPuzzles** (Tyagi et al., 2024). LogicPuzzles is a collection of 150 grid-based puzzles (50 train and 100 test). 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 ($3\times4,\ 3\times5,\ 4\times4,\ 4\times5,\ 4\times6$) and difficulty levels (easy, medium, hard).

To test the generalizability of our models, we also experiment with **ZebraLogic** (Lin et al., 2025; Dziri et al., 2024), which contains 1.000 puzzles with the task of matching different entities exclusively to their house number. Moreover, we conduct a small out-of-domain study on **GSM-Algebra** (He-Yueya et al., 2023). This datasets does not contain puzzles, but algebraic questions. More details are provided in appendix I.

6.2 Models

We evaluate four open-weight instruction-tuned LLMs: three general-purpose models, Llama-3.1 70B & 8B (Grattafiori et al., 2024) and Qwen-2.5 72B, and one model specifically tuned to programming, Qwen-2.5-Coder 32B (Qwen-Team, 2024; Yang et al., 2024). We train the two larger models in three settings: SFT, DPO and a combination of first SFT and then DPO. For Qwen-Coder, we focus on DPO. We train the Llama 8B model with combined SFT on the LLASP dataset (Coppolillo et al., 2024) and training data sampled from the 70B model for the LogicPuzzles dataset.

We test GPT-4o [6] to see if our test-time method can also improve standard closed API-based models. As further baselines, we study reasoning LMs (RLMs) with the distilled variants of DeepSeek-R1 (DeepSeek-AI, 2025). We use the prompt setup of Tyagi et al. (2024) to perform reasoning without an ASP solver.

[5] One issue when converting the problem into ASP is the ambiguity of string representation in the answer sets and the evaluation with accuracy 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 (see Appendix B)

[6] version *2024-08-06*

		LogicP.	GridP.
Baselines w/o ASP	*GPT-4-Turbo*	7.0*	5.1
	Llama-3.1 70B	9.0	3.6
Deepseek-R1 Distilled w/o ASP	*Llama-70B*	52.0	**21.9**
	Qwen-32B	54.0	18.2
	Llama-8B	20.0	4.4
Llama-3.1 70B	*Base*	16.0	2.6
	DPO	37.0	11.3
	SFT	39.0	12.0
	SFT+DPO	43.0	10.9
	+ TT (seq.)	55.6	21.2
Llama-3.1 8B	*Base*	0.0	0.0
	SFT+LLASP	17.0	6.9
	+ TT (seq.)	47.2	13.9
Qwen-2.5 72B	*Base*	14.0	3.3
	DPO	25.0	12.8
	SFT	30.0	6.9
	+ TT (seq.)	45.4	16.4
	SFT+DPO	16.0	7.7
Qwen-2.5 -Coder 32B	*Base*	6.0	2.9
	DPO	12.0	8.4
	+ TT (seq.)	25.6	13.1
GPT-4o	*Base*	49.0	-
	+ TT (seq.)	**61.0**	-

Table 1: Accuracy for **2-shot prompting with a single prompt** for LLM variants on grid-based puzzles. All test-time methods are averaged across five runs. **taken from Tyagi et al. (2024).*

We use 2 to 4 Nvidia H200 cards for training with model sharding. We use clingo (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.

More hyperparameters and GPU details are listed in Appendix C.

6.3 Preference Pair Statistics

We use the train split of LogicPuzzles to generate ASP preference pairs, i.e., the ASP training data used for SFT and DPO training of all open-weight models. This split contains 50 grid-based puzzles with unique solutions. As DPO assumes that the preference data is part of the original model's distribution, we run our algorithm on all three LLMs. Llama 70B produces most *chosen* responses, with an average of 3 out of 5 drawn responses, followed by the almost equally sized Qwen 72B with 1.9 and 1.3 for Qwen-Coder. Conversely, Qwen-Coder and Qwen 72B produce significantly more rejected instances (3.7 and 3.1, versus 2.0 for Llama 70B). This is also reflected in the number of pairs, as a more balanced number of *chosen* and *rejected* responses leads to the creation of more pairs by taking the Cartesian product between both. Sampling preference data from the Llama 8B model did not yield sufficient results as it initially lacks ASP knowledge and hence does not generate useful responses.

6.4 Settings

To test the impact of injecting ASP coding knowledge into LLMs, we compare two settings:

(1) We use a single **two-shot** prompt with limited engineering effort to generate the ASP program hint-wise as during preference pair generation. For this, we provide two dataset-specific examples[7] and explain choice rules and XOR in ASP.

(2) Alternatively, we plug our models into an existing **PromptPipeline** (*PP*, Ishay et al., 2023) consisting of 6 distinct prompts, specifically tailored to LogicPuzzles.[8] This resembles a major prompt engineering effort for tailored ASP program generation to a particular dataset.

7 Results and Analysis

7.1 Two-Shot Parsing

Table 1 displays our main results on LogicPuzzles and GridPuzzles in the single-prompt parsing setup. The base variants, i.e., the untrained models, are strongly outperformed by all our training settings (DPO, SFT and SFT+DPO) on the task of ASP generation, emphasizing the necessity of fine-tuning current LLMs on under-represented programming languages like ASP. Furthermore, we see that Qwen 72B shows the best results after SFT-based training, while Llama 70B further profits from DPO on top of SFT, indicating that different models respond differently to the different training methods on the same task.

The relative improvements of the various models are influenced by their ratios of chosen and rejected responses generated by the models for the self-training and to their overall number of training instances (cf. Section 6.3).

Similar performance improvements can also be observed for the harder GridPuzzles dataset on which the models were not trained. The training on LogicPuzzles seems to improve ASP generation

[7] We randomly select parts of two instances for GridPuzzles as there is no training split available.

[8] We slightly adapt the pipeline with instructions not to generate explanatory comments since non-GPT models tend to create more verbose output on these exact prompts.

	N	LogicPuzzles
Llama-3.1 70B	1	16.0
+SFT+DPO		43.0
+Seq	5	49.0
+Seq+Reg		51.6
+Seq+Back		53.2
+Seq+Both	5	55.6
	10	59.6
	25	63.8
	50	**65.4**
	100	64.4

Table 2: Study of different test-time extensions and values for N using Llama 70B on LogicPuzzles.

		ZebraLogic	GSM A.
Llama-3.1 70B	*w/o ASP**	24.9	64.7
	Base	19.0	60.3
	SFT+DPO	30.8	64.1
	+ TT (seq.)	**49.3**	**68.6**

Table 3: Accuracy comparison on ZebraLogic between base LLM, fine-tuned LLM with ASP and test-time enriched inference. **Results were taken from the public leaderboard on May 16th, 2025.*

abilities of LLMs on other datasets as well. We provide a more detailed analysis on GridPuzzles based on instance size and difficulty in Appendix J.

