PROOF OF THOUGHT : Neurosymbolic Program Synthesis allows Robust and Interpretable Reasoning

Debargha Ganguly*
Case Western Reserve University
debargha@case.edu

Srinivasan Iyengar
Microsoft Corporation
sriyengar@microsoft.com

Vipin Chaudhary
Case Western Reserve University
vipin@case.edu

Shivkumar Kalyanaraman
Microsoft Corporation
shkalya@microsoft.com

Abstract

Large Language Models (LLMs) have revolutionized natural language processing, yet they struggle with inconsistent reasoning, particularly in novel domains and complex logical sequences. This research introduces PROOF OF THOUGHT, a framework that enhances the reliability and transparency of LLM outputs. Our approach bridges LLM-generated ideas with formal logic verification, employing a custom interpreter to convert LLM outputs into First Order Logic constructs for theorem prover scrutiny. Central to our method is an intermediary JSON-based Domain-Specific Language, which by design balances precise logical structures with intuitive human concepts. This hybrid representation enables both rigorous validation and accessible human comprehension of LLM reasoning processes. Key contributions include a robust type system with sort management for enhanced logical integrity, explicit representation of rules for clear distinction between factual and inferential knowledge, and a flexible architecture that allows for easy extension to various domain-specific applications. We demonstrate PROOF OF THOUGHT's effectiveness through benchmarking on StrategyQA and a novel multimodal reasoning task, showing improved performance in open-ended scenarios. By providing verifiable and interpretable results, our technique addresses critical needs for AI system accountability and sets a foundation for human-in-the-loop oversight in high-stakes domains.

1 Introduction

Large language models (LLMs) have revolutionized the field of AI and enabled a wide range of applications. However, as these models are increasingly deployed to process unstructured data and perform complex tasks autonomously, their inconsistent reasoning capabilities remain a critical limitation [Marcus, 2020]. This inconsistency manifests in variable performance across out-of-domain reasoning, negation understanding, and extended logical chains, suggesting a reliance on superficial heuristics [Bender et al., 2021]. The implications are far-reaching, particularly in high-stakes domains where reliable and transparent decision-making is crucial [Rudin, 2019]. Errors or biases in these contexts could have severe consequences, underscoring the urgent need for more dependable and interpretable AI systems.

Recent advances in prompt engineering have shown promise in addressing these challenges. Techniques such as Chain-of-Thought (CoT) [Wei et al., 2022], Self-Consistency with CoT (CoT-SC)

*Work performed while at Microsoft Research.

Preprint. Under review.

[Wang et al., 2022], Tree of Thoughts (ToT) [Yao et al., 2024], and Graph of Thoughts (GoT) [Besta et al., 2024] have improved problem-solving capabilities. In the multimodal domain, techniques like Set-of-Marks (SoM) prompting have emerged [Yang et al., 2023]. Despite advancements in performance figures, the mechanisms behind these improvements remain opaque, causing blind spots in real-world usage as failure modes are not well understood. The fundamental issue lies in the lack of interpretability and guaranteed verifiability in LLM reasoning processes [Danilevsky et al., 2020]. This opacity hinders our ability to trust and validate LLM outputs, a critical concern in scenarios requiring explainable AI or human-in-the-loop oversight.

Real-world applications, especially in health and safety domains, face additional challenges in training and deploying models due to the scarcity of high-quality annotated data. This is particularly evident in sectors like energy, healthcare, and manufacturing, where ML has significant potential for operational efficiency [Jordan and Mitchell, 2015]. Complex tasks, such as identifying OSHA violations in cluttered visual data, illustrate the difficulties in long-tail, low-data scenarios. These scenarios are characterized by diverse and unpredictable phenomena, where specialized model training is impractical [Zhang et al., 2021]. With robust reasoning, LLMs' wide knowledge and commonsense abilities can potentially operate in these low-data paradigms, opening up more applications.

Contributions: To address these challenges, we :

1. Propose PROOF OF THOUGHT (PoT), a novel approach that leverages the in-context learning and code generation capabilities of LLMs while incorporating their inherent knowledge and spatial understanding. Our system employs a custom interpreter that parses "LLM-Thoughts" (represented as DSL code snippets) to generate First Order Logic programs, which are then verified by a Z3 theorem prover.
2. Introduce an intermediate JSON-based DSL (neurosymbolic representation) and the associated interpreter that operates on human-understandable abstract concepts using intuitive, near-English language constructs (see Fig 1). This strikes a balance between the precision required for logical definitions with formal first-order logic proofs and accessibility for non-expert users.
3. Benchmark performance over StrategyQA (a boolean multi-hop implicit NLP reasoning benchmark) and a novel multimodal real-world long-tail reasoning problem (Reddit-OSHA Benchmark). This shows that PoT works on complex, and a wide variety of tasks.

PROOF OF THOUGHT enhances the capabilities of LLMs in complex, open-ended scenarios by providing reasoning guarantees, conditioned on correctness of knowledge base and rule specifications, therefore furnishing a framework for human-in-the-loop oversight and verification.

2 Related Work

Early Integration Attempts laid the groundwork for neuro-symbolic AI. Works like EBL-ANN [Towell and Shavlik, 1994], KBANN [Towell et al., 1990], and C-ILP [d'Avila Garcez et al., 2009] incorporated propositional formulae into neural networks. While pioneering, these approaches struggled with scalability and expressiveness. Knowledge Graph integration advanced the field further. Methods proposed by Chen et al. [2020a] and Kampffmeyer et al. [2019] showed promise in leveraging structured knowledge for improved reasoning. However, maintaining interpretability and explicit rule incorporation remained challenging. Differentiable Logic Programming frameworks like DeepProbLog [Manhaeve et al., 2018] and Scallop [Li et al., 2023] demonstrated the potential of integrating probabilistic logic programming with neural networks. These approaches enable end-to-end training of neuro-symbolic systems but face limitations in handling complex reasoning tasks and diverse logical formalisms. Gupta and Kembhavi [2023] showed how compositional visual reasoning can be done by program generation without training.

Large Language Models have opened new avenues for neuro-symbolic reasoning. Techniques such as Chain-of-Thought [Wei et al., 2022], Tree-of-Thoughts [Yao et al., 2024], and Graph-of-Thoughts [Besta et al., 2024] have shown impressive results in complex reasoning tasks. However, these methods often produce inconsistent intermediate steps and struggle with out-of-domain reasoning. Interpretable Concept-based Models, as explored by Kim et al. [2018] and Chen et al. [2020b], aim to increase trust in deep learning models. However, state-of-the-art approaches often rely on high-dimensional concept embeddings that lack clear semantic meaning, limiting their interpretability.

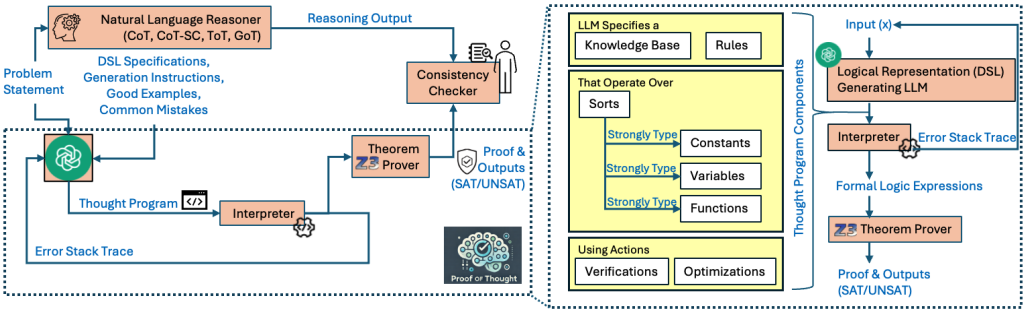

Figure 1: Architecture of the PROOF OF THOUGHT (PoT) framework, illustrating the integration of natural language reasoning with formal logical verification.

Advanced Neuro-symbolic Frameworks like the Deep Concept Reasoner (DCR) [Barbiero et al., 2023] and GENOME [Chen et al., 2023] have made progress in combining neural and symbolic components. DCR constructs syntactic rule structures from concept embeddings, while GENOME introduces a modular approach for visual reasoning tasks. Despite these advancements, challenges in scalability and generalization to diverse task domains persist. A critical issue is reasoning shortcuts, where models use unintended concept semantics. To address this, Marconato et al. [2024] introduced BEARS, improving model calibration and informative annotation acquisition. These developments showcase the potential for imparting robustness and reliability via neuro-symbolic AI systems.

LLM Code Generation and Low-Resource Translation demonstrate the remarkable adaptability of large language models. In code generation, LLMs like Codex [Chen et al., 2021] and AlphaCode [Li et al., 2022] show proficiency across multiple programming languages, often outperforming specialized models. These systems excel at understanding context, generating syntactically correct code, and even solving complex algorithmic problems. Paralleling this, research in low-resource language translation [Zoph et al., 2016, Tanzer et al., 2023] reveals LLMs' ability to rapidly adapt to new languages with minimal examples. Techniques like few-shot learning and cross-lingual transfer enable models to leverage knowledge from high-resource languages to improve performance on low-resource ones. Both domains highlight LLMs' capacity for quick adaptation and generalization, suggesting potential for enhanced neuro-symbolic systems that can efficiently learn and apply formal reasoning across diverse domains with limited training data.

