Instantiation-based Formalization of Logical Reasoning Tasks using Language Models and Logical Solvers

Mohammad Raza and **Natasa Milic-Frayling**
Qatar Computing Research Institute

Abstract

Robustness of reasoning remains a significant challenge for large language models, and addressing it is essential for the practical applicability of AI-driven reasoning systems. We introduce Semantic Self-Verification (SSV), a novel approach that addresses the key challenge in combining language models with the rigor of logical solvers: to accurately formulate the reasoning problem from natural language to the formal language of the solver. SSV uses a consistency-based approach to produce strong abstract formalizations of problems using concrete instantiations that are generated by the model and verified by the solver. In addition to significantly advancing the overall reasoning accuracy over the state-of-the-art, a key novelty that this approach presents is a feature of verification that has near-perfect precision over a significant coverage of cases, as we demonstrate on open reasoning benchmarks. We propose such *near-certain reasoning* as a new approach to reduce the need for manual verification in many cases, taking us closer to more dependable and autonomous AI reasoning systems.

1 Introduction

Logical reasoning remains a persistent challenge for large language models (LLMs). Although these models demonstrate reasoning capabilities across various domains, their reasoning often lacks robustness and becomes increasingly error-prone as task complexity increases. Many recent approaches have made notable advancements in this active area of research. Chain-of-thought (CoT) prompting has demonstrated how the quality of reasoning can be improved by prompting the model to explicitly generate the steps of reasoning in natural language before arriving at the final answer [Wei *et al.*, 2022]. Variants of CoT and other related prompting and fine-tuning approaches have shown further improvements [Zhou *et al.*, 2023; Wang *et al.*, 2023; Yu *et al.*, 2024; Weng *et al.*, 2023; Creswell *et al.*, 2023]. To address the logical inconsistencies that can arise in such natural language approaches, another interesting direction is to incorporate LLMs with logical solvers or automated reasoning tools [Pan *et al.*, 2023; Ye *et al.*, 2023]. Rather than directly attempting reasoning with the LLM, these approaches use the LLM to infer a formal representation of the problem as a program that can be executed by the solver, as such automated reasoning tools guarantee logically sound inference by construction.

While these approaches have demonstrated relative improvements in accuracy, we are still far from achieving robustness and reliability of reasoning. For instance, Figure 1 shows an example reasoning problem from the Law School Admissions Test on analytical reasoning [Zhong *et al.*, 2022]. On tasks of such complexity, the best reported accuracy, achieved by a solver-augmented system, is only 43% [Pan *et al.*, 2023]. Such lack of reliability especially hinders the practical usability of existing approaches: the burden of verifying correctness is *always* on the user, which can be especially difficult and error-prone for complex reasoning tasks. Therefore, having a reliable signal of correctness with high confidence can be hugely beneficial to help reduce the overall manual effort and cost of verification.

In this work, we propose a new approach to correctly formalizing reasoning problems called *Semantic Self-Verification* (SSV), which offers two key benefits: (1) it improves the overall accuracy of reasoning significantly over SoTA, and (2) it provides a novel feature of verification that has *near-perfect* precision. In our problem formulation, in addition to producing an answer to a given question, the system also indicates if it was able to *verify* the correctness of the answer: $\mathsf{Question} \rightarrow (\mathsf{Answer}, \mathsf{isVerified})$. This problem formulation is similar to confidence estimation in machine learning, except that in our case the $\mathsf{isVerified}$ indicator is a boolean rather than continuous value: if true, it indicates a "near certain" confidence in the correctness of the answer. Such high-confidence verification can reduce the need for manual checking in many cases.

At its core, our approach addresses the key challenge in combining LLMs with the robust reasoning of logical solvers: the formulation of a problem from informal natural language (NL) to the formal representation that is a program executable by the solver. For example, Figure 2 shows the formal representation of the NL problem from Figure 1. In this case the formalization is expressed as code in the language of the Z3 SMT solver [de Moura and Bjørner, 2008], which is a state-of-the-art industrial strength theorem prover that can produce the correct answer when given these correctly-expressed formal constraints. The crucial task, therefore, is for the LLM

In a repair facility, there are exactly six technicians: Stacy, Urma, Wim, Xena, Yolanda, and Zane. Each technician repairs machines of at least one of the following three types—radios, televisions, and VCRs—and no other types. The following conditions apply: Xena and exactly three other technicians repair radios. Yolanda repairs both televisions and VCRs. Stacy does not repair any type of machine that Yolanda repairs. Zane repairs more types of machines than Yolanda repairs. Wim does not repair any type of machine that Stacy repairs. Urma repairs exactly two types of machines. Which one of the following pairs of technicians could repair all and only the same types of machines as each other?

(A) Stacy & Urma
(B) Urma & Yolanda
(C) Urma & Xena
(D) Wim & Xena
(E) Xena & Yolanda

