COOL: A Constraint Object-Oriented Logic Programming Language and its Neural-Symbolic Compilation System

Jipeng Han*

Beijing Huagui Technology

September 2023

1 Abstract

This paper explores the integration of neural networks with logic programming, addressing the longstanding challenges of combining the generalization and learning capabilities of neural networks with the precision of symbolic logic. Traditional attempts at this integration have been hampered by difficulties in initial data acquisition, the reliability of undertrained networks, and the complexity of reusing and augmenting trained models. To overcome these issues, we introduce the COOL (Constraint Object-Oriented Logic) programming language, an innovative approach that seamlessly combines logical reasoning with neural network technologies. COOL is engineered to autonomously handle data collection, mitigating the need for user-supplied initial data. It incorporates user prompts into the coding process to reduce the risks of undertraining and enhances the interaction among models throughout their lifecycle to promote the reuse and augmentation of networks. Furthermore, the foundational principles and algorithms in COOL's design and its compilation system could provide valuable insights for future developments in programming languages and neural network architectures.

keywords: COOL; Programming Language; Neural-Symbolic Compilation System; U2N; Neural-Symbolic Layer.

*I am actively seeking a doctoral project opportunity. If my research interests align with your project and you would like to discuss potential collaboration, please feel free to contact me at coolang2022@qq.com. The ongoing project in this paper is hosted on GitHub and everyone is welcome to follow its evolution and contribute at https://github.com/coolang2022/COOLang.

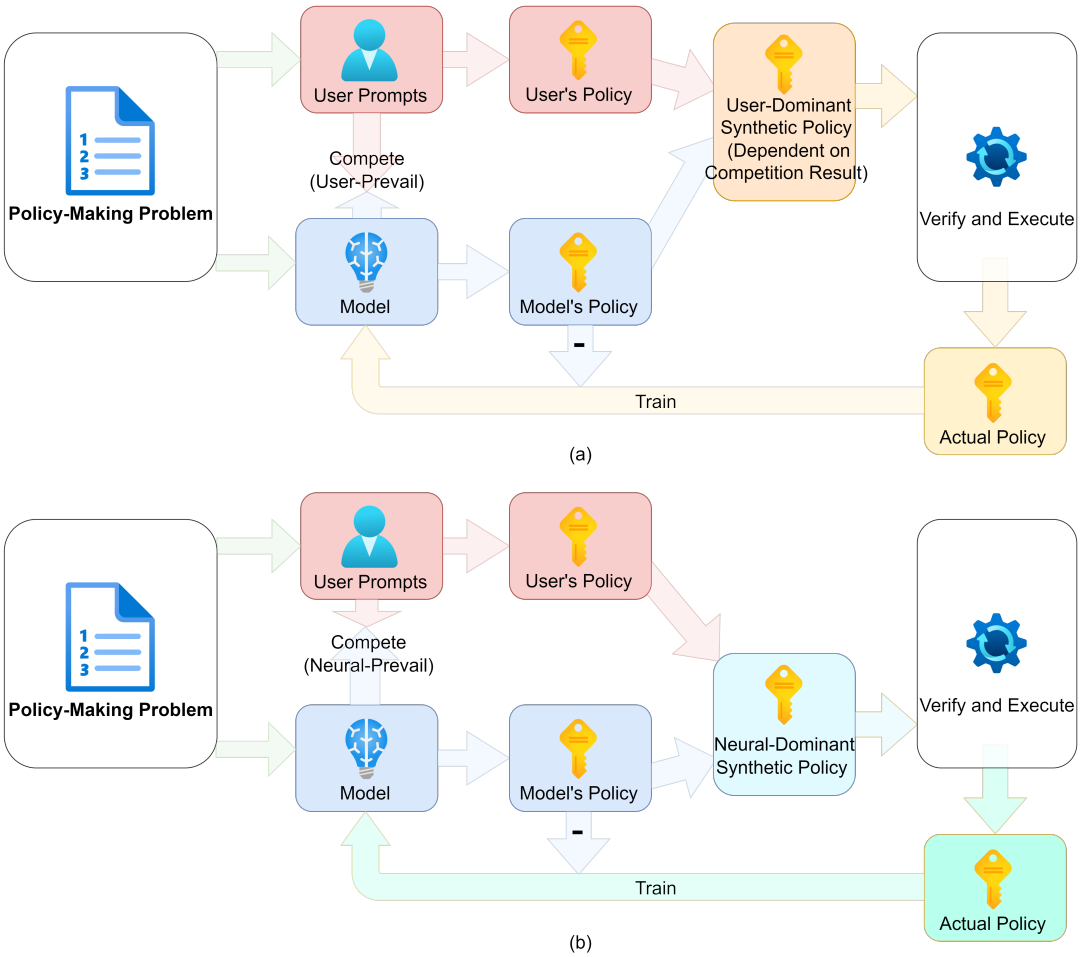

Figure 1: The U2N mechanism

2 Introduction

Neural networks have displayed excellent learning and generalization capabilities across diverse areas, such as classification (e.g., image[1] and speech recognition[2]), sequential analysis (e.g., natural language processing[3] and time series forecasting[4]), regression (e.g., predicting stock prices[5]), generation (e.g., art[6] and music creation[7]), etc. However, they often falter in calculation and formula manipulation. Therefore, the integration of neural networks and symbolic logical reasoning is important and promising, especially in cases that require both adaptability and precision, such as simulation, bioinformatics, fault diagnosis, and software engineering[8].

Nevertheless, this integration faces significant challenges in the context of

logic programming languages: 1) The scarcity of user and initial data for neural networks[9], 2) The low effectiveness of integrating undertrained neural networks with logical reasoning, and 3) The difficulty in reusing and expanding these hard-won neural network models.