Notably, the Qwen-Coder model lags far behind its general-purpose 72B counterpart on this code generation task. While this might be attributed to the smaller model size, it could also be caused by inferior language understanding. The latter is particularly relevant to correctly parse the complex text hints of logic puzzles.

Effect of Test-time Inference. Next, we apply our test-time method (see Section 5) to the best-performing systems on LogicPuzzles.

We observe great improvements for all models over greedy sampling with $T = 0.0$. Moreover, our test-time method lead to competetive performance compared to the RLMs. We observe similar absolute improvements on GridPuzzles.

Sampling multiple alternatives with a higher temperature and judging them independently using our reward function f_r greatly reduces the number of wrong and erroneous partial ASP encodings as seen by the increased accuracy. This leads to a better overall performance compared to the standard decoding process with $N = 1$, i.e., only a single produced ASP statement per input.

A detailed study on the different components used in our test-time methods is shown in Table 2. Both error fallbacks (backtracking and regeneration) are shown to be effective and further improve the model by up to 4pp. The table also shows a study on how the number of generations N influences our best-of-N sampling method. We observe bigger improvements up to $N = 50$ until it plateaus. This implies that there is a trade-off between additional compute for higher values of N and the performance gain.

		LogicP.	GridP.
GPT-4-Turbo	*Base*	72.0	43.1
Llama-3.1 70B	*Base*	27.0	21.5
	DPO	78.0	**54.4**
	SFT	**79.0**	51.8
	SFT+DPO	77.0	54.0
Llama-3.1 8B	*Base*	0.0	0.0
	SFT+LLASP	0.0	0.0
Qwen-2.5 72B	*Base*	**79.0**	43.4
	DPO	64.0	46.7
	SFT	68.0	43.8
	SFT+DPO	72.0	45.3
Qwen-2.5-Coder 32B	*Base*	43.0	24.1
	DPO	52.0	31.0

Table 4: Performance comparison of LLM variants using the **PromptPipeline** from Ishay et al. (2023).

For larger values like $N = 100$, the likelihood of generating semantically incorrect alternatives that yield fewer answer sets increases. However, these would be chosen by f_r due to the reciprocal factor $\frac{1}{M}$, indicating that it is important to find the "sweet spot" of ASP code that restricts the answer space just enough.

Transfer to other Datasets. Table 3 shows the results of different methods based on Llama 70B on ZebraLogic and GSM-Algebra. On ZebraLogic, our neuro-symbolic model with best-of-N sampling gets twice as many answers correct compared to the base Llama without logic solvers, and improves even more on the base Llama with ASP. The effects on GSM-Algebra are similar. In summary, we observe a great positive effect by combining training methods and test-time inference.

7.2 ASP-tuned Backbones for PromptPipeline

In Table 4, we plug our ASP models into the 6-step PromptPipeline of (Ishay et al., 2023) to test instruction-following capabilities. We replicate the pipeline with GPT-4-Turbo as backbone and evaluate the results using the same strict evaluation metrics as for our models.[9]

Fine-tuning Llama 70B and Qwen-Coder in a

[9] Ishay et al. (2023) only check for answer set uniqueness.

few-shot setting also improves their ASP generations in an instruction-following setting. This shows that combined training and prompt engineering is able to achieve strong results. However, we argue that over-engineering prompt pipelines to particular datasets is not the right path for improving neuro-symbolic models for general problems.

We observe a slight drop for Qwen 72B after fine-tuning. We identify the root cause to be that the model starts to solely copy an example from the input prompts into the output. When systematically removing this same code block from the erroneous instances, we can recover an accuracy of 82% in the case of the SFT-based model, thereby outperforming the untrained model by 3pp.

The smaller Llama 8B is not able to get any answer correct when using the PP, likely due to missing instruction-following capabilities of both the base and trained 8B model.

7.3 Quantitative and Qualitative Analysis

After manually checking error instances of the models, we found one of the main issues to be complex combinations of XOR. So-called *exhaustive enumerations* (that is, *One is A, one is B, one is C, and one is D*) are often over-simplified by the parsing models, leading to wrong constraints and therefore to either no solution (if it is too restrictive) or too many solutions (if it is not restrictive enough). We provide further analysis by error categories in Appendix E.

We also find evidence for LLMs leveraging their common-sense knowledge when generating ASP encodings. For example, to enable arithmetic on months, both the base and trained models start to generate helper predicates such as `month_ordering(Month, Num)` to translate month strings to ordinal numbers. Similarly, we found an LLM output defining five clock times in a numeric ordering from 1 to 5 to perform arithmetic operations on clock times. However, sometimes the LLMs do not introduce this numeric conversion, which ultimately leads to errors when trying to perform arithmetic operations.

7.4 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 preference generation method. Our experiments also reveal very different base performance and reactions to DPO and SFT for ASP coding between Llama, Qwen 72B, and Qwen-Coder, which suggests interesting future explorations.

Our experiments clearly demonstrate that DPO/SFT finetuning injects relevant knowledge on ASP coding into LLMs. We observe notable improvements on all tasks when training on a single grid puzzles dataset. We believe that given more diverse training data, our approach can help to support natural language understanding for problem solving more broadly.

Furthermore, our experiments show that best-of-N sampling mitigates stability issues of the ASP generation process with LLMs by filtering erroneous ASP encodings from a set of N alternatives during inference time.

8 Conclusion and Outlook

In this paper, we have presented a 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 meaningful training data for instruction tuning in a fully automated fashion as well as to provide guidance during inference to filter out ASP encodings of bad quality. We observe consistent improvements on four datasets for several trained models in both two-shot settings and when using a highly engineered prompt pipeline. This shows that our training methods based on solver feedback scale well and consistently to a wider range of problem cases.

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 preference sampling). This could then be used for instance with GRPO-based training (Shao et al., 2024). We leave this for future research.

Limitations

We had to fix experimental settings throughout all stages in this paper, i.e., data generation prompts, dataset sizes, hyperparameters for model training, and evaluation. However, one could use a different instruction prompt to sample data, e.g., by providing more dataset-specific information. The amount of training data is also variable. Hence, future research could address this limitation by testing more settings.

Furthermore, evaluating the output of neuro-symbolic systems is an involved task which requires a solver to process massive amounts encodings sequentially. However, especially for long encodings, LLMs can produce broken encodings that lead to the solver computing a huge answer space. This could lead to out-of-memory errors and non-terminating behavior. Therefore, we emphasize that finding good timeout values for the solver is crucial for such systems.