Figure 1: Sample problem from the Law School Admissions Test

```
technicians = [Stacy, Urma, Wim, Xena, Yolanda, Zane]
machines = [radios, televisions, VCRs]
repairs = Function('repairs', technicians_sort, machines_sort, BoolSort())

pre_conditions = []
pre_conditions.append(ForAll([t], Sum([If(repairs(t, m), 1, 0) for m in
machines]) >= 1))

# CONSTRAINT: Xena and exactly three other technicians repair radios.
pre_conditions.append(And(repairs(Xena, radios), Sum([If(And(t != Xena, repairs
(t, radios)), 1, 0) for t in technicians]) == 3))

# CONSTRAINT: Yolanda repairs both televisions and VCRs.
pre_conditions.append(And(repairs(Yolanda, televisions), repairs(Yolanda,
VCRs)))

# CONSTRAINT: Stacy does not repair any type of machine that Yolanda repairs.
pre_conditions.append(ForAll([m], Not(And(repairs(Stacy, m), repairs(Yolanda,
m)))))

# CONSTRAINT: Zane repairs more types of machines than Yolanda repairs.
pre_conditions.append(Sum([If(repairs(Zane, m), 1, 0) for m in machines]) > Sum
([If(repairs(Yolanda, m), 1, 0) for m in machines]))

# CONSTRAINT: Wim does not repair any type of machine that Stacy repairs.
pre_conditions.append(ForAll([m], Implies(repairs(Stacy, m), Not(repairs(Wim,
m)))))

# CONSTRAINT: Urma repairs exactly two types of machines.
pre_conditions.append(Sum([If(repairs(Urma, m), 1, 0) for m in machines]) == 2)

# OPTION A:
if is_sat(And(ForAll([m], repairs(Stacy, m) == repairs(Urma, m)))): print('(A)')

# OPTIONS B to E stated similarly ...
```

Figure 2: Sample problem formalization as Z3 code

to correctly translate the NL problem description to such a formal representation, and this is where LLMs can make significant errors, as shown by the limits of prior work [Pan *et al.*, 2023; Ye *et al.*, 2023].

Our approach of verifying that a formal representation is true to the original problem is inspired by how humans often create formalizations of problems expressed in natural language. For instance, when school students solve math word problems, they must first create the right algebraic equation that represents the problem, before they can solve it to get the answer. To ensure that their translation to an abstract equation represents the problem correctly, they are encouraged to consider various example instances of the problem and to check that the abstract equation consistently satisfies those instances so that it all "makes sense". In the same way, in the SSV approach, rather than just doing a single abstract translation from NL to a formal representation, we also use the LLM to additionally generate various *concrete instantiations*, or examples, of the general constraint, which are used as test cases to check the correctness of the abstract formalization. Using the logical solver, we verify that each of these instantiations is consistently satisfied by the formal representation. If all of these distinct semantic relationships consistently hold, then verification passes.

Figure 4 illustrates how the SSV approach works for the third constraint from the problem in Figure 2, which requires that Stacy and Yolanda cannot repair the same type of machine. A direct translation using the LLM may produce an incorrect abstract formalization of this constraint as shown in Figure 4a, where the condition is asserted only *for some* machine rather than *for all* machines because the Exists quantifier is incorrectly used. However, in the SSV approach, we use the LLM to additionally infer simple concrete instantiations, or examples, of the general constraint. For instance, a concrete positive example is that Stacy repairs radios and Yolanda repairs TVs. A negative example is that Stacy and Yolanda both repair TVs. After inferring these examples in NL, we also use the LLM to translate them to formal expressions in the language of the solver. We then use the solver to check that each of these expressions is consistent with the abstract formalization. In Figure 4a we see that the negative instantiation fails verification because the abstract formalization does not assert the condition for all machine types, so it still allows Stacy and Yolanda to both repair TVs. However, with the correct formalization in Figure 4b that uses the ForAll quantifier, both instantiations pass the solver verification, since the abstract formalization correctly disallows that *any* machine can be repaired by both technicians.

We note that any notion of verification from natural to formal language cannot provide formal correctness guarantees, since natural language itself is inherently informal and often ambiguous. However, as we demonstrate empirically, a passing verification in our case indicates a *near certain* confidence in the answer correctness since multiple independent semantic relationships are consistently satisfied. In this respect, our approach is akin to a consensus-based ensemble as it is based on agreement between multiple independent predictors [Zhou, 2012]. However, rather than all predictors addressing *the same* task, we have a *semantic ensemble* of predictors that are addressing different but semantically related tasks and the logical solver verifies the formal consistency between these. We also note that unlike standard proposer-verifier approaches, in our case there is no verifier that can check *correctness* of a proposed formalization: our verification is thus based on formal *consistency* between abstract and concrete inferences.

Furthermore, having such a high precision verification mechanism also allows us to improve the formalization itself, in two different respects. Firstly, any failing instantiation can be used as concrete guidance to refine the formalization further, as it can hint at potential errors. This is similar to

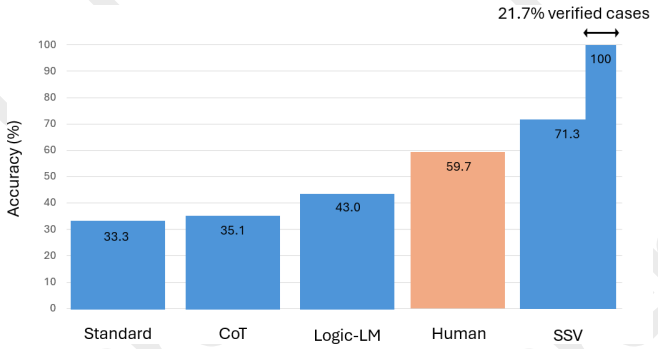


Figure 3: Towards near-perfect reasoning: SSV achieves new SoTA accuracy and 100% verification precision on the AR-LSAT law school tests dataset (*all systems using GPT-4 as base LLM*).

error-based refinement in code generation techniques [Chen *et al.*, 2024], except that here we are guided by *semantic* errors inferred from the instantiations rather than just *syntactic* execution errors in the code. Secondly, with our verification mechanism we can also explore the search space more extensively: using temperature sampling to create multiple candidate formalizations and selecting ones that pass verification.

