Prototype-then-Refine: A Neurosymbolic Approach for Improved Logical Reasoning with LLMs

Hashem Elezabi [1] Bassem Akoush [2]

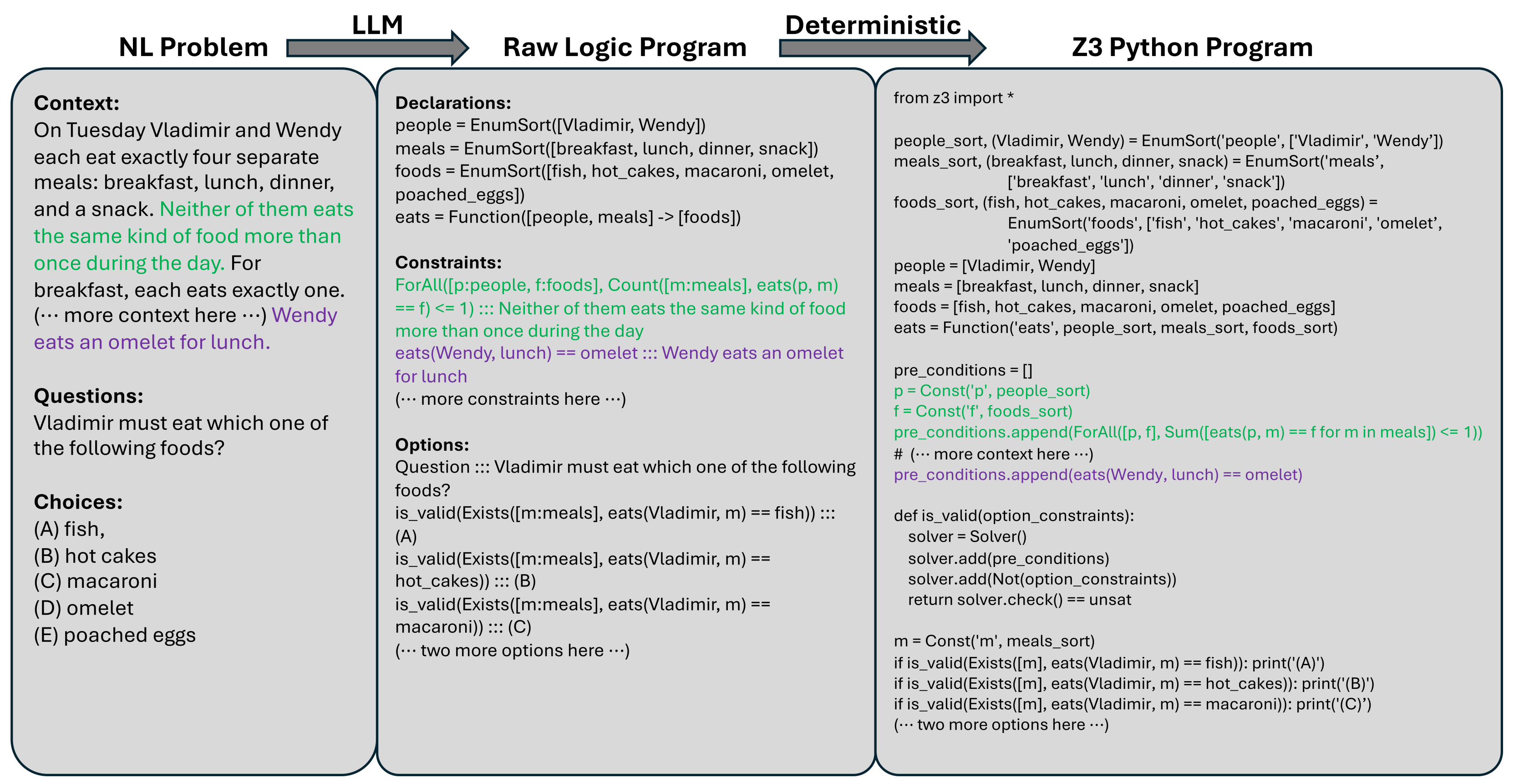
[1]Department of Computer Science [2]Department of Mechanical Engineering