3 PROOF OF THOUGHT: A Neurosymbolic Reasoning Framework

In this section, we introduce PROOF OF THOUGHT (PoT), a novel framework that bridges NLP with formal logical reasoning to enhance the interpretability and verifiability of LLM outputs. We first outline the foundational concepts and notations that underpin our approach.

3.1 Background Concepts & Notation from LLM Reasoning Literature

Let p_θ denote a pre-trained language model (LM) with parameters θ. In a conversation, user messages (prompts) and LM replies (thoughts) are exchanged. We use lowercase letters $x, y, z, ...$ to indicate LM thoughts, where the definition of a "thought" is use-case specific (e.g., a paragraph, document, or code block). The simplest Prompting Approach approach is Input-Output (IO) where an LM directly transforms an input sequence x into output y without intermediate steps. Chain-of-Thought (CoT) introduces intermediate thoughts $a_1, a_2, ...$ between x and y, enhancing performance on tasks like mathematical reasoning.

Multiple CoTs generalizes CoT by generating k independent chains and selecting the best output based on a prescribed scoring metric. This approach, introduced as Self-Consistency with CoT (CoT-SC), allows exploration of different reasoning paths. Tree of Thoughts (ToT) further enhances CoT-SC by modeling reasoning as a tree of thoughts. Each node represents a partial solution. For a given node, k new nodes are generated, then evaluated using an LM or human scores. The tree expansion is guided by search algorithms like BFS or DFS. Finally, Graph of Thoughts (GoT) extends

ToT by allowing more complex connections between thoughts, forming a directed graph structure. In GoT, thoughts can have multiple predecessors and successors, enabling more flexible reasoning paths and the combination of ideas from different branches. This approach allows for cyclic reasoning patterns and can capture more intricate problem-solving strategies.

3.2 Framework Overview

> "All our knowledge begins with the senses, proceeds then to the understanding, and ends with reason. There is nothing higher than reason."
> - Immanuel Kant, in *Critique of Pure Reason*

PROOF OF THOUGHT models the LLM's reasoning process as a structured transformation from natural language input to formal logical expressions that can be verified using theorem proving techniques. The framework consists of three primary components :

- **Logical Representation Generator** $\mathcal{G}$: Maps input x to a logical representation $\mathcal{L}$ using p_θ.
- **Interpreter** $\mathcal{I}$: Parses $\mathcal{L}$ and constructs formal logical expressions ϕ in first-order logic (FOL).
- **Theorem Prover** $\mathcal{T}$: Verifies the validity of ϕ and provides proofs or counterexamples.

The PoT reasoning process can thus be formalized as:

$$\mathcal{L} = \mathcal{G}(x; p_\theta); \phi = \mathcal{I}(\mathcal{L}); \text{Verification Result} = \mathcal{T}(\phi)$$

Guarantees: By using theorem proving, we rely on the principle of logical consequence, where conclusions are guaranteed to be true if the premises and inference rules, depicted in the logical representations (both of which are human readable, allowing interpretability and verifiability). These logical representations are the ultimate arbiter of truth and validity. The guarantees are what distinguishes PoT from other forms of reasoning. Inductive reasoning (drawing general conclusions from specific observations) can be useful, but doesn't offer the same level of certainty. In contrast, guarantees with theorem proving allow us to establish precise arguments with mathematical truths with absolute confidence, elevating the reasoning process beyond mere conjecture or intuition.

Our PROOF OF THOUGHT framework architecture (shown in Fig 1) introduces a JSON-based Domain Specific Language (DSL) along with the associated interpreter. Next, we discuss the design choices for these in detail.

3.3 Design of the JSON-Based Domain-Specific Language (DSL)

The core of our logical reasoning system is built upon a carefully designed JSON-based Domain-Specific Language (DSL) that serves as an intermediate representation for translating reasoning tasks into formal logic. This DSL was created with the primary challenge of being general-purpose enough to accommodate a wide range of reasoning problems. The choice of JSON as the underlying format was deliberate, leveraging its widespread use, human readability, and ease of parsing. This design decision ensures that the logical representations are both machine-parseable and accessible to users who may not have expertise in formal logic or programming. Moreover, JSON's compatibility with structured outputs from AI service providers like OpenAI and Google, which offer guaranteed outputs matching particular schemas, makes it an ideal choice for our system that doesn't rely on retraining models. The key components of the DSL are:

1. **Sorts** ($\mathcal{S}$) define the domains or types used in the logic. Let $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ be the set of sorts, where each S_i represents a specific domain. The inclusion of sorts in our system is a key differentiator. It allows for reasoning over high-level, human-understandable concepts, which is crucial for bridging the gap between natural language problem descriptions and formal logical representations. The sort definition in Fig 2 allows for more intuitive problem representation. For instance, instead of reasoning about abstract entities, we can now reason about Persons, Equipment, Tasks, Locations, and Time. This makes it easier to translate natural language problems into formal logic. The type-safe reasoning catches semantic errors early. For example, if we tried to apply a function meant for Equipment to a Person, the system would catch this error before any reasoning takes place. Moreover, this

```
"sorts": [
 {"name": "Person", "type": "DeclareSort"},
 {"name": "Equipment", "type": "DeclareSort"},
 {"name": "Task", "type": "DeclareSort"},
 {"name": "Location", "type": "DeclareSort"},
 {"name": "Time", "type": "RealSort"}
]
```

```
"variables": [
{"name": "p", "sort": "Person"},
{"name": "e", "sort": "Equipment"},
{"name": "t", "sort": "Task"},
{"name": "l", "sort": "Location"},
{"name": "time", "sort": "Time"}
]
```

```
"verifications": [
{
"name": "All Tasks Assigned",
"forall": [
{"name": "t", "sort": "Task"}
],
"constraint": "Exists([{'name': 'p', 'sort':
'Person'}], assigned_to(t, p))"
},
{
"name": "No Overqualified Assignments",
"forall": [
{"name": "p", "sort": "Person"},
{"name": "t", "sort": "Task"}
],
"implies": {
"antecedent": "assigned_to(t, p)",
"consequent": "skill_level(p, t) <= 7"
}
}
]
```

```
"functions": [
{"name": "assigned_to", "domain": ["Task",
"Person"], "range": "BoolSort"},
{"name": "location_of", "domain": ["Person"],
"range": "Location"},
{"name": "start_time", "domain": ["Task"],
"range": "Time"},
{"name": "duration", "domain": ["Task"],
"range": "Time"},
{"name": "skill_level", "domain": ["Person",
"Task"], "range": "IntSort"}
]
```

```
"knowledge_base": [
"assigned_to(task1, alice)",
"location_of(alice) == warehouse_A",
"start_time(task1) == 9.0",
"duration(task1) == 2.0",
"skill_level(alice, task1) == 5"
]
```

```
"constants": {
"persons": {
"sort": "Person",
"members": ["alice", "bob", "charlie"]
},
"equipment": {
"sort": "Equipment",
"members": ["forklift", "crane", "truck"]
},
"locations": {
"sort": "Location",
"members": ["warehouse_A",
"construction_site_B", "office_C"]
}
}
```

```
"rules": [
{
"name": "Task Assignment Rule",
"forall": [
{"name": "p", "sort": "Person"},
{"name": "t", "sort": "Task"}
],
"implies": {
"antecedent": "assigned_to(t, p)",
"consequent": "location_of(p) ==
location_of(t)"
}
}
```

```
"optimization": { "variables": [
{"name": "x", "sort": "Person"},
{"name": "y", "sort": "Task"}
],
"constraints": [
"ForAll([x, y], Implies(assigned_to(y, x),
skill_level(x, y) >= 3))",
"ForAll([y], Exists([x], assigned_to(y, x)))"
],
"objectives": [
{
"type": "minimize",
"expression": "Sum([x, y], If(assigned_to(y, x),
skill_level(x, y), 0))"
}]}
```

Figure 2: Example DSL program components of the PROOF OF THOUGHT (PoT) framework for a dummy task assignment verification and optimization problem. The figure displays the JSON-based Domain-Specific Language (DSL) structure, including sort definitions, variables, functions, constants, knowledge base, rules, verifications, and optimization constraints for a workforce management scenario.

structure allows for domain-specific reasoning strategies. For instance, we could implement specialized reasoning algorithms for time-based logic using the Time sort.

2. **Functions** ($\mathcal{F}$) definitions in our system go beyond simple predicates, allowing for rich, typed relationships between sorts. Let $f : S_1 \times S_2 \times \ldots \times S_k \to S_r$ represent a function, where $S_1, S_2, \ldots, S_k, S_r \in \mathcal{S}$. For predicates, $S_r = \text{Bool}$.