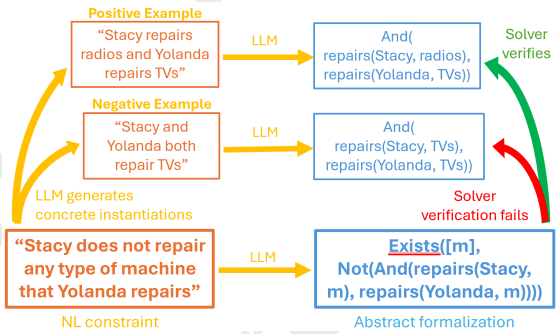
Our evaluation demonstrates how the SSV approach achieves a significant increase in overall accuracy, as well as a near-perfect precision (or selective accuracy) on the verified cases. Figure 3 highlights the results for the most challenging AR-LSAT law school tests dataset. Though better than direct LLM inference and CoT, the accuracy of the best performing existing system (the solver-augmented Logic-LM approach by [Pan *et al.*, 2023]) is at 43%, while SSV achieves a significantly higher accuracy of 71.3%, which also surpasses the average human performance. Moreover, the precision of the 21.7% of cases that it is able to verify is 100%. This means that a 21.7% reduction in manual verification effort can potentially be made on tasks of such high complexity. In our full evaluation we also show higher accuracy and coverage of verified cases on other standard reasoning datasets.

In summary, we make the following contributions in this work: (1) We propose the problem formulation of returning a boolean high-confidence verification indication in addition to the answer, which can be used to reduce manual cost of verification. (2) We present the novel technique of semantic self-verification, which uses concrete instantiations to verify the correctness of the problem formalization. (3) We show how SSV can also improve the formalization itself through instantiation-guided refinement and exploration of multiple candidate formalizations. (4) We present an extensive evaluation on five open benchmarks that shows a significant increase in overall accuracy over SoTA, as well as near-perfect selective accuracy over a significant coverage of verified cases.[1]

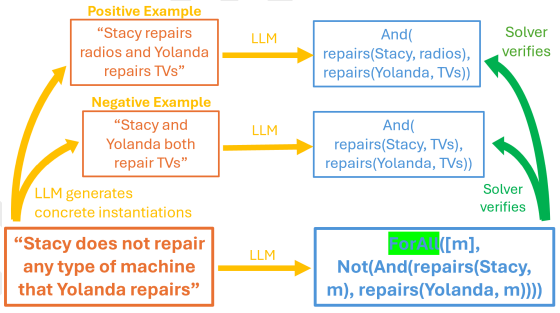
2 Semantic Self-Verification

This section describes the semantic self-verification approach for reasoning problems, which generates programs verified and refined by concrete instantiations. Figure 5 presents the

[1]code & data available at http://github.com/mohammadraza4/ssv



(a) Incorrect formalization (uses Exists quantifier)



(b) Correct formalization (uses ForAll quantifier)

Figure 4: Semantic self-verification of a general constraint: the negative example fails for the wrong formalization (a), while both instantiations are verified for the correct formalization (b)

main algorithm, illustrating the top-level flow and key components. As formulated, the algorithm takes a question (`Q`), such as the technicians problem in Figure 1, and outputs an answer along with an indication of verification success. Figure 5 also details the algorithm's configuration parameters: the chosen LLM and solver, LLM temperature values, and the maximum repair attempts. We first outline the general algorithm before discussing its key phases in detail.

For each temperature value to be explored, the algorithm first uses the LLM to infer a program P that the solver executes to answer the question `Q`, such as the program from Figure 2. If an executable program is generated ($P \neq \varnothing$), the verification loop begins (line 4). The solver first executes P to obtain an answer. Then, for verification, we infer concrete instantiations $\mathcal{I}$, which are test cases for the program's constraints and options, such as the six constraints and five options in Figure 2. The solver attempts to verify that each instantiation is formally satisfiable and returns any failing instantiation I_{fail}. For example, for the third constraint in the technicians program, inferred instantiations (Figure 4a) may yield the failing case: "Stacy and Yolanda cannot both repair TVs." If no failing instantiation is found (as in Figure 4b) and P satisfies general well-formedness properties, the algorithm returns its answer A along with verification success (line 12).

If verification fails, we attempt to repair the program P using the LLM and any failing instantiation, which provides insight into potential constraint implementation errors. For example, the failing instantiation in Figure 4a may guide the

LLM to assert the condition for all machine types using the forall quantifier, as shown in Figure 4b. After obtaining the repaired program, we repeat the verification loop. If no answer is verified across all temperatures and repair attempts, we exit the outer loop (line 16). If no executable program was inferred, we fall back to direct inference using the LLM with a chain-of-thought prompt, as in prior work [Pan *et al.*, 2023]. Otherwise, we return the best answer with verification failure. We next discuss key algorithm phases in more detail.

Program generation. The `GenProgram` function in Figure 5 uses the LLM to generate a solver-executable program for the given problem. A basic implementation relies on a direct LLM prompt, but we incorporate techniques from the code generation literature to improve quality. First, we use error-based refinement: syntax or execution errors in the generated program are fed back to the LLM for repair, a common approach in LLM-based code generation/reasoning domains [Chen *et al.*, 2024; Pan *et al.*, 2023]. Second, if direct code generation fails, we employ a compositional approach [Khot *et al.*, 2023; Pourreza and Rafiei, 2024], generating the program incrementally for each identified constraint. This improves code quality compared to direct prompting, which often produces syntax errors.

Semantic verification. While code generation ensures an executable solver program, it does not address *semantic* correctness—whether the program accurately implements the problem's intended constraints. SSV addresses this by generating and verifying concrete instantiations for each constraint in the generated program. The `GenInstantiations` function first parses the program P to extract constraints and their NL descriptions. Our program generation phase structures programs in segments of the form $P_{init} + C_1 + ... + C_N + O_1 + ... + O_M$, where P_{init} contains initial definitions, followed by explicitly segmented constraints and options, each annotated with NL comments (e.g. see "#CONSTRAINT:" and "#OPTION:" segments in Figure 2). This structure allows parsing constraints along with their NL descriptions.