Introduction

- Large language models (LLMs) have shown remarkable capabilities, but even the best LLMs still struggle with complex logical reasoning problems. GPT-4 achieves a low accuracy of 35.06% with Chain-of-Thought prompting on the challenging AR-LSAT dataset[1].
- Recent works like Logic-LM[2] showed how one can improve logical reasoning ability by combining LLMs with symbolic solvers. This neuro-symbolic approach works by 1) prompting the LLM to translate a natural language (NL) problem into a *logic program* and 2) executing a deterministic symbolic solver on the logic program to answer the given question. Logic-LM increases GPT-4 accuracy on AR-LSAT to 43.04%, but there is still a long way to go.
- **Building on Logic-LM, we propose *Prototype-then-Refine (ProRef)*, a new and extensible framework that aims to improve the ability of LLMs to generate correct logic programs using a series of arbitrarily powerful *prototypes* and *refiners*.**

Datasets

We used the dev split of the AR-LSAT dataset in our work. This dataset contains 231 logical reasoning questions from the Law School Admission Test between years 1991 and 2016. Given a problem, we formulate a prompt that includes instructions, the problem statement, the question, and multiple-choice answers. The task is to generate a logic program and output the correct answer out of the five given options. The NL problem below is one example from the dataset. It's translated into a raw logic program via an LLM, then deterministically mapped to a Z3 Python program that serves as our symbolic solver.


NL Problem
LLM
Raw Logic Program
Deterministic
Z3 Python Program

Context:
On Tuesday Vladimir and Wendy each eat exactly four separate meals: breakfast, lunch, dinner, and a snack. Neither of them eats the same kind of food more than once during the day. For breakfast, each eats exactly one. (⋯ more context here ⋯) Wendy eats an omelet for lunch.

Questions:
Vladimir must eat which one of the following foods?

Choices:
(A) fish,
(B) hot cakes
(C) macaroni
(D) omelet
(E) poached eggs

```
Declarations:
people = EnumSort([Vladimir, Wendy])
meals = EnumSort([breakfast, lunch, dinner, snack])
foods = EnumSort([fish, hot_cakes, macaroni, omelet,
poached_eggs])
eats = Function([people, meals] -> [foods])

Constraints:
ForAll([p:people, f:foods], Count([m:meals], eats(p, m)
== f) <= 1) ::: Neither of them eats the same kind of food
more than once during the day
eats(Wendy, lunch) == omelet ::: Wendy eats an omelet
for lunch
(⋯ more constraints here ⋯)

Options:
Question ::: Vladimir must eat which one of the following
foods?
is_valid(Exists([m:meals], eats(Vladimir, m) == fish)) :::
(A)
is_valid(Exists([m:meals], eats(Vladimir, m) ==
hot_cakes)) ::: (B)
is_valid(Exists([m:meals], eats(Vladimir, m) ==
macaroni)) ::: (C)
(⋯ two more options here ⋯)
```

```
from z3 import *

people_sort, (Vladimir, Wendy) = EnumSort('people', ['Vladimir', 'Wendy'])
meals_sort, (breakfast, lunch, dinner, snack) = EnumSort('meals',
                 ['breakfast', 'lunch', 'dinner', 'snack'])
foods_sort, (fish, hot_cakes, macaroni, omelet, poached_eggs) =
                 EnumSort('foods', ['fish', 'hot_cakes', 'macaroni', 'omelet',
                 'poached_eggs'])
people = [Vladimir, Wendy]
meals = [breakfast, lunch, dinner, snack]
foods = [fish, hot_cakes, macaroni, omelet, poached_eggs]
eats = Function('eats', people_sort, meals_sort, foods_sort)

pre_conditions = []
p = Const('p', people_sort)
f = Const('f', foods_sort)
pre_conditions.append(ForAll([p, f], Sum([eats(p, m) == f for m in meals]) <= 1))
# (⋯ more context here ⋯)
pre_conditions.append(eats(Wendy, lunch) == omelet)

def is_valid(option_constraints):
  solver = Solver()
  solver.add(pre_conditions)
  solver.add(Not(option_constraints))
  return solver.check() == unsat

m = Const('m', meals_sort)
if is_valid(Exists([m], eats(Vladimir, m) == fish)): print('(A)')
if is_valid(Exists([m], eats(Vladimir, m) == hot_cakes)): print('(B)')
if is_valid(Exists([m], eats(Vladimir, m) == macaroni)): print('(C)')
(⋯ two more options here ⋯)
```