   For example, in Fig 2, the function definition allows for complex domain modeling. For instance, ‘assigned_to‘ represents a relationship between Tasks and Persons, ‘location_of‘ transforms a Person into a Location, and ‘skill_level‘ represents a property of a Person in relation to a specific Task. The type-checking ensures that functions are only applied to arguments of the correct sort, reducing logical errors. For example, trying to find the ‘skill_level‘ of an Equipment item for a Task would be caught as a type error. In future work, we hope that these function definitions also open up the possibility of incorporating external algorithms. For instance, the ‘duration’ function could be linked to an external scheduling algorithm that calculates task durations based on various factors.

3. **Constants** ($\mathcal{C}$) with associated sorts provide grounding for abstract reasoning in concrete entities. Let $c_i : S_j$ denote that constant c_i is of sort S_j. In Fig 2 constant declaration grounds the abstract sorts in concrete entities. It allows for easy integration of domain-specific knowledge - for instance, we know that "alice", "bob", and "charlie" are Persons in our system. In future work, we believe this structure has the potential for linking with external databases or knowledge graphs. For example, the "equipment" constants could be linked to an external database containing detailed specifications for each piece of equipment.

4. **Variables** ($\mathcal{V}$) enable clear scoping rules for quantifiers and type-safe substitutions in logical formulas. Let $x_i : S_j$ denote that variable x_i ranges over sort S_j (see Fig 2).

5. **Knowledge Base** ($\mathcal{KB}$) contains axioms or facts assumed to be true within the logical system. Let $\mathcal{KB} = \{\varphi_1, \varphi_2, \ldots, \varphi_m\}$ where each φ_i is a well-formed formula in first-

order logic. The structured knowledge base allows for separation of axioms from rules and queries, and supports incremental knowledge addition. In Fig 2, the knowledge base contains factual information about the problem domain. It's separate from the rules and verifications, allowing for easy updates and additions without changing the core reasoning system. This structure also allows for potential consistency checking. For instance, we could verify that no person is assigned to two tasks at the same time, or that no task has a negative duration.

6. **Rules ($\mathcal{R}$)** specify logical constructs, often involving quantifiers and implications. Let $\mathcal{R} = \{r_1, r_2, \ldots, r_l\}$ where each r_i is a well-formed formula representing a rule. The explicit representation of rules enables a clear distinction between factual knowledge and inferential knowledge. These rules represent inferential knowledge - knowledge that can be derived from the facts in the knowledge base. They allow for easy addition of domain-specific reasoning patterns. This explicit representation of rules also supports explainable AI. When the system makes an inference, it can point to the specific rule(s) used, making its reasoning process transparent and understandable to users. For example, some rules are depicted in Fig 2 and here are a few more:
   - $\forall x : \text{Person}, \text{Worker}(x) \rightarrow \exists y : \text{Equipment}, \text{Wearing}(x, y)$
   - $\forall x : \text{Person}, \forall y : \text{Equipment}, \text{Wearing}(x, y) \wedge \text{SafetyGear}(y) \rightarrow \text{Safe}(x)$
7. **Verifications ($\mathcal{V}$)** state properties or conditions to be verified by the theorem prover. Let $\mathcal{V} = \{v_1, v_2, \ldots, v_p\}$ where each v_i is a well-formed formula to be verified. Separating verifications from the knowledge base and rules allows for clear goal-directed reasoning. These verifications (Fig 2) represent specific properties we want to check in our system. They allow for easy testing and validation of the knowledge base and rules. When a verification fails, the system can potentially generate counter-examples, providing valuable insights into why the desired property doesn't hold. For example:
   - $\forall x : \text{Person}, \text{Worker}(x) \rightarrow \text{Safe}(x)$
   - $\exists x : \text{Person}, \text{Worker}(x) \wedge \neg\text{Safe}(x)$
8. **Optimization (optional) ($\mathcal{O}$)** sections define problems with objectives and constraints. Let $\mathcal{O} = (f_{\text{obj}}, \mathcal{C})$ where f_{obj} is the objective function and $\mathcal{C}$ is a set of constraints. The optimization problem in Fig 2 seeks to minimize the total skill level of assigned persons while ensuring that all tasks are assigned and that each assigned person has at least the minimum required skill level. This ability to combine logical constraints with numerical optimization allows for the representation of complex real-world problems that go beyond pure logical satisfiability. We intend to benchmark this in future work. Here is an example:
   - minimize $f_{\text{obj}}(x) = \sum_{i=1}^{n} \text{cost}(x_i)$
   - subject to: $\forall i, 1 \leq i \leq n, \text{Safe}(x_i)$
9. **Actions ($\mathcal{A}$)** list what the interpreter has to perform. The only two possible actions are 'verify' and 'optimize'. This simple declaration tells the system what to do with the problem representation we've built up. It provides flexibility in choosing different reasoning or optimization approaches for the same problem representation. For extensibility, this structure also opens up possibilities for meta-reasoning about which actions to take. For instance, the system could analyze the problem structure to decide whether to attempt verification first or go straight to optimization.

3.4 Design of the Interpreter's Facilities and Capabilities

This section provides an in-depth description of the interpreter's facilities, detailing how it constructs and manipulates logical expressions.

Type System, Sort Management: The interpreter implements a robust type system, managing sorts and ensuring type safety across all expressions. It supports a variety of Z3 compatible sorts, including primitive sorts like Bool, Int, and Real, which form the foundation of the type system. User-defined sorts, known as declared sorts, allow for the representation of specific domains such as Person or Equipment. For situations requiring a finite set of elements, enumerated sorts are available. The type system also accommodates composite sorts, constructed using type constructors, which enable the creation of function sorts or tuple sorts. Throughout its operations, the interpreter rigorously enforces

type consistency, ensuring that functions and predicates are applied only to arguments of the correct sorts.

Symbol Table, Scope Management: Central to the interpreter's functionality is a symbol table that maintains mappings from identifiers to their definitions, including variables, constants, and functions. This table is crucial for scope management, particularly when dealing with quantified variables in logical expressions. The parsing process is another key component, where the interpreter builds abstract syntax trees (ASTs) that represent the structure of expressions. This process handles a wide range of logical constructs, from atomic formulas (basic predicates applied to terms) to complex formulas constructed using logical connectives and quantifiers. The interpreter pays special attention to quantifiers, carefully managing bound variables to ensure correct scoping. Additionally, it supports substitution of terms for variables, an essential operation in applying inference rules.

Pre-processing: While the bulk of reasoning is handled by the theorem prover, the interpreter applies basic inference and simplification rules to optimize expressions before passing them on. This includes simplification processes that reduce expressions using logical identities, such as eliminating double negations. Normalization is another crucial step, converting expressions into a standard form (like prenex normal form) to facilitate theorem proving. The interpreter also performs early error detection, identifying contradictory statements or type mismatches before they can cause issues in later stages of processing.

Feedback Loop: Adequate error handling and diagnostics are paramount in the interpreter's design. It provides detailed error messages to assist the LLM in identifying and correcting issues with its programs. These diagnostics cover a range of potential problems, including type errors that indicate inconsistencies or mismatches in the type system, alerts for undefined symbols when functions, predicates, or constants are used without proper definition, and syntax errors that highlight issues in the structure of logical expressions.

Future Proofing: The interpreter's architecture emphasizes extensibility and customization. Users have the flexibility to extend its capabilities in several ways. They can add new sorts to define additional domains of discourse, expanding the system's ability to represent complex scenarios. The logical language can be enhanced by defining new functions and predicates, allowing users to capture more intricate relationships within their domain of interest. Furthermore, the modular design of the interpreter facilitates integration with different theorem provers or logic systems, enhancing its versatility and applicability to various problem domains.

4 Results

4.1 StrategyQA - Complex Natural Language Reasoning

Task Setup: StrategyQA presents a significant challenge in natural language processing, testing a model's ability to perform multi-hop, implicit reasoning across diverse scenarios. This boolean question answering benchmark requires models to infer unstated reasoning steps, mirroring complex human cognitive processes. For example, "Did Aristotle use a laptop?" requires the implicit chain: "When did Aristotle live? When was the laptop invented? Do these time periods overlap?" This level of abstraction surpasses simple fact retrieval or explicit reasoning tasks in other benchmarks like BoolQ or Twenty Questions (20QA). While state-of-the-art language models have shown impressive performance on StrategyQA (e.g., PaLM-2 achieving 90.20% accuracy with few-shot Chain of Thought and Self Consistency [Anil et al., 2023]), they lack transparency and verifiability in their reasoning process. Our PROOF OF THOUGHT (PoT) framework addresses this limitation by providing complete, explicit, and verifiable reasoning chains. PoT breaks down implicit reasoning steps into explicit logical representations, defines the knowledge base used, and ensures each inference is provable through a theorem prover.