Evaluating neuro-symbolic output and comparing it to the ground truth is not trivial as exact string comparison is often not possible to ambiguity in the naming of constants and variables. Therefore, we need to apply heuristics as a fallback to still find all correct answers. Though, these heuristics are not perfect, leaving room for further improvement to avoid instances being falsely classified as wrong. Our manual evaluations on the LogicPuzzles dataset revealed a false negative rate of ~2% for our evaluation method in this setup.

Finally, our test-time method requires a carefully crafted value for N. In the case of sequential parsing, an instance $\mathcal{I}$ with K hints and one choice rule requires generating at least $\mathcal{O}((K+1) \times N)$ partial encodings. As a result, our test-time method requires a thorough consideration of runtime and compute trade-off between even further improving performance and requiring more compute.

Ethical Statement

All datasets that we use to evaluate our models only contain fictional stories without any connection to real-world events. Therefore, no personal or sensitive data is involved in any step of our research.

References

Janice Ahn, Rishu Verma, Renze Lou, Di Liu, Rui Zhang, and Wenpeng Yin. 2024. Large language models for mathematical reasoning: Progresses and challenges. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop*, pages 225–237, St. Julian's, Malta. Association for Computational Linguistics.

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

Robert H Baud, Christian Lovis, Anne-Marie Rassinoux, and Jean-Raoul Scherrer. 1998. Morpho-semantic parsing of medical expressions. In *Proceedings of the AMIA Symposium*, page 760. American Medical Informatics Association.

Johan Bos and Katja Markert. 2005. Recognising textual entailment with logical inference. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, pages 628–635, Vancouver, British Columbia, Canada. Association for Computational Linguistics.

Samuel R. Bowman, Gabor Angeli, Christopher Potts, and Christopher D. Manning. 2015. A large annotated corpus for learning natural language inference. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 632–642, Lisbon, Portugal. Association for Computational Linguistics.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. *Preprint*, arXiv:2005.14165.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *Preprint*, arXiv:2110.14168.

Erica Coppolillo, Francesco Calimeri, Giuseppe Manco, Simona Perri, and Francesco Ricca. 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, pages 834–844.

DeepSeek-AI. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

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

Haikang Deng and Colin Raffel. 2023. Reward-augmented decoding: Efficient controlled text generation with a unidirectional reward model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 11781–11791, Singapore. Association for Computational Linguistics.

Xiangjue Dong, Maria Teleki, and James Caverlee. 2024. A survey on llm inference-time self-improvement. *arXiv preprint arXiv:2412.14352*.

Yi Dong, Zhilin Wang, Makesh Narsimhan Sreedhar, Xianchao Wu, and Oleksii Kuchaiev. 2023. Steerlm: Attribute conditioned sft as an (user-steerable) alternative to rlhf. *Preprint*, arXiv:2310.05344.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jian, Bill Yuchen Lin, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena D. Hwang, Soumya Sanyal, Sean Welleck, Xiang Ren, Allyson Ettinger, Zaïd Harchaoui, and Yejin Choi. 2024. Faith and fate: Limits of transformers on compositionality. *Advances in Neural Information Processing Systems*, 36.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. PAL: Program-aided language models. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 10764–10799. PMLR.

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. 2017. Multi-shot ASP solving with clingo. *CoRR*, abs/1705.09811.

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

Paul Gochet, Eric Gregoire, Pascal Gribomont, Georges Louis, Eduardo Sanchez, Dominique Snyers, and Pierre Wodon. 1988. *From standard logic to logic programming: introducing a logic based approach to artificial intelligence*. John Wiley & Sons, Inc.

Aaron Grattafiori et al. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Qiuhan Gu. 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*, pages 2201–2203.

Simeng Han, Hailey Schoelkopf, Yilun Zhao, Zhenting Qi, Martin Riddell, Wenfei Zhou, James Coady, David Peng, Yujie Qiao, Luke Benson, Lucy Sun, Alexander Wardle-Solano, Hannah Szabó, Ekaterina Zubova, Matthew Burtell, Jonathan Fan, Yixin Liu, Brian Wong, Malcolm Sailor, Ansong Ni, Linyong Nan, Jungo Kasai, Tao Yu, Rui Zhang, Alexander Fabbri, Wojciech Maciej Kryscinski, Semih Yavuz, Ye Liu, Xi Victoria Lin, Shafiq Joty, Yingbo Zhou, Caiming Xiong, Rex Ying, Arman Cohan, and Dragomir Radev. 2024. FOLIO: Natural language reasoning with first-order logic. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 22017–22031, Miami, Florida, USA. Association for Computational Linguistics.

Masoud Hashemi, Oluwanifemi Bamgbose, Sathwik Tejaswi Madhusudhan, Jishnu Sethumadhavan Nair, Aman Tiwari, and Vikas Yadav. 2025. Dnr bench: Benchmarking over-reasoning in reasoning llms. *arXiv preprint arXiv:2503.15793*.

Joy He-Yueya, Gabriel Poesia, Rose Wang, and Noah Goodman. 2023. Solving math word problems by combining language models with symbolic solvers. In *The 3rd Workshop on Mathematical Reasoning and AI at NeurIPS'23*.

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

Edward J Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Jie Huang and Kevin Chen-Chuan Chang. 2023. Towards reasoning in large language models: A survey. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 1049–1065, Toronto, Canada. Association for Computational Linguistics.

Adam Ishay, Zhun Yang, and Joohyung Lee. 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, pages 374–383.

Piyush Jha, Prithwish Jana, Pranavkrishna Suresh, Arnav Arora, and Vijay Ganesh. 2024. Rlsf: Reinforcement learning via symbolic feedback. *arXiv preprint arXiv:2405.16661*.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *Preprint*, arXiv:2406.00515.

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

Xin Lai, Zhuotao Tian, Yukang Chen, Senqiao Yang, Xiangru Peng, and Jiaya Jia. 2024. Step-dpo: Step-wise preference optimization for long-chain reasoning of llms. *Preprint*, arXiv:2406.18629.

Vladimir Iosifovich Levenshtein. 1965. Binary codes capable of correcting deletions, insertions, and reversals.

Lei Li, Zhihui Xie, Mukai Li, Shunian Chen, Peiyi Wang, Liang Chen, Yazheng Yang, Benyou Wang, and Lingpeng Kong. 2023. Silkie: Preference distillation for large visual language models.

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

Bill Yuchen Lin, Ronan Le Bras, Kyle Richardson, Ashish Sabharwal, Radha Poovendran, Peter Clark, and Yejin Choi. 2025. Zebralogic: On the scaling limits of llms for logical reasoning. *Preprint*, arXiv:2502.01100.