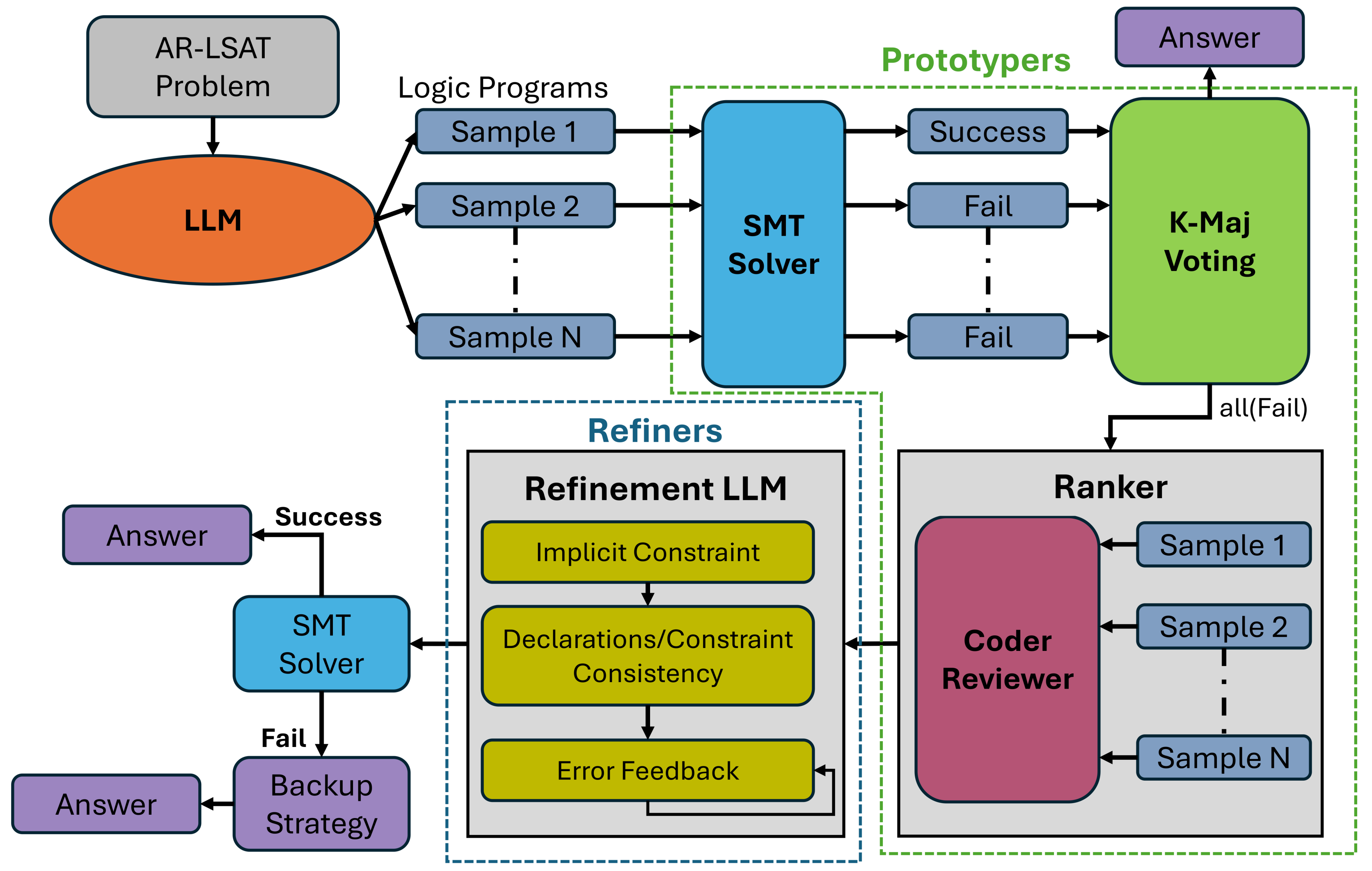
Methods

Our framework composes a series of *prototypers* in the Prototype stage:

- ***k*-majority prototyper.** Using a NL instruction x, sample k logic programs $y_{i\in 1:k}$ from a prompted LLM $p(y_i|x)$, execute them with a solver that outputs one of five possible answers (or Error), and output the majority answer among non-error answers.
- **Coder-Reviewer prototyper.** If all k samples lead to errors, we have no execution-based feedback, so we pick one final prototype via Coder-Reviewer reranking[3], where we first translate generated programs back to NL instructions via a *Reviewer* model $p(y|x)$ (created by prompting an LLM), then rank programs using the combined likelihood of the forward *Coder* and backward *Reviewer* models: $\log p(y|x)p(x|y) = \log p(y|x) + \log p(x|y)$. We retrieve log-probablities by setting `logprobs=True` in the OpenAI API call.

Next, we apply a series of *refiners* in the Refine stage:

- **Implicit constraints refiner.** This prompts the LLM to identify implicit constraints that are not explicitly mentioned in the problem statement, with the aim being to capture constraints that are common sense to humans but need to be explicitly enforced in the logic formulas, e.g. that Harry is a person.
- **Consistency refiner.** This prompts the LLM to fix inconsistencies between the Declarations and Constraints in the logic program, to help ensure the naming of the variables is consistent, e.g. if Declarations include a `children` array and a `gender()` function, Constraints shouldn't refer to `boys` and `girls` arrays since they aren't defined.