Previous efforts, including neural-guided logical reasoning[10, 11, 12], neural network logic regularization[13, 14, 15, 16, 17, 18], neural-logic program induction[19, 20, 21, 22, 23], and logical neural network templating[24], expand the aspects of neural network and logic programming integration but fail to pay enough attention to and address these practical issues: initial data acquisition, reliability in reasoning, and model reusability. This paper takes a different path from other studies by concentrating on solutions to these practical problems.

To address these challenges, this COOL is designed based on the User-to-Neural (U2N) mechanism: Neural networks learn from users, compete with users, and gradually take over the policy-making process. As illustrated in Figure 1, both the user and the neural network model contribute to the policy-making process. Initially, as depicted in part (a) of the figure, when a neural network model is undertrained, user prompts are predominant, and thus the resulting policy will be user-dominant. The compiler will reason following the guidance of this user-dominant policy. However, since the user's prompts may not be flawless and generalized, the policy that the compiler ultimately implements might not align completely with the initially formulated policy. By identifying and analyzing the differences between its proposed policy and the adopted one, the neural network models refine their strategies, better positioning themselves for subsequent rounds of competition. Leveraging their innate ability to learn and generalize, as the training progresses, neural network models begin to outcompete the user prompts in shaping the policies. This transition to a neural network-dominant policy is depicted in part (b) of the diagram. Throughout this evolutionary process, the shift in control from user-dominant to neural network-dominant in policy-making is gradual and localized.

The U2N mechanism offers several advantages over conventional approaches. First, in comparison to the Teacher-Student Model[25], U2N eliminates the need for a specialized teacher, and its competitive mechanism enables the model to surpass the user, who acts as a teacher. Second, relative to Human-in-the-Loop[26], U2N requires the user to provide guidance only once, significantly minimizing interaction overhead. Finally, the U2N mechanism permits the immediate deployment of undertrained neural network models in a production setting, provided the entire system meets the requisite standards. Over time, the model will autonomously refine and enhance its capabilities.

Furthermore, as depicted in Figure 2, a novel layer for neural network architecture, termed the neural-symbolic layer, is proposed. This layer consists of rules grouped together and neural network models that manipulate them, aided by user prompts. Figure 3 illustrates the feedforward and backpropagation processes, which rely on both the U2N mechanism and the COOL compilation system's automated management of the entire lifecycle of neural network models, as well as their interactions with each other and the logic rules. Initially, as demonstrated in part (a), the layer possesses no neural network models. However,

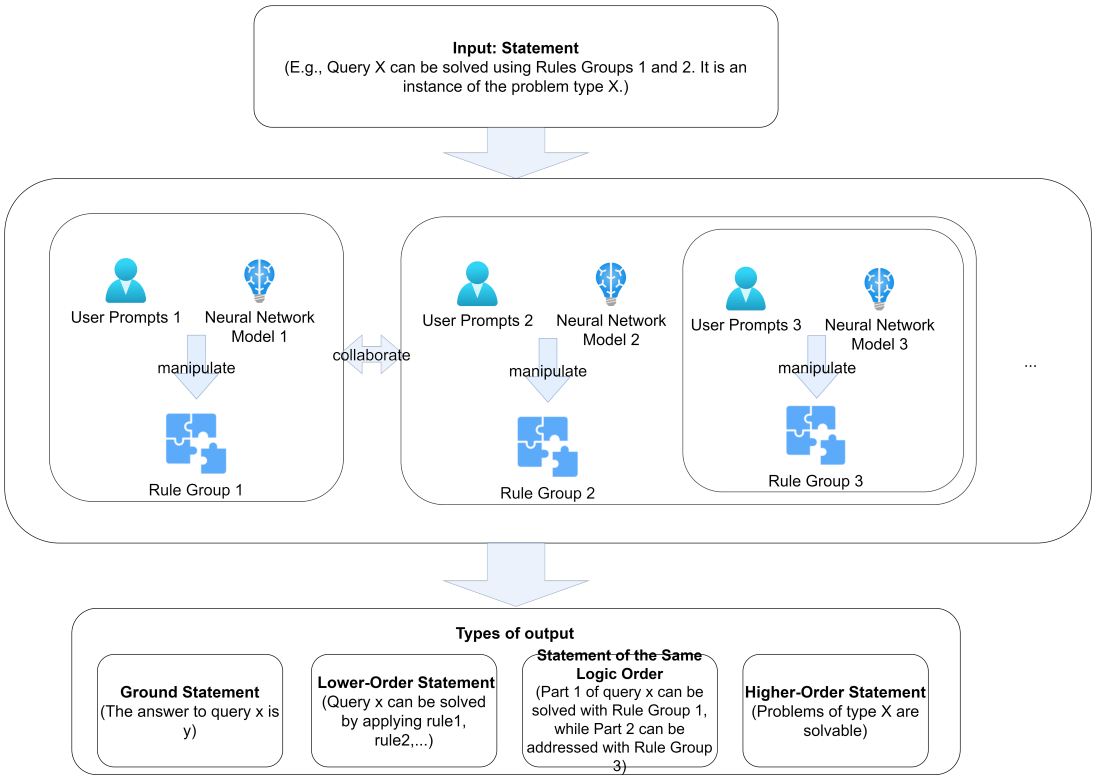

Figure 2: Neural-Symbolic Layer

over time, as depicted in part (b), the layer adaptively develops neural network models that work on one or more rule groups based on their utilization. The training status of each neural network model is not synchronous. Neural-symbolic layers emphasize logical reasoning. Incorporating neural-symbolic layers into neural network models can substantially enhance the logical reasoning capability of the entire architecture.

To summarize, this paper contributes to:

Programming Language Design: COOL, a novel logic programming language with distinct features is designed.