PoT Results: We evaluated PoT on a sample of 1000 questions from the StrategyQA dataset, focusing on the framework's ability to generate syntactically correct programs, produce provable reasoning chains, and match outputs with correct answers. The system, with the inclusion of a 3 step feedback loops (i.e., initial prompt, +2 attempts at resolving), successfully compiled and executed 82.4% of the 1000 processed questions, marking a significant improvement from runs with lower feedback loops. This increase in compilation success rate underscores the effectiveness of our feedback mechanism in addressing and resolving issues in generated logical representations. The system demonstrated

strong recall at 91.40%, indicating its proficiency in identifying true positive cases. The F1 score of 71.13% suggests a good balance between precision and recall, though precision (58.22%) presents an area for potential enhancement. The high recall, coupled with a false positive rate of 53.98%, indicates a tendency for the system to overpredict positive cases. This observation points to a need for future refinements in discriminating between positive and negative instances more accurately. While the compilation success rate is encouraging, the 17.6% of questions that failed to compile highlight an area for further improvement. Enhancing the robustness of the code generation process through improved prompting techniques, fine-tuning, and expansion of the feedback loop mechanism could potentially reduce this failure rate in future iterations.

4.2 Multimodal Reddit-OSHA Benchmark

Task Setup: We curated 103 samples from the r/OSHA subreddit, featuring individuals in extremely hazardous situations. This dataset represents long-tail, low-probability scenarios, mirroring challenging real-world deployment conditions for health and safety applications. The images encompass a wide range of problems with varied lighting, scene setups, visual clutter, and resolutions.

Baselines: We implemented four reasoning strategies using GPT-4 as the underlying language model: Chain of Thought (CoT), Chain of Thought with Self-Consistency (CoT-SC), Tree of Thought (ToT), and Graph of Thought (GoT). Each baseline processes base64-encoded images and uses a consistent system prompt instructing the model to act as a safety inspector. CoT encourages step-by-step reasoning, concluding with a binary hazard decision. CoT-SC extends this by generating 5 independent reasoning paths per image, with the final decision determined by majority voting. ToT explores multiple reasoning paths in a tree-like structure with a maximum depth of 3 and a breadth of 2 at each node. GoT implements an iterative reasoning process with 3 iterations per image, building upon previous analyses. Evaluation metrics include win rate (proportion of correctly identified hazards) and reasoning richness (number of sentences in model responses).

Baseline Results: All reasoning strategies demonstrated high performance, with CoT achieving a 99.03% win rate and CoT-SC, ToT, and GoT all achieving perfect 100% win rates. This suggests that advanced reasoning strategies can correct errors made by simpler approaches. Reasoning richness varied significantly, with ToT producing the most detailed responses (often over 1000 sentences per image) and CoT the most concise (25-30 sentences).

PoT Results: Our PROOF OF THOUGHT framework showed remarkable improvements on this dataset with the inclusion of a 3 step feedback loop (i.e., initial prompt, +2 attempts at resolving). Notably, we reduced compilation errors from 14.6% to 0%, demonstrating the effectiveness of our feedback and error correction mechanisms. The win rate on compiled programs increased from 72% to 81.55%, indicating both more reliable code generation and more accurate logical reasoning.

5 Discussion and Future work

Future research directions include expanding PoT to handle more complex logical structures, that extend past boolean SAT & UNSAT, using one versus all setups, and developing more sophisticated feedback mechanisms to further reduce compilation errors. We intend to explore JSON-like representations for non-boolean responses. Additionally, exploring ways to make the logical representations more accessible to non-expert users and investigating the scalability of PoT to larger, more diverse datasets will be important next steps. Further, integrating PoT with other techniques such as model updates using reinforcement learning, or supervised fine-tuned models on synthetically generated syntactically correct PoT programs might unlock at-scale "System 2" thinking.

6 Conclusion

PROOF OF THOUGHT bridges the gap between language models' flexibility and formal logic's rigor, offering a promising solution for trustworthy reasoning in vision-language models. By enhancing interpretability and providing reasoning guarantees, PoT addresses critical challenges in AI system accountability and reliability. Our results demonstrate its potential in both natural language and multimodal reasoning tasks, paving the way for more transparent, verifiable AI systems capable of complex reasoning in high-stakes domains.

References

Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*, 2023.

Pietro Barbiero, Gabriele Ciravegna, Francesco Giannini, Mateo Espinosa Zarlenga, Lucie Charlotte Magister, Alberto Tonda, Pietro Lió, Frederic Precioso, Mateja Jamnik, and Giuseppe Marra. Interpretable neural-symbolic concept reasoning. In *International Conference on Machine Learning*, pages 1801–1825. PMLR, 2023.

Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency*, pages 610–623, 2021.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 17682–17690, 2024.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Xiaojun Chen, Shengbin Jia, and Yang Xiang. A review: Knowledge reasoning over knowledge graph. *Expert systems with applications*, 141:112948, 2020a.

Zhenfang Chen, Rui Sun, Wenjun Liu, Yining Hong, and Chuang Gan. Genome: generative neuro-symbolic visual reasoning by growing and reusing modules. *arXiv preprint arXiv:2311.04901*, 2023.

Zhi Chen, Yijie Bei, and Cynthia Rudin. Concept whitening for interpretable image recognition. *Nature Machine Intelligence*, 2(12):772–782, 2020b.

Marina Danilevsky, Kun Qian, Ranit Aharonov, Yannis Katsis, Ban Kawas, and Prithviraj Sen. A survey of the state of explainable ai for natural language processing. *arXiv preprint arXiv:2010.00711*, 2020.

Artur S d'Avila Garcez, Luís C Lamb, and Dov M Gabbay. *Neural-symbolic learning systems*. Springer, 2009.

Tanmay Gupta and Aniruddha Kembhavi. Visual programming: Compositional visual reasoning without training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14953–14962, 2023.

Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

Michael Kampffmeyer, Yinbo Chen, Xiaodan Liang, Hao Wang, Yujia Zhang, and Eric P Xing. Rethinking knowledge graph propagation for zero-shot learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11487–11496, 2019.

Been Kim, Martin Wattenberg, Justin Gilmer, Carrie Cai, James Wexler, Fernanda Viegas, et al. Interpretability beyond feature attribution: Quantitative testing with concept activation vectors (tcav). In *International conference on machine learning*, pages 2668–2677. PMLR, 2018.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Ziyang Li, Jiani Huang, and Mayur Naik. Scallop: A language for neurosymbolic programming. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1463–1487, 2023.

Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. *Advances in neural information processing systems*, 31, 2018.

Emanuele Marconato, Samuele Bortolotti, Emile van Krieken, Antonio Vergari, Andrea Passerini, and Stefano Teso. Bears make neuro-symbolic models aware of their reasoning shortcuts. *arXiv preprint arXiv:2402.12240*, 2024.

Gary Marcus. The next decade in ai: four steps towards robust artificial intelligence. *arXiv preprint arXiv:2002.06177*, 2020.

Cynthia Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature machine intelligence*, 1(5):206–215, 2019.

Garrett Tanzer, Mirac Suzgun, Eline Visser, Dan Jurafsky, and Luke Melas-Kyriazi. A benchmark for learning to translate a new language from one grammar book. *arXiv preprint arXiv:2309.16575*, 2023.

Geoffrey G Towell and Jude W Shavlik. Knowledge-based artificial neural networks. *Artificial intelligence*, 70(1-2):119–165, 1994.

Geoffrey G Towell, Jude W Shavlik, and Michiel O Noordewier. Refinement of approximate domain theories by knowledge-based neural networks. In *Proceedings of the eighth National conference on Artificial intelligence-Volume 2*, pages 861–866, 1990.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Jianwei Yang, Hao Zhang, Feng Li, Xueyan Zou, Chunyuan Li, and Jianfeng Gao. Set-of-mark prompting unleashes extraordinary visual grounding in gpt-4v. *arXiv preprint arXiv:2310.11441*, 2023.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning (still) requires rethinking generalization. *Communications of the ACM*, 64(3):107–115, 2021.

Barret Zoph, Deniz Yuret, Jonathan May, and Kevin Knight. Transfer learning for low-resource neural machine translation. *arXiv preprint arXiv:1604.02201*, 2016.

7 Appendix

7.1 StrategyQA Results

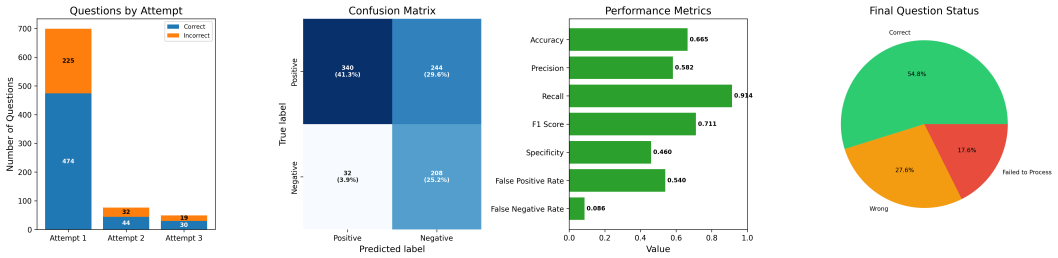

Figure 3: Performance analysis of the Proof of Thought (PoT) framework on the StrategyQA dataset. The includes four visualizations: (1) a stacked bar chart showing questions answered by attempt, (2) a pie chart displaying the final question status, (3) a confusion matrix for predicted vs. true labels, and (4) a bar chart of various performance metrics including accuracy, precision, recall, F1-score, specificity, and false positive rate.