We use the LLM to infer concrete instantiations for each of the constraints, using their NL descriptions. For each constraint C_i, our implementation prompts the LLM for one positive and one negative instantiation, and both instantiations are translated into solver expressions (Figure 4). Once all instantiations $\mathcal{I}$ are obtained, the `Verify` function uses the solver to check if each constraint is consistent with its respective instantiations. For each constraint C_i, we it verifies its positive instantiation I_p by constructing and executing the expression $P_{init} + C_i + I_p$ and checking that the solver returns SAT. For the negative instantiation I_n, it checks that the expression $P_{init} + C_i + I_n$ is UNSAT. If this holds for all constraints, the full program is considered verified. If verification fails, it returns the first failing instantiation $I_{\text{fail}} \in \mathcal{I}$.

Beyond verifying concrete instantiations, we also check general logical well-formedness properties using the `IsWellFormed` function, which ensures (1) the program follows the specified structure, (2) it returns a single answer, and (3) it avoids degenerate expressions—tautologies or vacuous implications that introduce redundancies or oversimplifications in the problem formalization.

Semantic program repair. If verification fails and a failing instantiation I_{fail} is found, the `RepairProgram` function attempts to repair the original program P, provided no answer has been found. Unlike error-based program repair, this is a *semantic* repair based on an instantiation inferred by the LLM rather than an execution error. In our repair prompt, we supply the initial definitions code, the constraint code with its NL description, and the failing instantiation expression. The LLM is prompted to first do a chain-of-thought analysis to infer whether the error lies in the initial definitions, the constraint code, or the instantiation itself, before inferring the corrected code. The prompts used for code generation/refinement, instantiation generation and semantic repair are available in [Raza and Milic-Frayling, 2025].

Require: `Q` *// the question*
Require: `LLM` *// the language model*
Require: `Solver` *// the logical solver*
Require: `Temperatures` *// LLM temperatures to try*
Require: `MaxRepairs` *// maximum repair attempts*
1: $A_{\text{best}} \leftarrow \varnothing$
2: **for each** $T \in$ `Temperatures` **do**
3: $P \leftarrow$ **GenProgram**(`LLM`, T, `Solver`, `Q`)
4: **while** $P \neq \varnothing$ **and under** `MaxRepairs` **do**
5: $A \leftarrow$ **ExecuteProgram**(`Solver`, P)
6: **if** $A_{\text{best}} = \varnothing$ **then**
7: $A_{\text{best}} \leftarrow A$
8: **end if**
9: $\mathcal{I} \leftarrow$ **GenInstantiations**(`LLM`, T, P)
10: $I_{\text{fail}} \leftarrow$ **Verify**(`Solver`, $\mathcal{I}$, P)
11: **if** $I_{\text{fail}} = \varnothing$ **and IsWellFormed**(P) **then**
12: **return** $(A,$ **True**$)$
13: **end if**
14: **if** $A = \varnothing$ **then**
15: $P \leftarrow$ **RepairProgram**(`LLM`, T, `Q`, P, I_{fail})
16: **end if**
17: **end while**
18: **end for**
19: **if** $A_{\text{best}} = \varnothing$ **then**
20: $A_{\text{best}} \leftarrow$ **InferLLMAnswer**(**LLM**, `Q`)
21: **end if**
22: **return** $(A_{\text{best}},$ **False**$)$

Figure 5: The Semantic Self-Verification Algorithm

3 Evaluation

We evaluate our SSV technique on open benchmarks for logical reasoning, focusing on two key aspects: (1) improving the general accuracy of reasoning over existing baselines and (2) assessing verification quality in terms of both precision (correctness) and coverage (proportion of verified cases).

Datasets. We use five common datasets for logical reasoning. All datasets follow a multiple-choice format, where each task includes a problem statement, a question, and answer options (e.g., Figure 1). **PrOntoQA** is a synthetic deductive reasoning dataset for LLM evaluation [Saparov and He, 2023]. We use its most challenging subset—fictional character tasks requiring 5 reasoning hops—comprising 500 test examples with 2 answer options (True/False). **ProofWriter** is a widely

Dataset	General Accuracy				SSV Verification	
	Standard	CoT	Logic-LM	SSV	Coverage	Precision
AR-LSAT	33.3	35.1	43.0	**71.3**	21.7	94.0 (100.0)
FOLIO	69.1	70.6	78.9	**80.9**	25.0	98.0 (100.0)
LogicalDeduction	71.3	75.3	87.6	**89.7**	43.7	100.0
PrOntoQA	77.4	98.8	83.2	**100.0**	66.0	100.0
ProofWriter	52.7	68.1	79.7	**98.0**	75.2	98.7 (100.0)

Table 1: General accuracy and SSV precision/coverage with GPT-4 base model. *Values in brackets are actual values on corrected datasets.*

used logical reasoning dataset [Tafjord *et al.*, 2021]. We use its open-world assumption subset with 5-hop reasoning tasks, following [Pan *et al.*, 2023], with 600 test examples and 3 answer options (True/False/Unknown). **FOLIO** is an expert-crafted dataset for logical reasoning [Han *et al.*, 2022], featuring real-world knowledge problems phrased in natural language and requiring complex first-order logic. We evaluate on its full test set of 204 examples, each with 3 answer options (True/False/Unknown). **LogDeduction** is a dataset from the BigBench benchmark [Srivastava *et al.*, 2023] involving object sequence ordering based on given conditions. The full test set contains 300 tasks with 3, 5, or 7 answer options. **AR-LSAT** consists of analytical reasoning questions from LSAT exams from 1991–2016 [Zhong *et al.*, 2022]. This challenging dataset has seen only marginally better-than-random accuracy from existing approaches [Pan *et al.*, 2023; Liang *et al.*, 2023]. The test set has 230 questions, each with 5 answer options.