Neural-Symbolic integration: COOL's compilation system deeply integrates neural network and logical reasoning technology to achieve optimal coordination.

Code Synthesis: COOL's compilation system uses a novel algorithm for efficient code synthesis during its reasoning process.

Guided Learning: The U2N mechanism is a novel guided learning paradigm with its own advantages.

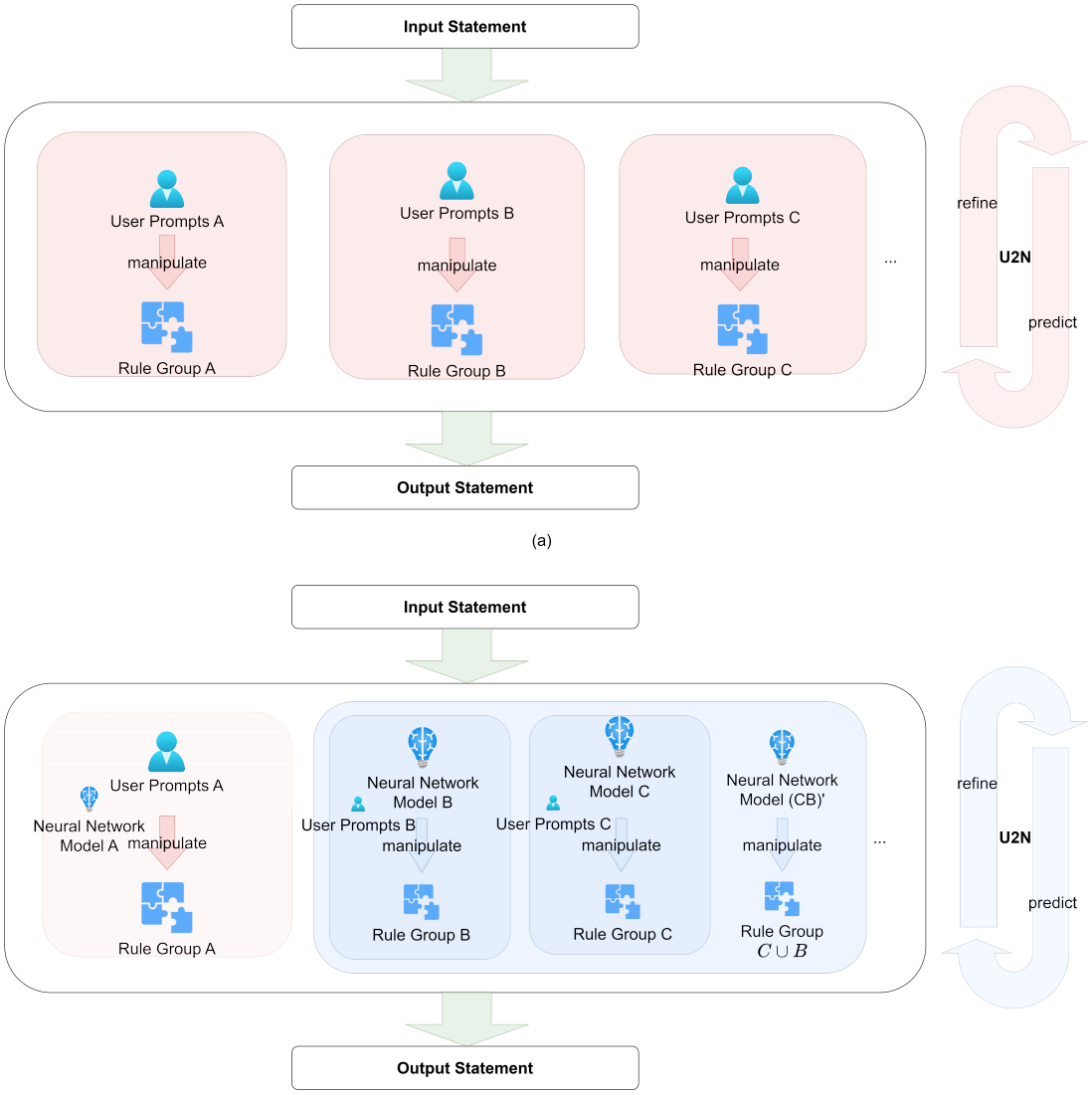

Figure 3: Evolution of Neural-Symbolic Layer

Neural Network Architecture: Neural-symbolic layer is proposed and is promising to promote the logical reasoning ability of neural networks significantly.

3 Related Work

3.1 Historical Overview

The integration of neural networks with logical reasoning has its roots in constraint logic programming. Since 1995, works like those by Bratko[27] focused on the expressiveness of constraint logic programming to address inherent challenges. Over time, the focus shifted towards utilizing neural networks as pattern recognition modules in hybrid logic programming systems like DeepProblog[16] and DeepStochLog[28]; or as prediction model to guide the reasoning process in logic programming system such as Neural Guided Constraint Logic Programming for Program Synthesis (NGCLPPS)[11], Neural Guided Deductive Search (NGDS)[29], Neural Program Search (NPS)[30] and Neural-Symbolic Program Synthesis (NSPS)[31]. Our research is closer to the latter.

3.2 Limitations of Existing Solutions

Solutions such as NGCLPPS, NGDS, NPS, and NSPS implement neural-symbolic integration largely adhering to the Powered By Example (PBE)[32, 33] paradigm. A prominent limitation of the PBE paradigm is its dependency on the initial dataset, making it less suited for logical programming scenarios with scarce data. In addition, for a large part of them, as DSLs (Domain-Specific Languages), their scalability and reusability have also been challenged.