Ximing Lu, Sean Welleck, Peter West, Liwei Jiang, Jungo Kasai, Daniel Khashabi, Ronan Le Bras, Lianhui Qin, Youngjae Yu, Rowan Zellers, Noah A. Smith, and Yejin Choi. 2022. NeuroLogic a*esque decoding: Constrained text generation with lookahead heuristics. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 780–799, Seattle, United States. Association for Computational Linguistics.

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

Arindam Mitra and Chitta Baral. 2015. Learning to automatically solve logic grid puzzles. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1023–1033, Lisbon, Portugal. Association for Computational Linguistics.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. 2025. s1: Simple test-time scaling. *arXiv preprint arXiv:2501.19393*.

Aliakbar Nafar, Kristen Brent Venable, and Parisa Kordjamshidi. 2024. Probabilistic reasoning in generative large language models. *Preprint*, arXiv:2402.09614.

Xuefei Ning, Zinan Lin, Zixuan Zhou, Zifu Wang, Huazhong Yang, and Yu Wang. 2024. Skeleton-of-thought: Prompting llms for efficient parallel generation. In *Proceedings 12th international conference on learning representations-ICLR 2024*.

Farid Nouioua and Pascal Nicolas. 2006. Using answer set programming in an inference-based approach to natural language semantics. In *Proceedings of the Fifth International Workshop on Inference in Computational Semantics (ICoS-5)*.

Theo Olausson, Alex Gu, Ben Lipkin, Cedegao Zhang, Armando Solar-Lezama, Joshua Tenenbaum, and Roger Levy. 2023. LINC: A neurosymbolic approach for logical reasoning by combining language models with first-order logic provers. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 5153–5176, Singapore. Association for Computational Linguistics.

OpenAI et al. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. *Preprint*, arXiv:2203.02155.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. 2023. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 3806–3824.

Shuofei Qiao, Yixin Ou, Ningyu Zhang, Xiang Chen, Yunzhi Yao, Shumin Deng, Chuanqi Tan, Fei Huang, and Huajun Chen. 2023. Reasoning with language model prompting: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5368–5393, Toronto, Canada. Association for Computational Linguistics.

Qwen-Team. 2024. Qwen2.5: A party of foundation models.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2024. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36.

John Alan Robinson. 1983. Logic programming—past, present and future—. *New Generation Computing*, 1(2):107–124.

Timo Pierre Schrader, Lukas Lange, Simon Razniewski, and Annemarie Friedrich. 2024. QUITE: Quantifying uncertainty in natural language text in Bayesian reasoning scenarios. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 2634–2652, Miami, Florida, USA. Association for Computational Linguistics.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *Preprint*, arXiv:2402.03300.

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

Lewis Tunstall, Edward Beeching, Nathan Lambert, Nazneen Rajani, Shengyi Huang, Kashif Rasul, Alexander M. Rush, and Thomas Wolf. 2023. The alignment handbook. https://github.com/huggingface/alignment-handbook.

Nemika Tyagi, Mihir Parmar, Mohith Kulkarni, Aswin Rrv, Nisarg Patel, Mutsumi Nakamura, Arindam Mitra, and Chitta Baral. 2024. Step-by-step reasoning to solve grid puzzles: Where do LLMs falter? In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 19898–19915, Miami, Florida, USA. Association for Computational Linguistics.

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

Zhilin Wang, Yi Dong, Jiaqi Zeng, Virginia Adams, Makesh Narsimhan Sreedhar, Daniel Egert, Olivier Delalleau, Jane Polak Scowcroft, Neel Kant, Aidan Swope, and Oleksii Kuchaiev. 2023. Helpsteer: Multi-attribute helpfulness dataset for steerlm. *Preprint*, arXiv:2311.09528.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc.

Wenyi Xiao, Zechuan Wang, Leilei Gan, Shuai Zhao, Wanggui He, Luu Anh Tuan, Long Chen, Hao Jiang, Zhou Zhao, and Fei Wu. 2024. A comprehensive survey of direct preference optimization: Datasets, theories, variants, and applications. *Preprint*, arXiv:2410.15595.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zhihao Fan. 2024. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*.

Zhun Yang, Adam Ishay, and Joohyung Lee. 2023. Coupling large language models with logic programming for robust and general reasoning from text. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 5186–5219, Toronto, Canada. Association for Computational Linguistics.

Jiali Zeng, Fandong Meng, Yongjing Yin, and Jie Zhou. 2024. Improving machine translation with large language models: A preliminary study with cooperative decoding. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 13275–13288, Bangkok, Thailand. Association for Computational Linguistics.

Wanjun Zhong, Siyuan Wang, Duyu Tang, Zenan Xu, Daya Guo, Jiahai Wang, Jian Yin, Ming Zhou, and Nan Duan. 2021. Ar-lsat: Investigating analytical reasoning of text. *Preprint*, arXiv:2104.06598.

A Explanatory ASP Encoding

In this section, we provide a full step-by-step explanatory ASP encoding for one test instance of LogicPuzzles.

We take instance 17 from the test split:

Problem Description:
MVP Events prides itself on planning large-scale parties for 50 or more people, anywhere in the United States. This month the company has several different events to plan. Using only the clues below, match each event to its number of attendees and the state in which it will be held, and determine which MVP employee is handling the logistics.
Entities:
people: 50, 75, 100, 125.
planners: Herbert, Joel, Susan, Teresa.
events: Anniversary, Birthday, Graduation, Wedding.
Clues:
1. Of the anniversary event and the event with 100 attendees, one will be handled by Joel and the other will be handled by Susan.
2. Herbert's assignment will involve 25 fewer people than Susan's assignment.
3. Of the assignment with 75 attendees and the assignment with 100 attendees, one will be handled by Susan and the other is the birthday.
4. Herbert's event is either the event with 50 attendees or the graduation job.

There are three entity types (*people*, *planners*, *events*) with four subjects each. Hence, it is a 3×4 grid puzzle.

We begin each ASP encoding by defining all entities and constants that are involved in the search problem. This is, we start by defining the three

predicates `people`, `planners/1`, and `events/1` and assign the 12 entities to each category respectively:

```
people(50;75;100;125).
planners(herbert;joel;susan;teresa).
events(anniversary;birthday;
       graduation;wedding).
```

We could also encode 4 times `people/1` with a different person inside, but using semi-colons is a much shorter approach.

These 12 entities are now **hard facts** in the encoding, resulting in them being part of every single possible answer set.