7.2 Multimodal Reddit-OSHA Benchmark

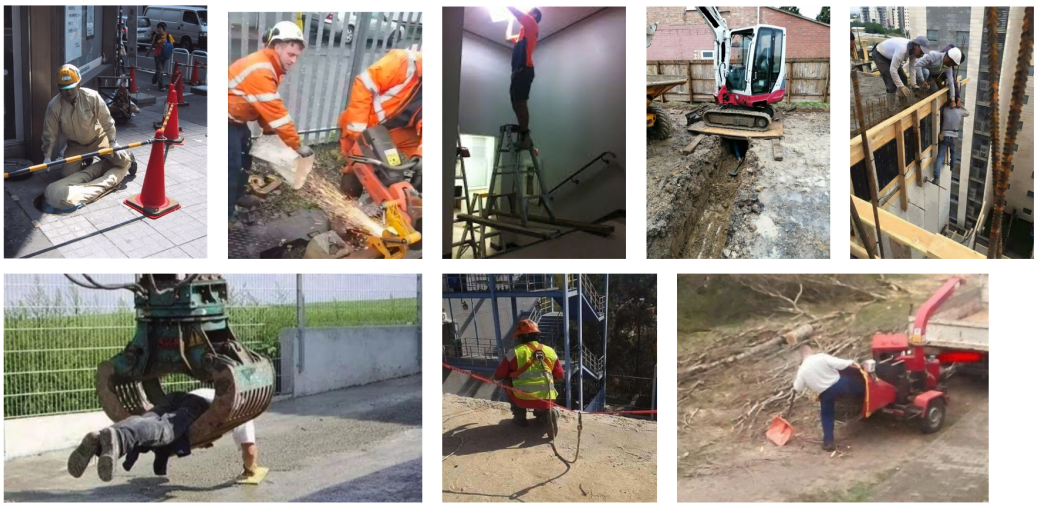

Figure 4: Sample Images from the Multimodal Reddit-OSHA Benchmark

7.3 Real Reasoning Traces using PoT : StrategyQA Dataset

1. **Question:** Could Javier Sotomayor jump over the head of the average giraffe?
   Answer: False
   Predicted Answer: UNSAT (False).

```
{
    "sorts": [
        {"name": "Person", "type": "DeclareSort"},
        {"name": "Animal","type": "DeclareSort"},
        {"name": "Real","type": "RealSort"}
    ],
    "functions": [
        {"name": "jump_height","domain": ["Person"],"range": "Real"},
        {"name": "height","domain": ["Animal"],"range": "Real"}
    ],
    "constants": {
        "persons": {"sort": "Person","members": ["javier_sotomayor"]},
        "animals": {"sort": "Animal","members": ["average_giraffe"]}
```

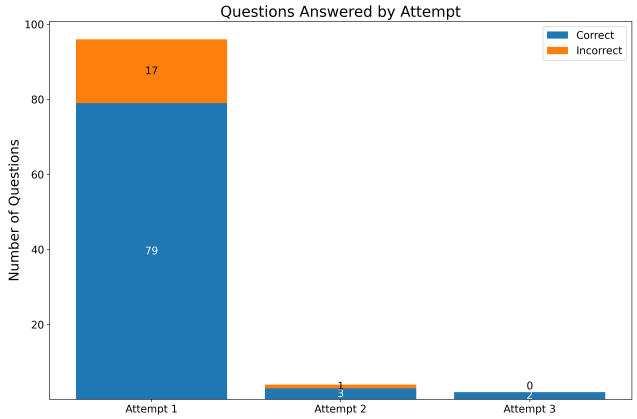

Figure 5: Performance analysis of the Proof of Thought (PoT) framework on the Multimodal Reddit-OSHA Benchmark dataset with a stacked bar chart showing questions answered by attempt.

```
    },
    "variables": [
        {"name": "p","sort": "Person"},
        {"name": "a","sort": "Animal"}
    ],
    "knowledge_base": [
        {"assertion": "jump_height(javier_sotomayor) == 2.45"},
        {"assertion": "height(average_giraffe) == 5.5"}
    ],
    "verifications": [
        {"name": "Sotomayor Jump Over Giraffe","constraint": "jump_height(javier_sotomayor) >=
              height(average_giraffe)"
        }
    ],
    "actions": ["verify_conditions"]
}
```

2. Question: Did the Cherokee people send a delegation to oppose allotment?
Answer: True
Predicted Answer: SAT (True)

```
{
    "sorts": [
        {"name": "Group","type": "DeclareSort"},
        {"name": "Action","type": "DeclareSort"},
        {"name": "Bool","type": "BoolSort"}
    ],
    "functions": [
        {"name": "send_delegation","domain": ["Group"],"range": "Bool"},
        {"name": "oppose_allotment","domain": ["Group"],"range": "Bool"}
    ],
    "constants": {
        "groups": {"sort": "Group","members": ["cherokee_people"]},
        "actions": {"sort": "Action","members": ["allotment"]}
    },
    "variables": [
        {"name": "g","sort": "Group"}
    ],
    "knowledge_base": [
        {"assertion": "send_delegation(cherokee_people)"},
        {"assertion": "ForAll([g], Implies(send_delegation(g), oppose_allotment(g)))",
            "variables": [{"name": "g","sort": "Group"}]
        }
    ],
    "verifications": [
        {"name": "Cherokee Oppose Allotment","constraint": "oppose_allotment(cherokee_people)
              "}
    ],
    "actions": ["verify_conditions"]
}
```

7.4 Real Reasoning Traces using PoT : Multimodal Reddit-OSHA Dataset

1. HSE Example 1

Figure 6: Example 1-3 from the HSE Reddit OSHA Dataset.

```
{
    "sorts": [
        {"name": "Person","type": "DeclareSort"},
        {"name": "Equipment","type": "DeclareSort"},
        {"name": "SafetyGear","type": "DeclareSort"}
    ],
    "functions": [
        {"name": "Using","domain": ["Person","Equipment"],"range": "BoolSort"
        },
        {"name": "Wearing","domain": ["Person","SafetyGear"],"range": "BoolSort"}
    ],
    "constants": {
        "persons": {"sort": "Person","members": ["worker"]},
        "equipments": {"sort": "Equipment","members": ["ladder"]},
        "safetyGears": {"sort": "SafetyGear","members": ["hardHat","harness"]}
    },
    "knowledge_base": [
        {"assertion": "Using(worker, ladder)","value": true},
        {"assertion": "Wearing(worker, hardHat)","value": false},
        {"assertion": "Wearing(worker, harness)","value": false}
    ],
    "rules": [
        {"name": "Hard Hat Rule","forall": [{"name": "p","sort": "Person"},
        {"name": "e","sort": "Equipment"}],"implies": {"antecedent": "Using(p, e)", "
            consequent": "Wearing(p, hardHat)"}},
        {"name": "Harness Rule","forall": [{"name": "p","sort": "Person"}, {"name": "e","sort
            ": "Equipment"}],
            "implies": {"antecedent": "Using(p, e)","consequent": "Wearing(p, harness)"}}
    ],
    "verifications": [
        {"name": "Verify Hard Hat Compliance","constraint": "Wearing(worker, hardHat)"},
        {"name": "Verify Harness Compliance","constraint": "Wearing(worker, harness)"}
    ],
    "actions": ["verify_conditions"]
}
```

2. HSE Example 2

```
{
    "sorts": [{"name": "Person","type": "DeclareSort"},
        {"name": "Equipment","type": "DeclareSort"},
        {"name": "SafetyEquipment","type": "DeclareSort"}
    ],
    "functions": [
        {"name": "StandingOn","domain": ["Person","Equipment"],"range": "BoolSort"},
        {"name": "UsingSafetyEquipment","domain": ["Person","SafetyEquipment"],"range": "
            BoolSort"},
        {"name": "IsSafe","domain": ["Person"],"range": "BoolSort"}
    ],
    "constants": {
        "persons": {"sort": "Person","members": ["worker"]},
        "equipments": {"sort": "Equipment","members": ["forklift","pallet"]},
        "safetyEquipments": {"sort": "SafetyEquipment","members": ["harness"]}
    },
    "knowledgebase": [
        {"assertion": "StandingOn(worker, pallet)","value": true},
        {"assertion": "UsingSafetyEquipment(worker, harness)","value": false}
    ],
    "rules": [
        {"name": "Safety Rule","forall": [{"name": "p","sort": "Person"}],
            "implies": {"antecedent": "And(StandingOn(p, pallet), Not(UsingSafetyEquipment(p,
                harness)))","consequent": "Not(IsSafe(p))"}}
```

```
    ],
    "verifications": [
        {"name": "Verify Safety","constraint": "IsSafe(worker)"}
    ],
    "actions": ["verify_conditions"]
}
```