- **Error feedback refiner.** This refiner is based on the self-refinement module of Logic-LM[2], which prompts the LLM with the error message and asks it to fix the program.



Experiments

We benchmarked Logic-LM with different LLMs, including both open-source and closed models. We then ran our ProRef framework with GPT-3.5-Turbo.

Model		Exec. Rate	Exec. Accuracy	Overall Accuracy
Logic-LM	**GPT-4**	**32.61%**	**60%**	**34.78%**
	GPT-3.5-Turbo	20.05%	31.82 %	25.53%
	Mixtral-8x7B-Instruct	2.60%	16.67	22.94%
	vicuna-13b-v1.5	3.48%	37.5%	24.24%
ProRef (Ours)	**GPT-3.5-Turbo**	**32.47%**	**28%**	**23.81%**

Below we show the incremental effect of adding the different prototypers and refiners. As an ablation, we report the results with and without Coder-Reviewer reranking (CR).

	Exec. Rate		Exec. Accuracy		Overall Accuracy	
	w/ CR	w/o CR	w/ CR	w/o CR	w/ CR	w/o CR
k-Majority Voting	27.27%	27.27%	26.98%	26.98%	22.51%	22.51%
Implicit Constraints	27.27%	29.00%	26.98%	28.36%	22.51%	22.94%
Consistency	27.27%	29.00%	26.98%	28.36%	22.51%	22.94%
Error Feedback @1	32.03%	32.46%	27.03%	25.33%	23.38%	25.54%
Error Feedback @2	32.03%	32.47%	27.03%	25.33%	23.38%	22.51%
Error Feedback @3	32.04%	32.47%	27.03%	25.33%	23.38%	22.51%
Error Feedback @4	32.47%	32.47%	28.00%	25.33%	23.81%	22.51%

For further analysis, we parse the generated error messages into the 9 error categories below. This table shows their counts for different methods. We also include error numbers after excluding all our prototypers and refiners ("w/o PR") to evaluate how they impact the errors.