3.3 Comparison with Existing Techniques

The COOL distinguishes itself from hybrid logic programming systems like DeepProblog, as well as neural-guided reasoning solutions like NGCLPPS, NGDS, NPS, and NSPS by leveraging the user's experiential knowledge in rule manipulation, automating the lifecycle of neural networks, and facilitating multi-model cooperation. Table 1, 2, and 3 offer a comprehensive comparison of COOL against existing techniques:

Framework	Base Language	User Contribution
DeepProbLog	ProbLog[34]	Dataset; Reasoning Rules
NGCLPPS	miniKanren[35]	Dataset
NGDS	PROSE DSL[36]	Dataset
NPS	LISP-inspired DSL	Dataset
NSPS	FlashFill DSL[37, 38]	Dataset
COOL	COOL (language)	Reasoning Rules; Reasoning Prompts

Table 1: Overview of Base Language and User Contribution

As illustrated in Table 1, The base language, COOL, is specially developed for the neural-symbolic system, aiming to provide a convenient environment for users to program with rules and prompts. This tailor-made language not only

better realizes its purpose of making full use of user knowledge but allows for greater expansibility compared with DSLs or other adapted languages.

Framework	Data Collection	Model Creation	Training, Testing and Updating
DeepProbLog	Manual	Manual	Manual
NGCLPPS	Manual	Manual	Manual
NGDS	Manual	Manual	Manual
NPS	Manual	Manual	Manual
NSPS	Manual	Manual	Manual
COOL	Auto	Auto	Auto

Table 2: Automation in Neural Network Model Pipeline

Table 2 showcases the degree of automation implemented by different frameworks throughout the stages of the neural network model lifecycle, including data collection, model creation, and the cycles of training, testing, and updating. While frameworks such as DeepProbLog, NGCLPPS, NGDS, NPS, and NSPS rely on manual processes at each stage, COOL distinguishes itself by fully automating the entire pipeline. COOL's comprehensive automation not only liberates users from repetitive tasks, enabling greater focus on development but also promotes consistency and improves the overall efficiency of managing neural network models.

Framework	Multi-Model Prediction	Multi-Model Training
DeepProbLog	Parallel	Parallel
NGCLPPS	Not Mentioned	Not Mentioned
NGDS	Not Mentioned	Not Mentioned
NPS	Not Mentioned	Not Mentioned
NSPS	Sequential	Sequential
U2N	Parallel; Composite	Contrastive

Table 3: Multi-Model Collaboration and Training

Table 3 presents the strategies employed by various frameworks for multi-model prediction and training. An effective collaboration strategy is crucial for improving the reusability and adaptability of neural models across a range of tasks. The U2N framework adopts a dual approach: models operate in parallel for distinct parts of a task, while for intersecting segments, their predictions are combined into a composite outcome. In addition, U2N employs a contrastive training method, which enhances the ability of each model to identify its unique function within a task. This method enhances data utilization and the overall efficiency of collaboration.

3.4 Advancements in Related Areas

Neural-symbolic integration and hybrid systems have experienced significant advancements in recent years. In this section, we review key developments that

are closely related to our work:

Integration of LPLs and GPLs: Several projects [39, 40, 41] have explored the integration of Logic Programming Languages (LPLs) with General-Purpose Languages (GPLs). Inspired by these efforts, we have devised an approach to seamlessly combine these two language paradigms in the development of COOL, which features enhanced rule expressiveness and syntactic constructs that cater to GPL users.

Neural-Symbolic Reasoning: Guiding logical reasoning with user prompts is a well-established practice. We have innovated by incorporating user prompts into the neural-symbolic reasoning process through the U2N mechanism, which is further integrated into the neural-symbolic layer designed for sophisticated logical reasoning.

Automation in Neural Network Model Lifecycle: COOL is designed to promote a transition from user-driven to neural-driven processes throughout the neural network model lifecycle. This comprehensive approach encompasses automated data acquisition, model development, training, testing, evaluation, and deployment. In this context, COOL broadens the application scenarios of and the implementation methods for relevant technologies like Zero-Shot Learning [42] and Self-Supervised Learning [43].

4 Fundamentals

This section introduces the U2N mechanism and the Neural-Symbolic layer, two foundational concepts in the COOL's neural-symbolic compilation system.

4.1 U2N Mechanism

As depicted in Figure 1, under the U2N mechanism, the user and the neural network model have a mentoring and competitive relationship at the same time. There are two primary attributes of the U2N mechanism: firstly, it permits the deployment of undertrained network models, and secondly, it enables neural network models to eventually surpass the user's performance, which acts as the initial teacher.

The key to achieving the first attribute lies in the competitive mechanism between the neural network and the user, ensuring that suboptimal policies are avoided and a minimum performance threshold is maintained post-deployment. This same mechanism also guarantees that the upper-performance limit is not restricted by user input. It is important to note, however, that while a neural network has the potential to eventually exceed user capabilities and tackle more intricate problems, its progression is incremental. As such, it is not designed to immediately address issues that are outside the scope of existing user policies.

The foundation of the second attribute builds on the first: the model's ability to be deployed while undertrained and actively participate in the policy-making

process allows it not only to learn from user strategy but also to adapt through direct environmental interaction, thus enabling the model to potentially surpass user expertise.

The User-to-Neural has two implications:

User-to-Neural Knowledge Distillation

Users store their knowledge in the form of prompts rather than data. These prompts implies users policy-making strategies. In each policy-making loop, models actively learn from the strategies. This emphasis on strategy-centered learning, promises an artificial intelligence paradigm more aligned with human cognition.

User-to-Neural Dominance in Policy-Making

The U2N policy-making process is initially user-dominated. this process becomes increasingly reliant on the neural network as it refines its capabilities. This change aligns human knowledge with machine potential effectively.