3. HSE Example 3

```
{
    "sorts": [
        {"name": "Person","type": "DeclareSort"},
        {"name": "Equipment","type": "DeclareSort"},
        {"name": "Location","type": "DeclareSort"}
    ],
    "functions": [
        {"name": "Worker","domain": ["Person"],"range": "BoolSort"},
        {"name": "Using","domain": ["Person","Equipment"],"range": "BoolSort"},
        {"name": "AtHeight","domain": ["Person"],"range": "BoolSort"},
        {"name": "HasFallProtection","domain": ["Person"],"range": "BoolSort"},
        {"name": "Stable","domain": ["Equipment"],"range": "BoolSort"}
    ],
    "constants": {
        "persons": {"sort": "Person","members": ["worker"]},
        "equipments": {"sort": "Equipment","members": ["ladder","scaffold"]},
        "locations": {"sort": "Location","members": ["worksite"]}
    },
    "knowledge_base": [
        "Worker(worker)","Using(worker, ladder)","Using(worker, scaffold)", "AtHeight(worker)"
            ,
        {"assertion": "Stable(ladder)","value": false},
        {"assertion": "Stable(scaffold)","value": false},
        {"assertion": "HasFallProtection(worker)","value": false}
    ],
    "rules": [
        {
        "name": "Safety Rule","forall": [{"name": "p","sort": "Person"}],
            "implies": {"antecedent": "And(Worker(p), AtHeight(p))", "consequent": "
                HasFallProtection(p)"}
        },
        {
            "name": "Stability Rule","forall": [{"name": "e","sort": "Equipment"}],
            "implies": {"antecedent": "Using(worker, e)","consequent": "Stable(e)"}
        }],
    "verifications": [
        {"name": "Verify Safety", "constraint": "And(HasFallProtection(worker), Stable(ladder)
            , Stable(scaffold))"}
    ],
    "actions": ["verify_conditions"]
}
```

7.5 A Qualitative Analysis of Generated DSL Programs and Reasoning Patterns

The generated DSL programs across both the StrategyQA dataset and the OSHA dataset illustrate how formal logical representations can be used to model complex reasoning tasks. In both cases, the structured use of sorts, functions, rules, and verifications ensures that the questions posed are systematically decomposed into logical assertions that can be verified by a theorem prover such as Z3. Here, we analyze key aspects of these programs and how they contribute to effective reasoning.

7.5.1 Sorts and Function Definitions as the Backbone of Logical Modeling

The use of `DeclareSort`, `BoolSort`, `RealSort`, and other basic sorts in the DSL programs serves as the foundation for defining the domains of discourse. For example, in the StrategyQA question involving Javier Sotomayor and a giraffe, the `Person` and `Animal` sorts allow the definition of relationships between humans and animals in terms of measurable attributes (e.g., jump height and height). Similarly, in the OSHA-related examples, `Person`, `Equipment`, and `SafetyGear` sorts model the entities relevant to workplace safety.

By defining functions like `jump_height`, `height`, `Wearing`, and `Using`, we map the relationships between entities and their properties. These functions serve as predicates that are later used in verifications or rule implications. In these cases, the functions provide critical context for understanding the state of the world and the conditions under which certain outcomes (e.g., compliance with safety regulations or reaching a height) hold true.

7.5.2 Knowledge Base and Its Role in Establishing Ground Truth

The `knowledge_base` section plays a vital role in grounding the reasoning process by introducing factual information, such as the jump height of Javier Sotomayor (2.45 meters) or the height of an average giraffe (5.5 meters). This knowledge is essential for theorem proving because it establishes the foundational truths that the logical system will work with. Similarly, in the OSHA examples, the knowledge base specifies whether a worker is using certain equipment, wearing protective gear, or working at height.

In both datasets, the knowledge base is used to capture the known facts that are assumed to be true at the start of reasoning. This helps set the initial conditions for the logical rules to be applied.

7.5.3 Rules as Key Drivers of Logical Implication

The `rules` section formalizes the relationships between entities based on conditional logic. These rules encapsulate the domain knowledge and drive the reasoning process. For instance, the "Hard Hat Rule" in the OSHA examples states that if a person is using equipment, they should also be wearing a hard hat, while the "Harness Rule" mandates that a harness should be worn when using certain equipment. In the StrategyQA example, no explicit rules are needed beyond the basic comparison of heights.

These rules introduce a level of generalization that allows the reasoning process to handle not just specific instances but also classes of entities. For example, the rule:

```
{
"name": "Hard Hat Rule",
"forall": [{"name": "p","sort": "Person"}, {"name": "e","sort": "Equipment"}],
"implies": {"antecedent": "Using(p, e)", "consequent": "Wearing(p, hardHat)"}
}
```

is applicable to any person and any equipment. This generalization is key in enabling logical inferences beyond the immediate facts provided in the knowledge base.

7.5.4 Verifications as the Core of Decision-Making

The `verifications` section in these programs forms the basis of decision-making by checking whether the conditions specified in the knowledge base and rules hold. In the StrategyQA case, the verification checks if Javier Sotomayor's jump height is greater than or equal to the height of an average giraffe. The outcome of this check (UNSAT) indicates that it is false, thus the predicted answer matches the correct answer.

In the OSHA-related examples, verifications are used to ensure compliance with safety rules. For example, the program checks if the worker is wearing a hard hat or harness while using a ladder. These verifications serve as the final step in determining whether the conditions needed for safety are met or not. The results of these verifications provide a direct SAT (true) or UNSAT (false) outcome, which can then be used to assess compliance or answer the given question.

7.5.5 Patterns in Error Detection and Resolution

The DSL framework helps detect inconsistencies or non-compliance in the input data. For example, the OSHA programs can flag situations where workers are using unsafe equipment or failing to follow safety protocols. This is achieved by systematically comparing the facts provided in the knowledge base with the rules and verifications, ensuring that errors are caught before any conclusions are drawn.

Additionally, the explicit use of logical operators like `And`, `Not`, and `Implies` makes it easy to trace the reasoning path when an outcome is SAT (true) or UNSAT (false). This traceability allows users to understand why a particular result was obtained, making the reasoning process more transparent and interpretable.

7.5.6 Overall Analysis and Utility of Generated DSL Programs

Across both the StrategyQA and OSHA datasets, the use of DSL programs enables structured, logical reasoning that is verifiable and interpretable. The modularity of the DSL—where different aspects of

the reasoning process (entities, relationships, rules, and verifications) are clearly separated—ensures that the programs remain adaptable to a variety of problem domains.

The reasoning traces provided by these DSL programs offer significant benefits:

- **Interpretability**: The modular structure makes it easy to follow the logical steps leading to a conclusion.
- **Error Detection**: The use of formal logic allows for early detection of contradictions or violations of safety rules.
- **Scalability**: The DSL framework can handle increasingly complex scenarios by adding new sorts, functions, rules, and verifications.
- **Generalization**: Rules written in a generalized form (e.g., using `ForAll`) can be applied across different entities and scenarios, making the system more flexible.

7.6 Exploring the possibilities: Satisfiable Neurosymbolic Programs

Our DSL designed to be very expressive, and future proof for additional scenarios. In this subsection we present some example problems that can be expressed and solved to be found SAT.

1. **Simple Arithmetic Verification** : Verify that there exists an integer x such that x + 2 = 5.

```
{
  "sorts": [
    {"name": "Int", "type": "IntSort"}
  ],
  "functions": [],
  "constants": {},
  "knowledge_base": [],
  "rules": [],
  "verifications": [
    {
      "name": "verify_addition",
      "exists": [
        {"name": "x", "sort": "Int"}
      ],
      "constraint": "x + 2 == 5"
    }
  ],
  "actions": ["verify_conditions"]
}
```

2. **Basic Safety Equipment Rule** : Ensure all workers are wearing hard hats.

```
{
    "sorts": [
      {"name": "Person", "type": "DeclareSort"},
      {"name": "Equipment", "type": "DeclareSort"}
    ],
    "functions": [
      {"name": "Worker", "domain": ["Person"], "range": "BoolSort"},
      {"name": "Wearing", "domain": ["Person", "Equipment"], "range": "BoolSort"}
    ],
    "constants": {
      "persons": {
        "sort": "Person",
        "members": ["alice", "bob"]
      },
      "equipments": {
        "sort": "Equipment",
        "members": ["hardHat"]
      }
    },
    "knowledge_base": [
      "Worker(alice)",
      "Worker(bob)",
      "Wearing(alice, hardHat)"
    ],
    "rules": [
      {
        "name": "Hard Hat Rule",
        "forall": [
          {"name": "p", "sort": "Person"}
        ],
        "implies": {
          "antecedent": "Worker(p)",
          "consequent": "Wearing(p, hardHat)"
        }
```

```
    }
  ],
  "verifications": [
    {
      "name": "Check Hard Hat Compliance",
      "forall": [
        {"name": "p", "sort": "Person"}
      ],
      "implies": {
        "antecedent": "Worker(p)",
        "consequent": "Wearing(p, hardHat)"
      }
    }
  ],
  "actions": ["verify_conditions"]
}
```