Next, the problem asks for matching each event to its planner and number of attendees. In ASP, this is called *generate*, i.e., generating potential solutions. This is achieved by the so-called *choice rule*:

```
1 {assignment(Event, Planner, Attendees) :
  planners(Planner), people(Attendees)} 1
  :- events(Event).
```

The choice rule **generates** instances of `assignment/3` with triples (event, planner, attendees). The syntax `m ... n` says that at least m and at most n grounded instances of `assignment/3` must be generated. The semantics of this particular choice rule reads as follows: "For every `event/1` that is part of the answer set, generate exactly one `assignment/3` that contains a single event, a single planner and a single number of attendees." Since we defined four distinct events above, the choice rule will generate four grounded `assignment/3` instances. However, so far, it does not exclude that planner Herbert for example is assigned to two different events, only that every `assignment/3` will refer to a different `event/1`. Therefore, an addition to the choice rule is required, which is formulated as **rule**:

```
{E1 = E2; P1 = P2; A1 = A2} = 0
:- assignment(E1, P1, A1),
   assignment(E2, P2, A2),
   (E1, P1, A1) != (E2, P2, A2).
```

It reads as follows: "For two `assignment/3` that are part of the answer set, and the triples (event, planner, attendees) are not exactly the same, there must be zero overlap in any of the three entities event, planner, and number of attendees." As we have seen above, there are four `assignment/3` that are part of the answer set. This addition now excludes that two distinct events are for example assigned the same planner Herbert.

Next, we add all rules and constraints based on the hints provided by the problem instance.

The first clue is *1. Of the anniversary event and the event with 100 attendees, one will be handled by Joel and the other will be handled by Susan.* It carries multiple implications: The anniversary event does not have 100 attendees and Joel and Susan must plan one each of out of the anniversary event and the event with 100 attendees. In ASP, this is achieved by the following encoding:

```
{E = anniversary; A = 100} = 1
:- assignment(E, joel, A).
```

This rule says that if there is an `assignment/3` with Joel, it can only either have the anniversary event or the event with 100 attendees.

Likewise for Susan:

```
{E = anniversary; A = 100} = 1
:- assignment(E, susan, A).
```

The case that both Susan and Joel get the event with 100 attendees assigned is already excluded by the addition to the choice rule described above. Same holds for the anniversary event.

The next clue (*2. Herbert's assignment will involve 25 fewer people than Susan's assignment.*) is a **constraint** that excludes a specific condition (as opposed to rules that model if-else statements). In ASP, this is written as follows:

```
:- assignment(_, herbert, A1),
assignment(_, susan, A2),
not A1 == A2 - 25.
```

This is a constraint that requires at least one atom to evaluate to `false`. This constraint reads as follows: "Every answer set that contains one `assignment/1` for Herbert and one for Susan must not allow for Herbert's number of attendees being anything else than the number of Susan's subtracted by 25."

The third clue (*3. Of the assignment with 75 attendees and the assignment with 100 attendees, one will be handled by Susan and the other is the birthday.*) is modeled in the same way as clue 1:

```
{E = birthday; P = susan} = 1
:- assignment(E, P, 75).

{E = birthday; P = susan} = 1
:- assignment(E, P, 100).
```

Finally, the fourth clue (*4. Herbert's event is either the event with 50 attendees or the graduation job.*) is an *exclusive-OR* (XOR) that is modeled similar to clues 1 and 3:

```
{E = graduation; A = 50} = 1
:- assignment(E, herbert, A).
```

The entire encoding looks as follows:

```
people(50;75;100;125).
planners(herbert;joel;susan;teresa).
events(anniversary;birthday;
       graduation;wedding).

1 {assignment(Event, Planner, Attendees) :
  planners(Planner), people(Attendees)} 1
  :- events(Event).

{E1 = E2; P1 = P2; A1 = A2} = 0
:- assignment(E1, P1, A1),
   assignment(E2, P2, A2),
   (E1, P1, A1) != (E2, P2, A2).

{E = anniversary; A = 100} = 1
:- assignment(E, joel, A).

{E = anniversary; A = 100} = 1
:- assignment(E, susan, A).

:- assignment(_, susan, A2),
not A1 == A2 - 25.

{E = birthday; P = susan} = 1
:- assignment(E, P, 75).

{E = birthday; P = susan} = 1
:- assignment(E, P, 100).

{E = graduation; A = 50} = 1
:- assignment(E, herbert, A).
```

Running clingo on this encoding returns the following uniquely determined answer set:

```
planners(herbert) planners(joel)
planners(susan) planners(teresa)
people(50) people(75)
people(100) people(125)
events(anniversary) events(birthday)
events(graduation) events(wedding)
assignment(anniversary,susan,75)
assignment(wedding,herbert,50)
assignment(birthday,joel,100)
assignment(graduation,teresa,125)
```

By comparing all four instances of `assignment/3` to the clues, we can see that this answer set is indeed a stable model of the problem that fulfills all constraints.

B Levenshtein Heuristics

Since exact string matching to compare ground truth and ASP output is often not possible, we implement a Levenshtein heuristics that automatically detects whether an ASP output corresponds to the ground truth or not. To achieve that, we use the Levenshtein string edit distance (Levenshtein, 1965) that measures how many atomic string edit operations on a character-level (insert, delete, replace) are necessary to transform one string into another.

We want to explain this using an example first. Consider the following solution of instance with ID 2 from the test_HA split, i.e., the split without explanations of how to arrive at the correct solution, of LogicPuzzles:

```
(2016, ISON-X42, Dr. Golden)
(2017, Egert Facility, Dr. Owens)
(2018, Zynga Complex, Dr. Weber)
(2019, Bale-Hahn SSC, Dr. Farley)
```

Running clingo on the ASP encoding parsed by Llama-3.1 70B yields the following answer sets:

```
assignment(ison_x42,golden,2016)
assignment(bale_hahn_ssc,farley,2019)
assignment(egert_facility,owens,2017)
assignment(zynga_complex,weber,2018)
```

We can see that the main differences are dashes and spaces being converted into underscores, as well as some shortenings such as the prefix "Dr." being removed. These differences make it impossible to perform direct string comparisons. Therefore, for comparing computed output and ground truth, we first transform the ground truth into a representation as it could be used in ASP encodings. However, if comparison still fails, we apply our Levenshtein heuristics to compare both sets. This heuristic applies the following steps:

1. For each computed set of assignments, iterate over each ground truth tuple and every item contained in it and compare it to every single item in the computed answer sets. In this example, we compare every item out of the 12 ground truth entities to all 12 computed items. This results in a runtime complexity of $\mathcal{O}(n^2)$, with $n = 12$ in this running example. For example, taking `ison_x42` and comparing it to `(2016, ISON-X42, Dr. Golden)` results in the following three edit distances computed by NLTK's implementation of Levenshtein:

```
edit_distance("ison_x42",
```

```
                "2016") = 8
edit_distance("ison_x42",
                "ISON-X42") = 6
edit_distance("ison_x42",
                "Dr. Golden") = 10
```

After each comparison, we store the index tuple $(i, j), i, j \in [1, \ldots, 4]$ where i and j refer to the two row indices where the edit distance was the lowest for i. In the case above, we would have $(1, 1)$ unless there is another row where `ison_x42` needs less edits. However, if there are minimum edit distances, we perform an additional step that checks whether two strings are contained in each other or not. If so, this match will be preferred over the match without overlapping strings.