4.2 Neural-Symbolic Layer

A *neural-symbolic layer* is an architecture comprising multiple neural networks and an extensive set of grouped rules, where the neural networks manipulate the rules to facilitate formal transformations from one logical statement to another. As illustrated in Figure 2, a neural-symbolic layer can transform a logical statement into a ground statement (with no logical variable or undetermined variable, see in Section 5.1), a lower-order statement, a statement of the same logic order, or a higher-order statement [44].

It is noteworthy that when addressing deductive problems, the layer should aim to produce a statement of a lower logical order; conversely, for inductive problems, it should strive to yield a statement of a higher logical order. If the logical order of a statement experiences fluctuations through multiple layers, the problem itself may undergo inequivalent transformations. While such transformations should typically be avoided when resolving inductive or deductive issues, they may be beneficial in generative tasks.

4.3 COOL's Neural-Symbolic Compilation System

The COOL Neural-Symbolic Compilation System consists of a compiler and a neural network agent (introduced in Section 6), which orchestrates the model management. Figure 4 illustrates the compilation flow from source code to bytecode, with steps a through f representing the sequence of data transformation:

1. The compiler translates the source code into Intermediate Representation (IR), as shown in steps a and b.

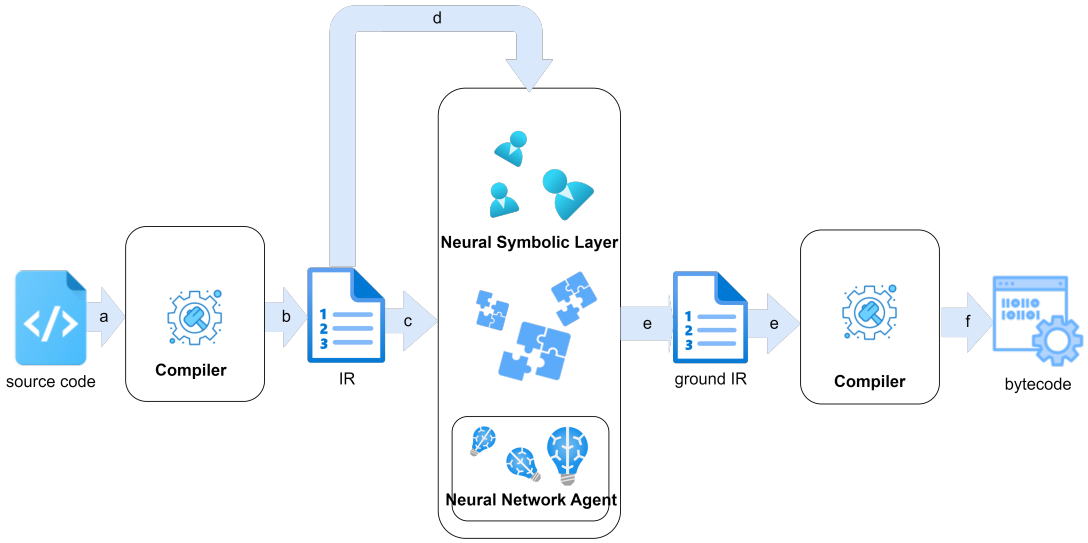

Figure 4: Neural-Symbolic Compilation Process

2. The IR, containing queries, rules, process control prompts, and domain-specific prompts (defined in Section 4), is fed into the neural-symbolic layer. Each query is paired with domain-specific prompts that delineate the rule groups applicable for resolving the query, depicted in step c. Process control prompts, indicating user strategies for rule manipulation during reasoning, are sent to the neural-symbolic layer in conjunction with the rules to establish the neural-symbolic structure, as indicated in step d.

3. The neural-symbolic layer translates the input IR into grounded IR (defined in Section 5.4), illustrated in steps c and e.

4. Finally, the grounded IR is compiled into bytecode, as indicated by steps e and f.

In this paper, while the COOL compiler is less central to the discussion and is not extensively covered, the logical reasoning components of COOL are thoroughly discussed in Section 5. The neural network agent, representing the neural network aspect, is detailed in Section 6.

5 COOL: Constraint Object-Oriented Logic Programming

This section introduces COOL's language and compiler, including its structures, distinct paradigms, intermediate representation, approach to program synthesis, and the grounding process. .

```
@add(a)to(b){ //Fact function 1, forward function
    b=b+a;
}
@add(a,b){  //Fact function 2, forward function
    return:a+b;
}
@{a+$x==b}{  //Fact function 3, inverse function
    x=b-a;
}
@{a+$x}{  //Fact function 4, inverse function
    x=ans-a;
}
new:x=0;
1+$x==2;  //Call the fact function 3
```

Code 5: Fact Function

5.1 Key Constructs in COOL

The subsequent content delves into the core constructs of COOL. Each item will be discussed with a structured approach: beginning with a definition, followed by an elucidation rooted in code examples, and culminating in a summary to encapsulate the essence of the structure.

1. **Fact Function:**

   Definition: A function that (1) produces either a simple value or has an implicit return, (2) includes at least one undetermined variable, and (3a) embeds undetermined variables within its name, termed an *inverse function*, wherein the function body explicitly presents a determined value denoted as "ans" for its return, or (3b) designates the undetermined variable as its return, referred to as a *forward function*.

   Example: (Code 5) Fact functions 1 and 2, although distinct in syntax, bear a resemblance to traditional GPL functions; notably, fact function 2 has an implicit return. Fact function 3 introduces an expression-based function name; the '$' adorned 'x' signifies an undetermined variable in the equation $a + x == b$. This notation enhances the expressiveness of function names. Fact function 4, has a similar functionality to fact function 3, but resorts to the return "ans" to compute the value of x.