3. **Parent-Child Relationship** : Define a family tree and verify that a grandparent relationship holds.

```
{
  "sorts": [
    {"name": "Person", "type": "DeclareSort"}
  ],
  "functions": [
    {"name": "parent_of", "domain": ["Person"], "range": "Person"}
  ],
  "constants": {
    "persons": {
      "sort": "Person",
      "members": ["alice", "bob", "charlie"]
    }
  },
  "knowledge_base": [
    "parent_of(bob) == alice",
    "parent_of(charlie) == bob"
  ],
  "rules": [],
  "verifications": [
    {
      "name": "Verify Grandparent",
      "constraint": "parent_of(parent_of(charlie)) == alice"
    }
  ],
  "actions": ["verify_conditions"]
}
```

4. **Transitive Relation Verification** : Verify that a transitive property holds in a relation.

```
{
    "sorts": [
      {"name": "Element", "type": "DeclareSort"}
    ],
    "functions": [
      {"name": "related", "domain": ["Element", "Element"], "range": "BoolSort"}
    ],
    "constants": {
      "elements": {
        "sort": "Element",
        "members": ["x", "y", "z"]
      }
    },
    "knowledge_base": [
      "related(x, y)",
      "related(y, z)"
    ],
    "rules": [
      {
        "name": "Transitive Rule",
        "forall": [
          {"name": "a", "sort": "Element"},
          {"name": "b", "sort": "Element"},
          {"name": "c", "sort": "Element"}
        ],
        "implies": {
          "antecedent": "And(related(a, b), related(b, c))",
          "consequent": "related(a, c)"
        }
      }
    ],
    "verifications": [
      {
        "name": "Verify Transitivity",
        "constraint": "related(x, z)"
      }
    ],
    "actions": ["verify_conditions"]
```

```
}
```

5. **Scheduling Without Conflicts** : Ensure two tasks are scheduled at different times.

```
{
    "sorts": [
      {"name": "Task", "type": "DeclareSort"},
      {"name": "TimeSlot", "type": "IntSort"}
    ],
    "functions": [
      {"name": "scheduled_at", "domain": ["Task"], "range": "TimeSlot"}
    ],
    "constants": {
      "tasks": {
        "sort": "Task",
        "members": ["task1", "task2"]
      }
    },
    "knowledge_base": [],
    "rules": [],
    "verifications": [
      {
        "name": "Verify Scheduling",
        "exists": [
          {"name": "t1", "sort": "TimeSlot"},
          {"name": "t2", "sort": "TimeSlot"}
        ],
        "constraint": "And(scheduled_at(task1) == t1, scheduled_at(task2) == t2, t1 != t2)"
      }
    ],
    "actions": ["verify_conditions"]
  }
```

6. **Graph Coloring Problem** : Assign colors to nodes such that adjacent nodes have different colors.

```
  {
    "sorts": [
      {"name": "Node", "type": "DeclareSort"},
      {"name": "Color", "type": "DeclareSort"}
    ],
    "functions": [
      {"name": "color_of", "domain": ["Node"], "range": "Color"},
      {"name": "connected", "domain": ["Node", "Node"], "range": "BoolSort"}
    ],
    "constants": {
      "nodes": {
        "sort": "Node",
        "members": ["node1", "node2", "node3"]
      },
      "colors": {
        "sort": "Color",
        "members": ["red", "green", "blue"]
      }
    },
    "knowledge_base": [
      "connected(node1, node2)",
      "connected(node2, node3)",
      "connected(node1, node3)"
    ],
    "rules": [
      {
        "name": "Coloring Rule",
        "forall": [
          {"name": "n1", "sort": "Node"},
          {"name": "n2", "sort": "Node"}
        ],
        "implies": {
          "antecedent": "connected(n1, n2)",
          "consequent": "color_of(n1) != color_of(n2)"
        }
      }
    ],
    "verifications": [
      {
        "name": "Verify Coloring",
        "exists": [
          {"name": "c1", "sort": "Color"},
          {"name": "c2", "sort": "Color"},
          {"name": "c3", "sort": "Color"}
        ],
        "constraint": "And(color_of(node1) == c1, color_of(node2) == c2, color_of(node3) == c3
            )"
      }
    ],
    "actions": ["verify_conditions"]
  }
```

7. **Health and Safety Scenario** : Verify that all workers at heights above 6 feet are wearing safety harnesses.

```
{
  "sorts": [
    {"name": "Person", "type": "DeclareSort"},
    {"name": "Equipment", "type": "DeclareSort"},
    {"name": "Location", "type": "DeclareSort"}
  ],
  "functions": [
    {"name": "Worker", "domain": ["Person"], "range": "BoolSort"},
    {"name": "At", "domain": ["Person", "Location"], "range": "BoolSort"},
    {"name": "Wearing", "domain": ["Person", "Equipment"], "range": "BoolSort"},
    {"name": "Height", "domain": ["Location"], "range": "IntSort"}
  ],
  "constants": {
    "persons": {
      "sort": "Person",
      "members": ["worker1", "worker2"]
    },
    "equipments": {
      "sort": "Equipment",
      "members": ["safetyHarness"]
    },
    "locations": {
      "sort": "Location",
      "members": ["groundLevel", "highLevel"]
    }
  },
  "knowledge_base": [
    "Worker(worker1)",
    "Worker(worker2)",
    "At(worker1, groundLevel)",
    "At(worker2, highLevel)",
    "Height(groundLevel) == 0",
    "Height(highLevel) == 20",
    "Wearing(worker1, safetyHarness)"
  ],
  "rules": [
    {
      "name": "Fall Protection Rule",
      "forall": [
        {"name": "p", "sort": "Person"},
        {"name": "l", "sort": "Location"}
      ],
      "implies": {
        "antecedent": "And(Worker(p), At(p, l), Height(l) > 6)",
        "consequent": "Wearing(p, safetyHarness)"
      }
    }
  ],
  "verifications": [
    {
      "name": "Check Fall Protection",
      "forall": [
        {"name": "p", "sort": "Person"},
        {"name": "l", "sort": "Location"}
      ],
      "implies": {
        "antecedent": "And(Worker(p), At(p, l), Height(l) > 6)",
        "consequent": "Wearing(p, safetyHarness)"
      }
    }
  ],
  "actions": ["verify_conditions"]
}
```

8. **Electrical Safety Scenario**: Ensure workers using energized equipment above 250V are wearing insulated gloves.

```
{
    "sorts": [
      {"name": "Person", "type": "DeclareSort"},
      {"name": "Equipment", "type": "DeclareSort"}
    ],
    "functions": [
      {"name": "Worker", "domain": ["Person"], "range": "BoolSort"},
      {"name": "Using", "domain": ["Person", "Equipment"], "range": "BoolSort"},
      {"name": "IsEnergized", "domain": ["Equipment"], "range": "BoolSort"},
      {"name": "Voltage", "domain": ["Equipment"], "range": "IntSort"},
      {"name": "Wearing", "domain": ["Person", "Equipment"], "range": "BoolSort"}
    ],
    "constants": {
      "persons": {
        "sort": "Person",
        "members": ["worker1"]
      },
      "equipments": {
```

```
        "sort": "Equipment",
        "members": ["circuitBreaker", "insulatedGloves"]
      }
    },
    "knowledge_base": [
      "Worker(worker1)",
      "Using(worker1, circuitBreaker)",
      "IsEnergized(circuitBreaker)",
      "Voltage(circuitBreaker) == 480"
    ],
    "rules": [
      {
        "name": "High Voltage Safety Rule",
        "forall": [
          {"name": "p", "sort": "Person"},
          {"name": "e", "sort": "Equipment"}
        ],
        "implies": {
          "antecedent": "And(Worker(p), Using(p, e), IsEnergized(e), Voltage(e) > 250)",
          "consequent": "Wearing(p, insulatedGloves)"
        }
      }
    ],
    "verifications": [
      {
        "name": "Verify Electrical Safety",
        "forall": [
          {"name": "p", "sort": "Person"},
          {"name": "e", "sort": "Equipment"}
        ],
        "implies": {
          "antecedent": "And(Worker(p), Using(p, e), IsEnergized(e), Voltage(e) > 250)",
          "consequent": "Wearing(p, insulatedGloves)"
        }
      }
    ],
    "actions": ["verify_conditions"]
  }
```