2. To judge whether a computed solution matches its ground truth counterpart, we check if for every item in ground truth row i the matching row j is the same. Furthermore we require j to be assigned to only one row i. For example, the following Levenshtein matrix indicates a valid solution for the running example:

```
[[(1,1), (1,1), (1,1)],
 [(2,3), (2,3), (2,3)],
 [(3,4), (3,4), (3,4)],
 [(4,2), (4,2), (4,2)]]
```

C Fine-Tuning Hyperparameters

To perform fine-tuning of large LLMs, we use LoRA fine-tuning (Hu et al., 2022, *Low-Rank Adaptation*), which adds trainable adapters to the base model. These adapters carry only a small percentage of trainable parameters relatively compared to the original base model (e.g., a LoRA rank of 128 for Llama-3.1 70B results in 1.6B trainable parameters).

We aim at finding the best hyperparameter settings by evaluating the trained models on the LogicPuzzles test split in the zero-shot setting. We find that the default SFT and DPO fine-tuning parameters of the Hugging Face alignment-handbook[10] (Tunstall et al., 2023) for their own Zephyr model are already very good parameters to start with.

We perform DPO and SFT training on 2-4 Nvidia H200 GPUs with model sharding enabled, i.e., the LLM parameters and optimizers are split across the 4 GPUs, which allows for bigger models to be trained. We use DeepSpeed ZeRO Stage 3[11] to perform sharding.

Table 5 lists the hyperparameters to fine-tune the three different open-weight models using the different training approaches. We find that higher LoRA ranks r seem to be important for capturing the nuances of ASP programming during DPO and SFT training. Moreover, it becomes clear that only single or very few training epochs are already sufficient to achieve good results. We also find that conducting too many DPO training epochs leads to a strong model degeneration, hence we do not recommend to train for too many epochs.

To mitigate for the fewer preference pairs sampled from the Qwen models compared to Llama-3.1 70B, we perform DPO for 3 epochs instead of 1 as we do with Llama-3.1 70B.

The SFT+DPO training setting uses for both steps the corresponding config from SFT and DPO.

The batch size is always equal to the number of GPUs used for training.

D Detailed Overview on each Evaluation Dataset

D.1 GridPuzzles Example

GridPuzzles (Tyagi et al., 2024) is very similar in its structure to LogicPuzzles. The main difference is that it goes beyond grid sizes of 3×4 by additionally providing problems of sizes 3×5, 4×4, 4×5, and 4×6.

The following instance with ID 8526 (note that there are still 274 instances) is an example for a 4×4 puzzle:

[10] https://github.com/huggingface/alignment-handbook/blob/95dc47218cf109659b74466d74be950525c53e67/recipes/zephyr-7b-beta/

[11] https://github.com/microsoft/DeepSpeed

	LoRA Rank r	LoRA α	DPO β	Learning Rate	Train Epochs	# GPUs
DPO						
Llama-3.1 70B	128	128	0.9	$5e-6$	1	4
Qwen-2.5 72B	128	128	0.1	$5e-6$	3	4
Qwen-2.5-Coder 32B	128	128	0.1	$5e-6$	3	4
SFT						
Llama-3.1 70B	128	128	–	$5e-5$	10	2
Qwen-2.5 72B	128	128	–	$5e-5$	10	2

Table 5: The hyperparameters used for training all LLMs.

Problem Description:
Maurice had several customers in his tattoo parlor today, each of which requested a simple tattoo of their astrological sign. Using only the clues below, match the prices to the options from customers, colors, and zodiac signs. Remember, as with all grid-based logic puzzles, no option in any category will ever be used more than once.
Entities:
prices: $35, $40, $45, $50.
customers: Bonita, Carole, Kendra, Neil.
colors: black, pink, red, violet.
zodiac signs: Pisces, Sagittarius, Taurus, Virgo.
Clues: 1. Bonita was the Taurus.
2. Of the person who paid $50 and the Virgo, one got the pink tattoo and the other got the violet tattoo.
3. The Taurus was either the customer who got the red tattoo or the customer who got the violet tattoo.
4. Kendra was either the person who paid $50 or the Pisces.
5. Of the customer who paid $35 and Neil, one got the red tattoo and the other was the Pisces.
6. Neil paid 10 dollars more than the customer who got the black tattoo.

We approach this dataset in the same way as LogicPuzzles, i.e., we start by defining all the constants:

```
price(35;40;45;50).
customer(bonita;carole;kendra;neil).
color(black;pink;red;violet).
zodiac_sign(pisces;sagittarius;taurus;virgo).
```

Next, we formulate the choice rule. However, we need to slightly modify it to fit to the 4×4 instance instead of the 3×4 ones from LogicPuzzles:

```
1 {assignment(P, C, CO, Z) :
    price(P), customer(C), color(CO)} 1
    :- zodiac_sign(Z).

{ P1 = P2; C1 = C2; CO1 = CO2; Z1 = Z2 } = 0
    :- assignment(P1, C1, CO1, Z1),
    assignment(P2, C2, CO2, Z2),
    (P1, C1, CO1, Z1) != (P2, C2, CO2, Z2).
```

The main difference to the choice rule of LogicPuzzles is that we now have 4 entity types in the choice rule.

Next, we look at the first hint: *1. Bonita was the Taurus.* This is a hard fact. However, in ASP, we cannot add facts with unknown arguments. However, we only know Bonita being the Taurus, but not how much was paid or what tattoo color there is. Hence, we need to formulate it as a constraint, thereby filtering out all answer sets generated by the choice rule that do not fulfill this constraint:

```
:- assignment(_, bonita, _, Z), Z != taurus.
```

The second hint (*2. Of the person who paid $50 and the Virgo, one got the pink tattoo and the other got the violet tattoo.*) enforces an XOR across two assignments:

```
{ Co1 = pink; Co2 = pink}  = 1
  :- assignment(50, _, Co1, _),
     assignment(_, _, Co2, virgo).
{ Co1 = violet; Co2 = violet } = 1
  :- assignment(50, _, Co1, _),
     assignment(_, _, Co2, virgo).
```

These two statements enforce pink being assigned to exactly one of both (the $50 person and the Virgo), as well as violet being assigned to only one of both. It is not possible that one gets both red and violet in the answer set as this is forbidden by the second statement of the choice rule block above.