   Essence: Despite the structural resemblance to GPL functions, fact functions diverge significantly in mathematical characteristics and expression evaluation order. The concise mathematical definitions of the fact function and the algorithm for deducing the appropriate evaluation order are presented in Definition 5.1 and Algorithm 5.1, respectively. In the scope of this research, undetermined variables are treated as *logic variables*, and expressions that include logic variables are categorized as *queries*. Under

Algorithm 5.1 Evaluation Order Deduction

Require: An expression $\mathcal{E}$ composed of fact function calls.
Ensure: An ordered sequence of function calls for evaluation.
Parse $\mathcal{E}$ from left-to-right[45] to produce $\mathbf{f} = [f_1, f_2, \ldots, f_n]$.
Partition f into:

$$\mathbf{f}_{\text{forward}} = [f_{\text{fwd},1}, \ldots, f_{\text{fwd,k}}]$$
$$\mathbf{f}_{\text{inverse}} = [f_{\text{ivs},1}, \ldots, f_{\text{ivs,l}}]$$

with the constraint $k + l = n$. Maintain relative order from $\mathbf{f}$ within both subsequences.
Construct the final evaluation sequence $\mathbf{f}^*$ as:

$$\mathbf{f}^* = \mathbf{f}_{\text{forward}} \parallel \text{reverse}(\mathbf{f}_{\text{inverse}})$$

In the context of this algorithm, $\parallel$ denotes the concatenation operation and reverse is the operation to reverse the sequence.
return $\mathbf{f}^*$

this conceptual framework and within the realm of logic programming, when a fact function is invoked, it leads to the instantiation of logic variables within the body of the function. Essentially, fact functions encapsulate the unification process in logic programming systems.

Definition 5.1.
Let f be a fact function. Formally, f can be defined as:

$$f : \mathbf{x} \mapsto y \tag{1}$$

where $\mathbf{x}$ denotes a set of input variables derived from the function's name and y is the resultant value of the function. Specifically, the function adheres to one of the two distinct formulations:

(a) $f : \mathbf{x} \mapsto y$ where $\mathbf{x}$ consists only of determined variables. f is termed as the *forward function*.
(b) $f^{-1} : (\mathbf{x}_d, y) \mapsto \mathbf{x}_u$ in which $\mathbf{x}_d$ and $\mathbf{x}_u$ represent the determined and undetermined parts of the input variables, respectively. f^{-1} is termed as the *inverse function*.

□

2. **Rule Function:**

Definition: A function returns an expression, distinctively marked by the 'expr' modifier. This function's return will be integrated at the point of invocation.

Example (Code 6): In rule functions 1, 2, and 3, the variable b is presented without a prefix, prefixed with `$` or prefixed with `#`, respectively. These

```
expr:@{ln(a*b)}{  //Rule function 1
    return:ln(a)+ln(b);
}
expr:@{ln(a*$b)}{  //Rule function 2
    return:ln(a)+ln(b);
}
expr:@{ln(a*#b)}{  //Rule function 3
    return:ln(a)+ln(b);
}
expr:@{(a) is the parent of (b) && (b) is the parent of (c)
    }{  //Rule function 4
    return: (a) is the parent of (b) && (b) is the parent of
         (c) && (a) is the grandparent of (c);
}
```

Code 6: Rule Function

denote b as a determined variable, an undetermined variable, or a variable that could be either, in that order. While rule functions share similarities with macros or inline functions, they differ in invocation; they are not directly callable. Rule functions do support recursive structures, as exemplified in Rule function 4.

Essence: In terms of logic programming, rule functions can be interpreted as functionally-represented rules.

3. **Constraint-Query Function Group:**

   Definition: A construct comprised of two essential parts: (1) a comprehensive fact function with solely determined variables in its name, and (2) at least one declarative fact function name containing one or more undetermined variables.

   Example (Code 7): The first part, referred to as the "constraint component," clearly defines the constraints and relationships between various variables. This includes relationships such as the one between quantity, unit price, and total price, or between initial speed, launch angle, and the landing point's distance. On the other hand, the second part, named the "query component", specifies which variables are to be queried. It's crucial that the names of these variables remain consistent with those in the constraint component. For instance, it determines the unknown quantity when the unit price and total price are fixed, or the undetermined initial speed when both the distance to the landing point and the launch angle are known. Significantly, these two distinct components can be invoked independently.

   Essence: The core capability of constraint-query function groups lies in their ability to facilitate inverse parameter computations[46, 47, 48] based

```
/*Constraint-query function group 1*/
@(a) kg of apples at (b) per kg costs{ //part 1, constraint component
    return:a*b;
}=>@($a) kg of apples at (b) per kg given costs; //part 2, query component
new:w = 0;
($w) kg of apples at (3) per kg given costs == 50; //call the part 2

/*Constraint-query function group2*/
@projectile distance with speed (v0) and angle (θ){  //part 1, constraint component
     v0x=v0*cos(θ);
     v0y=v0*sin(θ);
     t=2*v0y/g;
     return:v0y*t;
}=>@speed ($v0) at angle (θ) given distance;  //part 2, query component
new:v=0;
speed ($v) at angle (π/3) given distance == 1000;  //call the part 2
```

Code 7: Constraint-Query Function Group

on return values. From a logical programming perspective, constraint-query function groups can be viewed as functionally-represented constraint sequences and query clauses.

4. **User Prompts:**

   Definition: Syntactic constructs within COOL that serve as interfaces, enabling users to relay high-level computational instructions to the compiler. These are comprised of two categories: domain-specification prompts and process-control prompts.

   Domain-Specification Prompts (DSP): Constructs demarcating the range of permissible functions for invocation, including the inheritance mechanisms in object-oriented programming and file inclusion protocols.

   Process-Control Prompts (PCP): Constructs determining the circumstances under which a function can be invoked. These are represented by prefix arrays placed before function names (following the '@' symbol).