Baselines. We compare our technique against three baselines, which represent approaches of reasoning using the LLM alone, as well as the combination of formal logical solvers with LLMs. Each of these baselines and our own system is parametric in the LLM used, and in our experiments we investigate all systems with both the GPT-4 model (a current best general LLM for reasoning) as well as the weaker GPT-3.5 model from Open AI. We use the baselines and their results for these models as reported in [Pan *et al.*, 2023]. The baselines are as follows. **Standard** is the direct approach of prompting the LLM, leveraging in-context learning to answer the question. **CoT** (Chain-of-Thought) [Wei *et al.*, 2022] follows a step-by-step reasoning process, generating explanations before the final answer. **Logic-LM** is a state-of-the-art method that integrates LLMs with solvers for formal reasoning [Pan *et al.*, 2023], where the LLM is prompted to generate a solver program to solve the task. **SSV** is our semantic self-verification technique (Figure 5). Our implementation uses the Z3 SMT solver [de Moura and Bjørner, 2008] and applies identical prompts for both models, with 1-4 few-shot examples drawn from training datasets (detailed in the Appendices). Our full SSV implementation sets `MaxRepairs` $= 2$ and `Temperatures` $= [0, 0.3, 0.4, 0.5]$ (covering low to mid-range values), with parameter variations explored in the ablation analysis.

3.1 Results

Main results Table 1 presents the main results, with all systems evaluated using GPT-4 as the underlying LLM. The table reports general accuracy as well as the precision and coverage of SSV verification. General accuracy represents the percentage of correct answers across the dataset. For SSV, precision denotes the percentage of correct answers among those flagged as verified, while coverage indicates the percentage of verified cases relative to the entire dataset. The key observations are as follows:

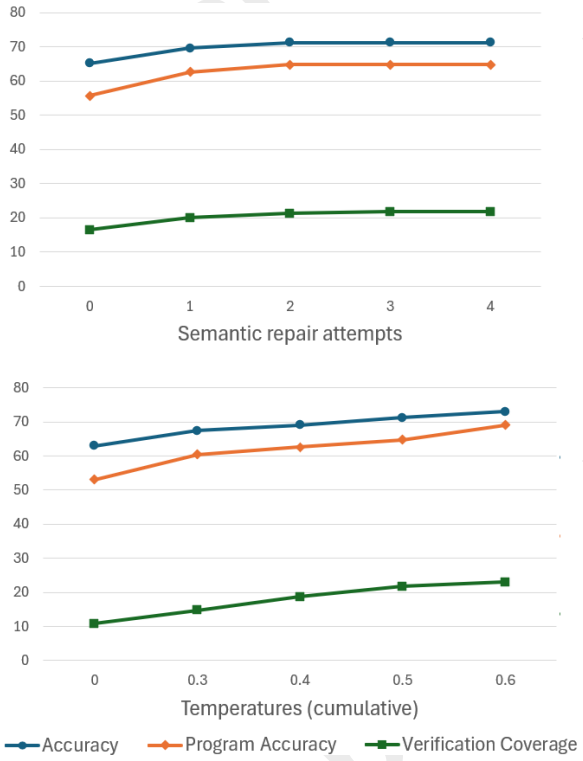


Figure 6: Repair attempts and temperature variations on AR-LSAT

1. SSV outperforms all baselines in general accuracy. Our technique achieves a higher general accuracy over all baseline systems across all datasets. We especially note the drastic increase of 28.3% over the current best Logic-LM system on the most difficult AR-LSAT dataset. This shows the strong effectiveness of our technique in producing robust problem formalizations in contrast to just a direct LLM translation from the natural language description to the solver program.

2. SSV verification has perfect precision across all datasets. With GPT-4 as base model, SSV achieves 100% verification precision on all datasets. Notably, on AR-LSAT, FOLIO,

and ProofWriter, our verification mechanism identified erroneous cases where the datasets contained incorrect answers. However, for comparison with baselines, in Table 1 we also report results based on the original datasets (showing slightly lower precision due to mislabelled cases). See [Raza and Milic-Frayling, 2025] for details of corrections. For AR-LSAT cases we also verified our corrections against the original test answers[2]. This empirically perfect precision highlights SSV's robustness on complex reasoning tasks.

3. SSV verification has significant coverage on all datasets. Although the precision is very high, we know that SSV verification does not always succeed. However, we find that the coverage is significant across all datasets, with the lowest coverage of 21.7% on the most difficult AR-LSAT dataset. As expected, we find the coverage increases on the relatively easier datasets, with a verification coverage of up to 75.2% on ProofWriter. This significant coverage of verification shows that the SSV approach can help in avoiding manual human verification in a significant proportion of cases to reduce overall cost and effort.

Effect of semantic repair and temperature exploration. Figure 6 shows the impact of varying semantic repair attempts (`MaxRepairs`) and temperatures (`Temperatures`) on the AR-LSAT dataset. We analyze overall accuracy, program accuracy (how often program generation succeeds rather than direct LLM answers), and verification coverage. Semantic repair improves accuracy by 6.1%, while temperature exploration increases it by 10.0%. Verification coverage gains 5.2% with repair and more than doubles with temperature exploration, rising 12.2% above an initial 10.9%. Repair attempts yield diminishing returns and cease to improve any metric beyond three attempts, while temperature exploration continues to show some gains up to 0.6. Additionally, the gap between program accuracy and overall accuracy narrows (from 9.8% to 5.2%, when averaged over both temperature and repair attempts), indicating greater reliance on program generation with these enhancements.