The third clue (*3. The Taurus was either the customer who got the red tattoo or the customer who got the violet tattoo.*) is an XOR for one assignment, allowing for only one of two (Taurus being

the customer with red tattoo or violet tattoo) to be true:

```
{ Co = red; Co = violet } = 1
  :- assignment(_, _, Co, taurus).
```

Hint 4 (*4. Kendra was either the person who paid $50 or the Pisces.*) is modeled the same way as hint 3:

```
{ P = 50; Z = pisces } = 1
  :- assignment(P, kendra, _, Z).
```

Hint 5 (*5. Of the customer who paid $35 and Neil, one got the red tattoo and the other was the Pisces.*) is similar to hint 2:

```
{ Co1 = red; Co2 = red}  = 1
  :- assignment(35, _, Co1, _),
     assignment(_, neil, Co2, _).

{ Z1 = pisces; Z2 = pisces } = 1
  :- assignment(35, _, _, Z1),
    assignment(_, neil, _, Z2).
```

Finally, the last hint (*6. Neil paid 10 dollars more than the customer who got the black tattoo.*) is again an ASP constraints that filters all answer sets that do not adhere to this constraint:

```
:- assignment(P1, neil, _, _),
  assignment(P2, _, black, _),
  P1 != P2 + 10.
```

To summarize, this is the full ASP encoding for this GridPuzzles instance:

```
price(35;40;45;50).
customer(bonita;carole;kendra;neil).
color(black;pink;red;violet).
zodiac_sign(pisces;sagittarius;taurus;virgo).

1 {assignment(P, C, CO, Z) :
  price(P), customer(C), color(CO)} 1
  :- zodiac_sign(Z).
{ P1 = P2; C1 = C2; CO1 = CO2; Z1 = Z2 } = 0
  :- assignment(P1, C1, CO1, Z1),
    assignment(P2, C2, CO2, Z2),
    (P1, C1, CO1, Z1) != (P2, C2, CO2, Z2).

:- assignment(_, bonita, _, Z), Z != taurus.

{ Co1 = pink; Co2 = pink}  = 1
  :- assignment(50, _, Co1, _),
  assignment(_, _, Co2, virgo).
{ Co1 = violet; Co2 = violet } = 1
  :- assignment(50, _, Co1, _),
  assignment(_, _, Co2, virgo).

{ Co = red; Co = violet } = 1
  :- assignment(_, _, Co, taurus).

{ P = 50; Z = pisces } = 1
  :- assignment(P, kendra, _, Z).

{ Co1 = red; Co2 = red}  = 1
  :- assignment(35, _, Co1, _),
  assignment(_, neil, Co2, _).

{ Z1 = pisces; Z2 = pisces } = 1
  :- assignment(35, _, _, Z1),
  assignment(_, neil, _, Z2).

:- assignment(P1, neil, _, _),
  assignment(P2, _, black, _),
  P1 != P2 + 10.
```

D.2 GSM-Algebra Example

GSM-Algebra (He-Yueya et al., 2023) is fundamentally different to the other three datasets as it requires solving an algebraic question instead of a grid-based matching problem.

We restrict the search space from $-15k$ to $15k$ as the solving time grows exponentially with size. We evaluate integer-based instances only (70.3% of all instances) since ASP does not support floating point arithmetic.

For example, the instance with ID 163 looks as follows:

> **Problem Description:**
> Dolores bought a crib on sale for $350. The sale price was 40% of the original price. What was the original price of the crib?

To model this problem in ASP, we start in the exact same way as with all the other problems, which is to define constants. However, there are no explicit entities such as countries or people. Instead, the algebraic group of integers $\mathbb{Z}$ is of relevance. Therefore, we first define a search space s.t. the solver can use this number space for grounding later:

```
number(-15000..15000).
```

This defines a search space of a total of 30.001 numbers. Note that with increasing search space, the computation time increases.

Next, we define the price of the crib as a hard fact in the ASP encoding:

```
price_of_crib(350).
```

Finally, we formulate the equation that computes the answer `Z` by checking all possible combinations produced by the answer space `number`:

```
result(Z) :- number(Z), price_of_crib(X),
    Z = 10 * X / 4.
```

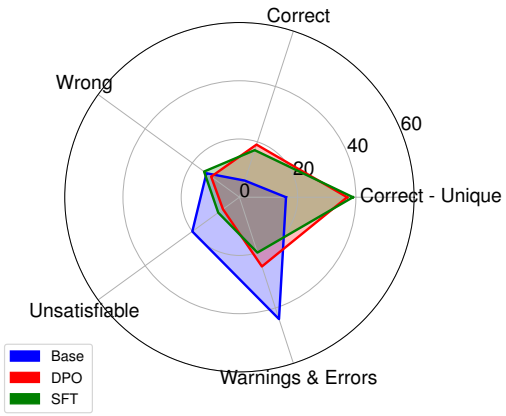

Figure 2: Result type comparison between the base Llama-3.1 70B and its DPO and SFT trained counterparts on LogicPuzzles in the two-shot setting.

This equation starts by defining `Z` as a number. If it was without this statement, clingo would return an error as the type of `Z` has never been defined before. Next, `X` can only be grounded to 350 as this is a hard fact stated above. Finally, the original price is calculated by using $Z = \frac{10}{4} * X$, equaling to 875.

The entire ASP encoding looks as follows:

```
number(-15000..15000).
price_of_crib(350).
result(Z) :- number(Z), price_of_crib(X),
    Z = 10 * X / 4.
```

E Further Analysis

Figure 2 visualizes the distribution of the different evaluation categories for Llama-3.1 70B on LogicPuzzles in the two-shot setting. It displays the distribution for the base Llama-3.1 as well as the SFT and DPO counterparts.

We can see that both SFT and DPO lead to similar, positive shifts towards more correctly solvable instances. The base Llama-3.1 produces many erroneous and unsatisfiable instances, whereas the trained variants are able to produce much more correct ASP encodings. This underlines that our solver-guided training method is able to adapt the models to better ASP coders. Furthermore, both DPO and SFT lead to very similar results, showing solver-guided training is very valuable for both training approaches.

Status	Description
Correct	Single and correct answer set returned.
Partially correct	Multiple answer sets including the single correct one.
Wrong	None of the answer sets are correct solutions.
Unsatisfiable	Contradicting constraints; no answer set exists.
Warnings & errors	Incorrect encoding leads to warnings/errors, preventing outputs.