To better elucidate the underlying mechanics of PCP within the COOL compile, we present the following formal mathematical definitions and

principles:

Definition 5.2.
Let **pcp** be a vector, termed a Process-Control Prompt, given by:

$$\mathbf{pcp} = [pcp_1, pcp_2, \ldots, pcp_n] \tag{2}$$

where n signifies the total number of steps in the procedure.
Each element pcp_i in **pcp** is defined as:

$$pcp_i = \begin{cases} 0 & \text{if the associated function is not invokable at step } i \\ r_p & \text{if the associated function is invokable at step } i \text{ with reward } r_p \end{cases} \tag{3}$$

where r_p is a non-zero real number. □

Definition 5.3.
Let $\boldsymbol{\tau} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ be a set of steps, where each s_i represents a distinct phase in the problem-solving process as dictated by a user. The "step set" of a function associated with **pcp**, denoted as $\boldsymbol{\tau}_p$, is given by:

$$\boldsymbol{\tau}_p = \{\tau_i \mid pcp_i \neq 0\} \tag{4}$$

Hence, if $pcp_i \neq 0$ for a given **pcp**, the corresponding operation is permissible during the i^{th} step. □

Definition 5.4.
Let R_p denote the reward vector corresponding to **pcp**, with each r_{pi} (or equivalently pcp_i when non-zero) representing the reward associated with executing an operation at step i. Formally,

$$R_p = \{r_{pi} \mid r_{pi} = pcp_i \text{ and } pcp_i \neq 0\} \tag{5}$$

A higher magnitude of r_p (where r_p is an element of R) indicates a stronger user preference for executing the associated operation. □

Principle 5.1.
Let τ_m be the current step in the problem-solving process, and $\boldsymbol{\tau}_p$ be the step set derived from a **pcp**. A function bound with the **pcp** is executable if and only if:

$$\exists \tau_i \in \boldsymbol{\tau}_p : i \geq m \tag{6}$$

Define r_{um} as the reward for executing this function at step τ_m:

$$r_{um} = r_{ui}, \text{ where } i = \min(\{k | \tau_k \in \boldsymbol{\tau}_p \text{ and } k \geq m\}) \tag{7}$$

After execution, update τ_m to:

$$\tau_m \leftarrow \tau_k, \text{ where } k = \min(\{k | \tau_k \in \boldsymbol{\tau}_p \text{ and } k \geq m\}) \tag{8}$$

```
#load(io);  //Domain-specification prompt
class: Quadratic Equation Solver{
    exp:@(-2,0,0){$a==b}{  //Process-control prompt
        return:a-b==0;
    }
    exp:@(0,1,0){#a-b}{  //Process-control prompt
                return:a+(-b);
    }
    @(0,0,2){ a*$x^2+b*x+c==0 }{  //Process-control prompt
        x=(-b+(b^2-4*a*c)^0.5)/(2*a);
    }
}
class : Main << Quadratic Equation Solver{ //Domain-
    specification prompt
    new:x=1;
    1*$x^2+4*x==100;
};
Main:m;
m.x-->screen;
```

Code 8: User Prompts

For a complete problem-solving journey, the terminal step must be included:

$$m_{final} \in \boldsymbol{\tau}_p, \text{ where } final = \max\{i | \tau_i \in \boldsymbol{\tau}_p\} \tag{9}$$

□

Example (Code 8): The provided code segment aptly demonstrates the implementation of DSPs and PCPs. DSPs, such as "#load(io)" and "Main << Quadratic Equation Solver" (indicating that "Main" inherits the "Quadratic Equation Solver"), delineate functions that are permissible for use. DSPs play an essential role in problem resolution and must be provided by the user during programming.

Consider the three PCPs in class "Quadratic Equation Solver" :

$$\mathbf{pcp}_1 = [-2, 0, 0] \tag{10}$$

$$\mathbf{pcp}_2 = [0, 1, 0] \tag{11}$$

$$\mathbf{pcp}_3 = [0, 0, 2] \tag{12}$$

Associated with the PCPs are three distinct operations:

(a) $\mathbf{pcp}_1$ - Moving the right side of the equation to the left.
(b) $\mathbf{pcp}_2$ - Converting subtraction to addition.
(c) $\mathbf{pcp}_3$ - Resolving the equation in its standardized form.

From $\mathbf{pcp}_1$, it is evident that the first preferred action is to transpose terms from the equation's left side to the right at the 1^{st} step with a reward of -2. The negative value suggest the user's intention to minimize or be cautious about this operation, because this operation only needs to be performed at most once. Performing it repetitively will make the equation more complex.

$\mathbf{pcp}_2$ signifies that during the second step, the user prefers converting subtraction operations into additions, as evidenced by a reward of 1. Although this step streamlines the process, it doesn't constitute the key to the solution.

Conversely, $\mathbf{pcp}_3$ conveys that at the 3^{rd} step, there's a strong inclination to solve the equation in its standard form, as seen by the relatively high reward of 2.

If the compiler has currently advanced to the 2^{nd} step, only the operation connected with $\mathbf{pcp}_2$ and $\mathbf{pcp}_3$ can be executed next, ensuring the user's intentions are respected in the solution process.

Essence: From the perspective of logic programming, user prompts are a high-level kind of constraint on rules. Domain specification prompts ascertain the rules feasible for the grounding process, and process control prompts elucidate the methodology these rules should follow during said process.

These foundational elements of COOL are pivotal in comprehending the advanced concepts discussed in the subsequent chapters.

5.2 COOL's Programming Paradigm

The COOL language combines elements of GPLs like C++ and Java while adopting dynamic typing similar to Python. It seamlessly merges imperative, declarative, procedural, object-oriented, and functional paradigms. This integration endows COOL with the modularity, flexibility, and ease of use commonly associated with GPLs. As shown in Code 5, 6, 7, and 8 , COOL distinguishes itself from traditional logic programming languages, especially ones similar to ProbLog and its offshoots[34, 49, 50]. Unlike these languages where rules and sequenced constraints prevail, COOL interprets relations and constraints as functions to center its paradigm.