Model	Logic-LM				ProRef (Ours), GPT-3.5	
	GPT-4	GPT-3.5	Mixtral	Vicuna	w/ PR	w/o PR
Underconstrained	33	12	3	5	2	5
Overconstrained	13	22	7	1	10	10
TypeError	29	41	5	22	23	13
SyntaxError	26	39	144	113	51	41
NameError	26	11	12	16	26	4
AttributeError	0	5	0	6	3	2
Parsing Error	14	13	52	22	25	24
Z3 Exception	11	40	2	38	18	29
No Output	3	4	0	0	10	28
Total	155	189	225	223	168	156

Underconstrained: Programs that are missing essential constraints, causing the solver to evaluate all choices as satisfaible. **Overconstrained:** Programs that are too restrictive or even contradictory, causing the solver to evaluate all choices as unsatisfiable.

Result Analysis

- Compared to Logic-LM, our framework significantly improves the executable rate of generated logic programs from 20.05% to 32.47% (12.42% gain), enabling GPT-3.5-Turbo to almost match the executable rate of the much more expensive GPT-4 (32.61%).
- The Prototype stage increases Exec. Rate from 20.05% to 27.27% (7.22% gain) while the Refine stage further increases it from 27.27% to 32.47% (5.2% gain).
- However, the Exec. Accuracy and Overall Accuracy decrease. This means that by only optimizing for executable programs, we aren't necessarily able to get more *correct* programs.
- Our ablation shows that the Coder-Reviewer reranker is effective in improving the final Exec. and Overall accuracies after refinement, from 25.3% to 28% and from 22.51% to 23.81% respectively, showing that Coder-Reviewer helps pick higher-quality candidates for refinement.
- After running a bare version of ProRef ("w/o PR"), we find that ProRef decreases the count of TypeErrors, SyntaxErrors, NameErrors, AttributeErrors, and ParsingErrors, but increases Underconstrained, Z3 Exception, and No Output errors.

Conclusion & Future Work

Our ProRef framework improves the Executable Rate of LLM-generated logic programs by an absolute gain of 12.42% on the AR-LSAT dev set. We also see an improvement in handling various error types, especially NameErrors and TypeErrors. However, our method doesn't improve Executable or Overall Accuracy. Below we propose some ideas that can improve the accuracy of our framework:

- To mitigate syntax errors, we can incorporate descriptive feedback from Python translations into the error feedback refinement process, including line numbers where the error occurred.
- We can implement a *constrained decoder prototyper* that ensures adherence to correct grammar rules in generated logic programs.
- To improve the *correctness* of logic programs, we can implement more advanced *semantics-guided* refiners that use semantic heuristics to attempt to refine logic program prototypes into more correct ones. This can be based on failure patterns observed in incorrect programs from the training or dev set.

References

[1] Wanjun Zhong, Siyuan Wang, Duyu Tang, Zenan Xu, Daya Guo, Jiahai Wang, Jian Yin, Ming Zhou, and Nan Duan. Ar-lsat: Investigating analytical reasoning of text. arxiv e-prints, page. *arXiv preprint arXiv:2104.06598*, 2021.

[2] Liangming Pan, Alon Albalak, Xinyi Wang, and William Yang Wang. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. *arXiv preprint arXiv:2305.12295*, 2023.

[3] Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida Wang. Coder reviewer reranking for code generation. In *International Conference on Machine Learning*, pages 41832–41846. PMLR, 2023.