We also ran a full ablation on AR-LSAT without any repair or temperature sampling (effectively replicating Logic-LM but using compositional code generation). This scored 55.7% vs. our 71.3% (Logic-LM: 43%), showing our novel features add 15.6%, and other enhancements contribute 12.7%.

Evaluation on GPT-3.5. We also evaluated our system and all baselines using GPT-3.5 as the underlying LLM. The results are shown in Table 2. Firstly, we note that while the general accuracy of all systems drops significantly with this weaker model, our SSV system still performs best overall, with an average accuracy of 56.2%. However, Logic-LM performs better than SSV on FOLIO and LogicalDeduction (this could be partly due to differences in the code generation quality for the different solver languages that Logic-LM uses for these datasets). Secondly, we observe that while the coverage of SSV verification also drops significantly, with two of the more difficult datasets (AR-LSAT and LogicalDeduction) having no coverage at all, the precision of SSV is very minimally affected. On the three datasets where there is coverage, we still see an average precision of 97%. This demonstrates an important property of reliability of SSV verification: even for weaker models, if verification succeeds then it is still very reliable (and much more reliable than general accuracy), though it may succeed much less often. In practical terms, such reliability could even allow one to adopt a tiered strategy to optimize costs: trying weaker (cheaper) models for tasks first and fall-back on more expensive models if verification fails.

Verification failures We conducted a manual analysis on a sample of cases where verification did not pass. Classification of key reasons: program not well-formed (13.3%), program incorrect (53.3%), example incorrect (10%), both incorrect (23.3%). Thus in most cases the program was incorrect, which aligns with the expectation that examples inference is generally simpler than abstract program formulation.

4 Limitations and Future Directions

Since natural language is informal, any verification approach with NL specifications cannot guarantee full correctness. While SSV verification achieves near-perfect empirical precision (100% with GPT-4), we discuss the kinds of errors illustrated by some cases of incorrect verification observed with GPT-3.5 (specifically, one case in PrOntoQA and four in ProofWriter where incorrect answers passed verification).

1. *Concrete instantiations are insufficient*. Since verification relies on concrete examples (test cases), these may not cover all aspects of a general constraint, particularly corner cases. This caused two failures with GPT-3.5. For instance, in one case, the conditions "Gary is nice" and "Gary is kind" were conflated into a single predicate "is_kind(Gary)" in the formalization. An instantiation asserting "Gary is nice but not kind" could have detected this error.

2. *Concrete instantiation and program are both mutually consistent but wrong*. This is the unlikely case where both the program and the test case have the same error and therefore pass verification. We found only one such case which was a rather confusingly trivial error: for some reason the constraint "Fiona is quiet" was translated as its negation "Not(is_quiet(Fiona))" in both the program and the concrete instantiation independently generated by GPT-3.5.

3. *Missing or superfluous constraints*. The LLM may omit required constraints or introduce unintended ones. Since our approach relies on explicitly demarcated constraints parsed from the LLM-generated program, such errors can cause verification failures. Two GPT-3.5 failures resulted from superfluous constraints.

In general, such errors are rare, more common in weaker LLMs, and expected to decrease as LLMs improve. Errors of types (1) and (2) could be mitigated with a more exhaustive examples inference strategy, as our implementation generates only one positive and one negative example per constraint. Class (3) errors arise from structural inconsistencies where program constraints do not match the original problem. Such cases may be addressed by training specialized modules to more robustly enforce core structural properties.

Another potential limitation is that while industrial provers like Z3 are effectively decidable for many practical problems (we observed no failures due to the solver), in more complex

[2]https://img.cracklsat.net/lsat/pt/pt80.pdf

Dataset	General Accuracy				SSV Verification	
	Standard	CoT	Logic-LM	SSV	Coverage	Precision
AR-LSAT	20.3	17.3	26.4	**28.3**	0	-
FOLIO	45.1	57.4	**62.7**	59.3	1.5	100.0
LogicalDeduction	40.0	42.3	**65.7**	48.3	0	-
PrOntoQA	47.4	67.8	61.0	**72.8**	4.2	95.2
ProofWriter	35.5	49.2	58.3	**72.5**	16.2	94.8 (95.9)

Table 2: General accuracy and SSV precision/coverage with GPT-3.5 base model. *Values in brackets are actual values on corrected datasets.*

cases our method will conservatively fail verification, as decidability of first-order logic is undecidable in general. Future work may also explore addressing this limitation using iterative LLM reasoning to assist solver convergence.

5 Related Work

Reasoning with LLMs. Improving the robustness of reasoning in large language models is a very active area of research. One direction of work is to fine-tune or train specialized models that show improved reasoning ability [Tafjord *et al.*, 2022; Clark *et al.*, 2020; Yang *et al.*, 2022]. Another direction is to develop sophisticated prompting strategies to elicit better reasoning from LLMs. Chain-of-thought prompting [Wei *et al.*, 2022] has shown how the quality of reasoning can be improved by prompting the model to explicitly generate the steps of reasoning in natural language before arriving at the final answer. Other examples of prompting approaches include self-consistency [Wang *et al.*, 2023], analogical reasoning [Yu *et al.*, 2024], and various modular approaches to address complex problems by decomposition to simpler sub-problems [Zhou *et al.*, 2023; Khot *et al.*, 2023; Creswell *et al.*, 2023]. While these approaches show relative improvements in accuracy, the reasoning is still based on informal natural language and is prone to errors in the reasoning steps. In contrast, we follow the approach of off-loading the reasoning task to a formal solver that can guarantee correctness of the reasoning steps. Our particular focus is on the key challenge of ensuring correct formalization of the problem.