Table 6: Evaluation categories for ASP encodings.

F DPO Loss Function

During DPO fine-tuning, a model is shown both a chosen and rejected response for a given input and optimized for the following loss function:

$$\mathcal{L}_{\text{DPO}}(\pi_\theta; \pi_{\text{ref}}) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}} \left[\log \sigma \left(\beta \log \frac{\pi_\theta(y_w \mid x)}{\pi_{\text{ref}}(y_w \mid x)} - \beta \log \frac{\pi_\theta(y_l \mid x)}{\pi_{\text{ref}}(y_l \mid x)}\right)\right].$$

π_{ref} is the reference model, in our case $\mathcal{M}_S$ from which the preference data was sampled. π_θ is the LLM to be optimized. Initially, it holds that $\pi_{\text{ref}} = \pi_\theta$. Intuitively, the loss objectives increases the likelihood margin between the chosen and the rejected response. Hence, the overall goal is to make the chosen responses more likely relative to the rejected ones.

G Evaluation Taxonomy

Table 6 shows the five distinct categories which we use to evaluate LLM-generated ASP encodings.

H Number of Possible Solutions for Grid-based Puzzles

In this section, we want to provide a visual explanation for why the number of possible solutions of an $m \times n$ assignment equals to $(n!)^{(m-1)}$, i.e., for a puzzle with m entity categories with n entities in each category.

Exemplarily, we use a 3×4 grid puzzle. That could be matching four dogs with four owners and four houses.

We approach this combinatorial problem as graph as shown in figure 3. We start by matching the first 4 entities with another 4 entities and only allow for $1:1$ matchings. When selecting the first entity of category 1, there are 4 opportunities to match it with an entity of category two.

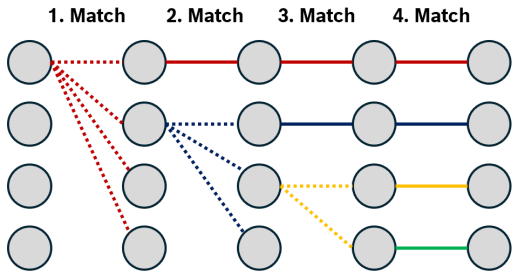

Figure 3: Matching two types of entities with four subjects each. There are $4 \times 3 \times 2 \times 1 = 4!$ possibilities to find exclusive matchings.

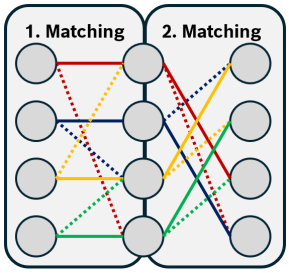

Figure 4: Visualization of an 3×4 grid puzzles that requires $2 = 3 - 1$ independent matchings with $4!$ possibilities in each subgraph.

Afterwards, when matching the second entity of category 1, there are only 3 entities left from category 3, resulting in only 3 opportunities. Likewise, the third entity of category 1 has only 2 opportunities left for a matching, while the final one from category 1 has only a single matching possibility. This results in $4 \times 3 \times 2 \times 1 = 4!$ opportunities. The general number for n entities per category is therefore $n!$.

The same holds for the second matching between entities from category 2 and category 3. Hence, we get $2 = 3 - 1$ independent matching problems as shown in figure 4. The general formula hence is $m - 1$. Putting all together results in $(n!)^{(m-1)}$ possible solutions for an unconstrained grid puzzle.

I Evaluation on GSM-Algebra

To show that our method also improves LLMs on generating ASP encodings for different use-cases, we conduct a small experiment in which we run our trained models on GSM-Algebra (He-Yueya et al., 2023), which requires solving algebraic problems. Table 7 shows the accuracy of each model. As ASP does not support floating points, we only focus on instances that work with integers.

With the exception of the coding-specialized Qwen, we can see improvements for each model over its untrained counterpart. The performance increase is especially large for the 70B Llama model with about 8 pp. This indicates that our method of training models on ASP for grid-based puzzles also generalizes to other domains. We interpret the very good performance of the Qwen Coder model as a natural implication of the fact that it was trained specifically for code generation. Since algebraic questions are much closer to traditional coding questions than semantic parsing, we believe that this is where the coding knowledge excels. However, our method also brings the non-coding models much closer to the performance of the coder model.

		GSM-Alg.
Llama-3.1 70B	*Base*	60.3
	DPO	68.6
	SFT	64.7
	SFT+DPO	64.1
Qwen-2.5 72B	*Base*	60.9
	DPO	60.9
	SFT	64.1
	SFT+DPO	65.4
Qwen-2.5-Coder 32B	*Base*	**76.3**
	DPO	73.1

Table 7: Percentage of correct solutions (↑) on GSM-Algebra.

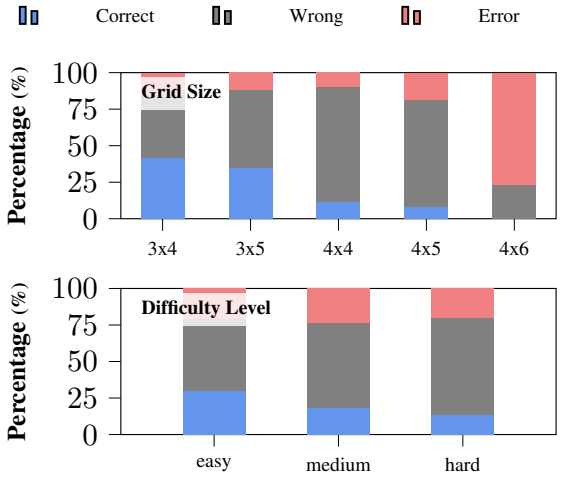

Figure 5: Llama-3.1 70B SFT+DPO combined with best-of-N sampling results on GridPuzzles, categorized by grid sizes (top) and difficulty levels (bottom). We can see that our neuro-symbolic approach has almost equal performance on medium and hard puzzles, but degrades with increasing puzzles size.

J Performance Analysis on GridPuzzles

Figure 5 breaks down the results of Llama-3.1 70B SFT+DPO combined with best-of-N sampling on

GridPuzzles by grid size (top) and by problem difficulty (bottom).

First, increasing puzzles size leads to decreasing performance. There are mainly two reasons for that: the LLM-based generation degrades due to the increasing context and puzzles of very large sizes might not be solvable due to a huge potential solution space that could lead to out-of-memory issues. Second, the human-judged difficulty does not seem to have the same big influence as the puzzle size since there is almost equal performance between puzzles of medium and hard difficulty. However, we still observe a drop compared to the easy puzzles, mainly because medium and hard puzzles contain more XOR-based constraints.