9. **Chemical Handling Safety** : Ensure workers handling corrosive chemicals are wearing gloves and goggles.


```
{
    "sorts": [
      {"name": "Person", "type": "DeclareSort"},
      {"name": "Chemical", "type": "DeclareSort"},
      {"name": "Equipment", "type": "DeclareSort"}
    ],
    "functions": [
      {"name": "Worker", "domain": ["Person"], "range": "BoolSort"},
      {"name": "Handling", "domain": ["Person", "Chemical"], "range": "BoolSort"},
      {"name": "IsCorrosive", "domain": ["Chemical"], "range": "BoolSort"},
      {"name": "Wearing", "domain": ["Person", "Equipment"], "range": "BoolSort"}
    ],
    "constants": {
      "persons": {
        "sort": "Person",
        "members": ["worker1"]
      },
      "chemicals": {
        "sort": "Chemical",
        "members": ["acid"]
      },
      "equipments": {
        "sort": "Equipment",
        "members": ["gloves", "goggles"]
      }
    },
    "knowledge_base": [
      "Worker(worker1)",
      "Handling(worker1, acid)",
      "IsCorrosive(acid)"
    ],
    "rules": [
      {
        "name": "Corrosive Chemical Handling Rule",
        "forall": [
          {"name": "p", "sort": "Person"},
          {"name": "c", "sort": "Chemical"}
        ],
        "implies": {
          "antecedent": "And(Worker(p), Handling(p, c), IsCorrosive(c))",
          "consequent": "And(Wearing(p, gloves), Wearing(p, goggles))"
        }
      }
    ],
    "verifications": [
      {
        "name": "Verify Chemical Safety",
```

```
      "forall": [
        {"name": "p", "sort": "Person"},
        {"name": "c", "sort": "Chemical"}
      ],
      "implies": {
        "antecedent": "And(Worker(p), Handling(p, c), IsCorrosive(c))",
        "consequent": "And(Wearing(p, gloves), Wearing(p, goggles))"
      }
    }
  ],
  "actions": ["verify_conditions"]
}
```

10. **Resource Allocation Optimization** : Allocate tasks to workers while minimizing total cost.

```
{
  "sorts": [
    {"name": "Worker", "type": "DeclareSort"},
    {"name": "Task", "type": "DeclareSort"},
    {"name": "Cost", "type": "IntSort"}
  ],
  "functions": [
    {"name": "assigned_to", "domain": ["Task"], "range": "Worker"},
    {"name": "cost_of", "domain": ["Worker"], "range": "Cost"}
  ],
  "constants": {
    "workers": {
      "sort": "Worker",
      "members": ["worker1", "worker2"]
    },
    "tasks": {
      "sort": "Task",
      "members": ["taskA", "taskB"]
    }
  },
  "knowledge_base": [
    "cost_of(worker1) == 50",
    "cost_of(worker2) == 30"
  ],
  "rules": [],
  "verifications": [],
  "optimization": {
    "constraints": [
      "assigned_to(taskA) != assigned_to(taskB)"
    ],
    "objectives": [
      {
        "type": "minimize",
        "expression": "cost_of(assigned_to(taskA)) + cost_of(assigned_to(taskB))"
      }
    ]
  },
  "actions": ["optimize"]
}
```

7.7 Exploring the possibilities: Unsatisfiable Neurosymbolic Programs

Our DSL designed to be very expressive, and future proof for additional scenarios. In this subsection we present some example problems that can be expressed and solved to be found UNSAT.

1. **Pigeonhole Principle:** A classic unsatisfiable problem where more pigeons than holes cannot be assigned uniquely.

```
{
    "sorts": [
      {
        "name": "Pigeon",
        "type": "EnumSort",
        "values": ["p1", "p2", "p3", "p4"]
      },
      {
        "name": "Hole",
        "type": "EnumSort",
        "values": ["h1", "h2", "h3"]
      }
    ],
    "functions": [
      {"name": "assigned_to", "domain": ["Pigeon"], "range": "Hole"}
    ],
    "constants": {},
    "knowledge_base": [],
    "rules": [],
    "verifications": [
      {
```

```
        "name": "Pigeonhole Verification",
        "constraint": "Distinct(assigned_to(p1), assigned_to(p2), assigned_to(p3), assigned_to
             (p4))"
      }
    ],
    "actions": ["verify_conditions"]
  }
```

2. **Non-Three-Colorable Graph :** A complete graph with four nodes cannot be colored with only three colors without adjacent nodes sharing the same color.

```
{
    "sorts": [
      {
        "name": "Node",
        "type": "EnumSort",
        "values": ["n1", "n2", "n3", "n4"]
      },
      {
        "name": "Color",
        "type": "EnumSort",
        "values": ["red", "green", "blue"]
      }
    ],
    "functions": [
      {"name": "color_of", "domain": ["Node"], "range": "Color"},
      {"name": "connected", "domain": ["Node", "Node"], "range": "BoolSort"}
    ],
    "constants": {},
    "knowledge_base": [
      "connected(n1, n2)",
      "connected(n1, n3)",
      "connected(n1, n4)",
      "connected(n2, n3)",
      "connected(n2, n4)",
      "connected(n3, n4)"
    ],
    "rules": [
      {
        "name": "Coloring Rule",
        "forall": [
          {"name": "n1", "sort": "Node"},
          {"name": "n2", "sort": "Node"}
        ],
        "implies": {
          "antecedent": "And(connected(n1, n2), n1 != n2)",
          "consequent": "color_of(n1) != color_of(n2)"
        }
      }
    ],
    "verifications": [
      {
        "name": "Verify 3-Coloring",
        "constraint": "True"
      }
    ],
    "actions": ["verify_conditions"]
  }
```

3. **Contradictory Mathematical Constraints**

```
{
    "sorts": [
      {"name": "Int", "type": "IntSort"}
    ],
    "functions": [],
    "constants": {},
    "knowledge_base": [],
    "rules": [],
    "verifications": [
      {
        "name": "Contradictory Constraints",
        "exists": [
          {"name": "x", "sort": "Int"}
        ],
        "constraint": "And(x > 0, x < 0)"
      }
    ],
    "actions": ["verify_conditions"]
  }
```

4. **An Unsatisfiable Boolean formula** in conjunctive normal form (CNF).

```
{
    "sorts": [
      {"name": "Bool", "type": "BoolSort"}
```

```
  ],
  "functions": [],
  "constants": {
    "variables": {
      "sort": "Bool",
      "members": ["A", "B"]
    }
  },
  "knowledge_base": [],
  "rules": [],
  "verifications": [
    {
      "name": "Unsatisfiable CNF",
      "constraint": "And(A, Not(A))"
    }
  ],
  "actions": ["verify_conditions"]
}
```

5. **Mutual Exclusivity**: Defining two constants that cannot be equal, but also constrained to be equal.

```
{
  "sorts": [
    {"name": "Element", "type": "DeclareSort"}
  ],
  "functions": [],
  "constants": {
    "elements": {
      "sort": "Element",
      "members": ["e1", "e2"]
    }
  },
  "knowledge_base": [
    "e1 != e2",
    "e1 == e2"
  ],
  "rules": [],
  "verifications": [
    {
      "name": "Mutual Exclusivity Verification",
      "constraint": "True"
    }
  ],
  "actions": ["verify_conditions"]
}
```

6. **Inconsistent Equations**

```
{
  "sorts": [
    {"name": "Int", "type": "IntSort"}
  ],
  "functions": [],
  "constants": {},
  "knowledge_base": [],
  "rules": [],
  "verifications": [
    {
      "name": "Inconsistent Equations",
      "exists": [
        {"name": "x", "sort": "Int"},
        {"name": "y", "sort": "Int"}
      ],
      "constraint": "And(x + y == 10, x + y == 5)"
    }
  ],
  "actions": ["verify_conditions"]
}
```

7. **Unsolvable Scheduling Conflict** : Tasks that must occur at the same time and also at different times.

```
{
  "sorts": [
    {"name": "Task", "type": "EnumSort", "values": ["task1", "task2"]}
  ],
  "functions": [
    {"name": "scheduled_at", "domain": ["Task"], "range": "IntSort"}
  ],
  "constants": {},
  "knowledge_base": [
    "scheduled_at(task1) == scheduled_at(task2)",
    "scheduled_at(task1) != scheduled_at(task2)"
```

```
    ],
    "rules": [],
    "verifications": [
      {
        "name": "Scheduling Conflict Verification",
        "constraint": "True"
      }
    ],
    "actions": ["verify_conditions"]
  }
```

8. **Invalid Parent-Child Relationship**

```
{
    "sorts": [
      {"name": "Person", "type": "EnumSort", "values": ["bob"]}
    ],
    "functions": [
      {"name": "parent_of", "domain": ["Person"], "range": "Person"}
    ],
    "constants": {},
    "knowledge_base": [
      "parent_of(bob) == bob"
    ],
    "rules": [
      {
        "name": "No Self Parenting Rule",
        "forall": [
          {"name": "p", "sort": "Person"}
        ],
        "implies": {
          "antecedent": "True",
          "consequent": "parent_of(p) != p"
        }
      }
    ],
    "verifications": [
      {
        "name": "Self Parenting Verification",
        "constraint": "True"
      }
    ],
    "actions": ["verify_conditions"]
  }
```

9. **Impossible Optimization**

```
  {
  "sorts": [
    {"name": "Int", "type": "IntSort"}
  ],
  "functions": [],
  "constants": {},
  "knowledge_base": [],
  "rules": [],
  "verifications": [],
  "optimization": {
    "constraints": [
      "x > 0",
      "x < 0"
    ],
    "objectives": [
      {
        "type": "minimize",
        "expression": "x"
      }
    ]
  },
  "actions": ["optimize"]
}
```