Tool-augmented reasoning. Integrating LLMs with specialized tools for performing various tasks is becoming increasingly common [Schick *et al.*, 2023]. This approach has also been adopted to improve the reasoning quality by augmenting the LLM with logical solvers or automated reasoning tools [Pan *et al.*, 2023; Ye *et al.*, 2023; Nye *et al.*, 2021]. The key challenge with these approaches is to ensure that the LLM correctly translates the reasoning problem from NL to the formal language of the solver. This is the main focus of our work, where we show how verification and refinement with respect to concrete instantiations generated by the LLM can both improve accuracy and also provide verification with near-perfect precision. [Kalyanpur *et al.*, 2024] also infer logic programs with test cases, but their test cases are arbitrary logical expressions inferred together with the program, and thus prone to similar errors the LLM may make in the program. In contrast, we generate concrete instantiations (literal assignments) independently from the program constraints, which the LLM can infer from the NL without any logical formulation. This yields very high precision verification which we can offer as a standalone feature, unlike any prior work. Tool-augmented approaches have also been explored in the related areas of planning [Kambhampati *et al.*, 2024; Guan *et al.*, 2024] and auto-formalization [Wu *et al.*, 2022; Jiang *et al.*, 2023; He-Yueya *et al.*, 2023], where informal mathematical proofs are translated to formal specifications defined in theorem provers like Isabelle [Paulson, 1994] and Lean [de Moura *et al.*, 2015]. While our work focuses on logical reasoning, the principle of consistency-based verificaion and refinement of formalizations using concrete instantiations is also potentially applicable to these other domains.

Self-verification approaches. Many related works have also explored the notion of self-verification by LLMs [Weng *et al.*, 2023; Madaan *et al.*, 2023; Xie *et al.*, 2023; Ling *et al.*, 2023; Miao *et al.*, 2024]. The general idea is that using the LLM to inspect and verify its own reasoning can show improvements, though in some domains self-critiquing has also shown diminished performance [Valmeekam *et al.*, 2023]. Our approach of verification is different: instead of asking the LLM to verify the abstract chain of reasoning, we only ask it to generate concrete examples of the general constraints in the problem. The task of verification is then done with the solver to formally check that the examples are consistent with the abstract formalization. Thus apart from not relying purely on the LLM for verification, we also avoid the more complex task of verifying an abstract chain of reasoning which can itself be highly error-prone. We show how this approach provides a very high precision verification, as opposed to just relative improvements in accuracy.

6 Conclusion

We have presented the Semantic Self-Verification approach, which infers strong problem formalizations based on concrete instantiations, using a consistency-based verification paradigm that leverages LLMs and logical solvers. Beyond achieving state-of-the-art accuracy, SSV introduces a novel verification feature that has near-perfect empirical precision. As the reasoning power of LLMs continues to advance, such near-certain verification can serve as a complementary dimension to general accuracy gains in order to ensure confidence on arbitrarily complex tasks.

Acknowledgments. The first author is grateful for the discussions with his daughter to help with her middle school studies, which provided the inspiration for this work.

References

[Chen *et al.*, 2024] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *ICLR*. OpenReview.net, 2024.

[Clark *et al.*, 2020] Peter Clark, Oyvind Tafjord, and Kyle Richardson. Transformers as soft reasoners over language. In Christian Bessiere, editor, *IJCAI*, pages 3882–3890. ijcai.org, 2020. Scheduled for July 2020, Yokohama, Japan, postponed due to the Corona pandemic.

[Creswell *et al.*, 2023] Antonia Creswell, Murray Shanahan, and Irina Higgins. Selection-inference: Exploiting large language models for interpretable logical reasoning. In *ICLR*. OpenReview.net, 2023.

[de Moura and Bjørner, 2008] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[de Moura *et al.*, 2015] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *CADE*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015.

[Guan *et al.*, 2024] Lin Guan, Karthik Valmeekam, Sarath Sreedharan, and Subbarao Kambhampati. Leveraging pretrained large language models to construct and utilize world models for model-based task planning. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2024. Curran Associates Inc.

[Han *et al.*, 2022] Simeng Han, Hailey Schoelkopf, Yilun Zhao, Zhenting Qi, Martin Riddell, Luke Benson, Lucy Sun, Ekaterina Zubova, Yujie Qiao, Matthew Burtell, David Peng, Jonathan Fan, Yixin Liu, Brian Wong, Malcolm Sailor, Ansong Ni, Linyong Nan, Jungo Kasai, Tao Yu, Rui Zhang, Shafiq R. Joty, Alexander R. Fabbri, Wojciech Kryscinski, Xi Victoria Lin, Caiming Xiong, and Dragomir Radev. Folio: Natural language reasoning with first-order logic. *CoRR*, abs/2209.00840, 2022.

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

[Jiang *et al.*, 2023] Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothée Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *ICLR*. OpenReview.net, 2023.

[Kalyanpur *et al.*, 2024] Aditya Kalyanpur, Kailash Saravanakumar, Victor Barres, Jennifer Chu-Carroll, David Melville, and David A. Ferrucci. Llm-arc: Enhancing llms with an automated reasoning critic. *CoRR*, abs/2406.17663, 2024.

[Kambhampati *et al.*, 2024] Subbarao Kambhampati, Karthik Valmeekam, Lin Guan, Mudit Verma, Kaya Stechly, Siddhant Bhambri, Lucas Paul Saldyt, and Anil B Murthy. Position: LLMs can't plan, but can help planning in LLM-modulo frameworks. In *Forty-first International Conference on Machine Learning*, 2024.

[Khot *et al.*, 2023] Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. In *ICLR*. OpenReview.net, 2023.

[Liang *et al.*, 2023] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Cosgrove, Christopher D. Manning, Christopher Ré, Diana Acosta-Navas, Drew A. Hudson, Eric Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue Wang, Keshav Santhanam, Laurel J. Orr, Lucia Zheng, Mert Yüksekgönül, Mirac Suzgun, Nathan Kim, Neel Guha, Niladri S. Chatterji, Omar Khattab, Peter Henderson, Qian Huang, Ryan Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang, and Yuta Koreeda. Holistic evaluation of language models. *Trans. Mach. Learn. Res.*, 2023, 2023.

[Ling *et al.*, 2023] Zhan Ling, Yunhao Fang, Xuanlin Li, Zhiao Huang, Mingu Lee, Roland Memisevic, and Hao Su. Deductive verification of chain-of-thought reasoning. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *NeurIPS*, 2023.

[Madaan *et al.*, 2023] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[Miao *et al.*, 2024] Ning Miao, Yee Whye Teh, and Tom Rainforth. Selfcheck: Using LLMs to zero-shot check their own step-by-step reasoning. In *The Twelfth International Conference on Learning Representations*, 2024.

[Nye *et al.*, 2021] Maxwell I. Nye, Michael Henry Tessler, Joshua B. Tenenbaum, and Brenden M. Lake. Improving coherence and consistency in neural sequence models with dual-system, neuro-symbolic reasoning. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *NeurIPS*, pages 25192–25204, 2021.

[Pan *et al.*, 2023] Liangming Pan, Alon Albalak, Xinyi Wang, and William Yang Wang. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. In Houda Bouamor, Juan Pino, and Ka-

lika Bali, editors, *EMNLP (Findings)*, pages 3806–3824. Association for Computational Linguistics, 2023.

[Paulson, 1994] Lawrence C Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.

[Pourreza and Rafiei, 2024] Mohammadreza Pourreza and Davood Rafiei. Din-sql: decomposed in-context learning of text-to-sql with self-correction. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2024. Curran Associates Inc.

[Raza and Milic-Frayling, 2025] Mohammad Raza and Natasa Milic-Frayling. Instantiation-based formalization of logical reasoning tasks using language models and logical solvers (full version). *CoRR*, abs/2501.16961, January 2025.

[Saparov and He, 2023] Abulhair Saparov and He He. Language models are greedy reasoners: A systematic formal analysis of chain-of-thought. In *ICLR*. OpenReview.net, 2023.

[Schick *et al.*, 2023] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, M. Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2 2023.

[Srivastava *et al.*, 2023] Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *Trans. Mach. Learn. Res.*, 2023, 2023.

[Tafjord *et al.*, 2021] Oyvind Tafjord, Bhavana Dalvi, and Peter Clark. Proofwriter: Generating implications, proofs, and abductive statements over natural language. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *ACL/IJCNLP (Findings)*, volume ACL/IJCNLP 2021 of *Findings of ACL*, pages 3621–3634. Association for Computational Linguistics, 2021.

[Tafjord *et al.*, 2022] Oyvind Tafjord, Bhavana Dalvi Mishra, and Peter Clark. Entailer: Answering questions with faithful and truthful chains of reasoning. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *EMNLP*, pages 2078–2093. Association for Computational Linguistics, 2022.

[Valmeekam *et al.*, 2023] Karthik Valmeekam, Matthew Marquez, and Subbarao Kambhampati. Can large language models really improve by self-critiquing their own plans? In *NeurIPS 2023 Foundation Models for Decision Making Workshop*, 2023.

[Wang *et al.*, 2023] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *ICLR*. OpenReview.net, 2023.

[Wei *et al.*, 2022] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *NeurIPS*, 2022.

[Weng *et al.*, 2023] Yixuan Weng, Minjun Zhu, Fei Xia, Bin Li, Shizhu He, Shengping Liu, Bin Sun, Kang Liu, and Jun Zhao. Large language models are better reasoners with self-verification. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *EMNLP (Findings)*, pages 2550–2575. Association for Computational Linguistics, 2023.

[Wu *et al.*, 2022] Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *NeurIPS*, 2022.

[Xie *et al.*, 2023] Yuxi Xie, Kenji Kawaguchi, Yiran Zhao, Xu Zhao, Min-Yen Kan, Junxian He, and Qizhe Xie. Self-evaluation guided beam search for reasoning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[Yang *et al.*, 2022] Kaiyu Yang, Jia Deng, and Danqi Chen. Generating natural language proofs with verifier-guided search. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *EMNLP*, pages 89–105. Association for Computational Linguistics, 2022.

[Ye *et al.*, 2023] Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. SatLM: Satisfiability-aided language models using declarative prompting. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[Yu *et al.*, 2024] Junchi Yu, Ran He, and Zhitao Ying. Thought propagation: an analogical approach to complex reasoning with large language models. In *ICLR*. OpenReview.net, 2024.

[Zhong *et al.*, 2022] Wanjun Zhong, Siyuan Wang, Duyu Tang, Zenan Xu, Daya Guo, Yining Chen, Jiahai Wang, Jian Yin, Ming Zhou, and Nan Duan. Analytical reasoning of text. In Marine Carpuat, Marie-Catherine de Marneffe, and Ivan Vladimir Meza Ruiz, editors, *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 2306–2319, Seattle, United States, July 2022. Association for Computational Linguistics.

[Zhou *et al.*, 2023] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V. Le, and Ed H. Chi. Least-to-most prompting enables complex reasoning in large language models. In *ICLR*. OpenReview.net, 2023.

[Zhou, 2012] Zhi-Hua Zhou. *Ensemble Methods: Foundations and Algorithms*. Chapman & Hall/CRC, 1st edition, 2012.