The evolution of this paradigm is rooted in key requirements from users and reasoning.

User requirements: A prevailing dilemma in the realm of logic programming, especially within constraint logical reasoning, is the dissonance between its theoretical design and its practical applications. While declarative rules and constraints are conceptualized to facilitate users' logical reasoning processes, the often opaque nature of rule interactions can lead novices to overlook rules or introduce flawed constraints. Thus, to effectively harness the capabilities of logic programming languages, users must deeply understand the nature of

Code Type	LHS	RHS	Operator	Result	Attribute Flags			
					LHS	RHS	Operator	Result
1				1	0	0	0	0
4	a	b	COMMA	2	2	2	2	0
4	add_ARG_to_ARG_	2	CALL	ans	3	0	2	2
2				1	0	0	0	0
6	1	100	@	1	0	1	2	0
1				6	0	0	0	0
4	b	a	+	7	2	2	2	0
4	b	7	=	b	2	0	2	2
5					0	0	0	0
2				6	0	0	0	0
6	1	6		1	0	0	0	0

Table 4: Intermediate representation for the fact function 1 in Code 5

the problem and the steps involved in problem-solving. Such languages fall short in providing a gradient environment, where users can gradually familiarize themselves with, and master, the problem-solving process. Consequently, many gravitate towards more intuitive GPLs, reflected in the teaching process of ProLog[51]. After overcoming challenges with a GPL, the allure to revert to an LPL diminishes, especially as most LPLs are niche Domain Specific Languages (DSLs). This dynamic suggests a likely predilection among users to prioritize proficiency in one or several GPLs over specializing in diverse LPLs. Notably, COOL is designed to offer holistic support for users with varying preferences and proficiencies. Users can initially engage with it as a GPL and, over time, delve deeper to harness the principles of Logic Programming to address challenges.

Reasoning requirements: Through the introduction of GPL-style functions, the expressiveness of constraints and rules is elevated. This allows users to formulate more intricate constraints and rules to represent and tackle problems. With this advanced descriptiveness, previously exclusive to GPLs, COOL becomes capable of reasoning through more complex issues. Besides, Opting for functions instead of clauses provides the flexibility to embed additional information into the rules, such as the PCPs, which act as function attributes. Incorporating PCPs streamlines the logical reasoning process. This enhancement not only improves scalability but also significantly reduces the potential of encountering infinite loops due to cyclical rules[52].

5.3 Intermediate Representation

In the domain of compiler design and intermediate representations (IR), COOL utilizes the three-address code (TAC) as depicted in Table 4. Historically prevalent in GPLs [53], TAC offers a structured and concise format adept at delineating complex operations. Its design facilitates ease in parsing and

Algorithm 5.2 Code synthesis

Let IR be an array of instructions.
Let $IR_{segment}$ be a segment of the array IR corresponding to a query.
A ground $IR_{segment}$ has each instruction bound with a fact function.
while ungrounded $IR_{segment}$ exists **do**
  $\mathbf{f} \leftarrow$ {invokable functions from DSPs}
  $ground\_flag \leftarrow$ False
  repeat
    Apply $f_{rule} \in \mathbf{f}$ or bind $f_{fact} \in \mathbf{f}$ to $IR_{segment}$ based on **principle** 5.1
    if $IR_{segment}$ is ground **then**
      Update original $IR_{segment}$
      $ground\_flag \leftarrow$ True
      break
    end if
  until resources exhausted or $ground\_flag$ is True
  if not $ground\_flag$ **then**
    Error Exit
  end if
end while
Success Exit

subsequent modifications. With the one-to-one correspondence of each TAC instruction to a node in the rules and constraint trees, TAC is an ideal intermediate representation for COOL and an optimal data format for further neural network modeling.

5.4 Program Synthesis in COOL

Program synthesis in the context of COOL involves the transformation of COOL's Intermediate Representation (IR) from a hybrid declarative-imperative paradigm to an exclusively imperative form. This methodology echoes the strategies employed in Structured Query Language (SQL) and Halide, wherein physical expressions are generated[54] and specific schedules are determined[55], respectively.

Program synthesis in COOL is tied to its *grounding* process. Originating from the logic programming domain, in which 'grounding' denotes the conversion of terms with logical variables into concrete ground facts, the term takes on a nuanced meaning in COOL. Here, 'grounding' characterizes the transformation of a query into fact function invocations. A comprehensive description of this transformation process is provided in algorithm 5.2.

Program synthesis is indispensable for COOL's execution efficiency. Unlike conventional logic programming languages, such as ProLog, which perform grounding during execution, COOL separates the grounding phase from the execution phase. Owing to COOL's design that gathers queries, DSPs, and PCPs throughout the programming phase, the reasoning process does not re-

quire additional query or instructional input from interaction with users during execution. Therefore, integrating the grounding process within the compilation phase becomes a logical step. This integration ensures that, during the actual execution of the program, the time complexity parallels the efficiency standards of GPLs. A similar approach is adopted in Answer Set Programming (ASP), which generates ground programs before solving, thus reducing the problem to a propositional level[56].

5.5 U2N Guided Grounding Process in COOL

The purpose of the grounding process is to translate queries into specific fact function invocations. This involves working with the Abstract Syntax Tree (AST). Initially, the AST is composed of nodes that are not bound to any fact function. The compiler's role is to iteratively apply rule functions to modify the AST structure and bind fact functions to specific subtrees. A function can be applied or bound to a specific subtree only when the name of the function shares the same structure as the subtree, and this operation affects all nodes in the subtree simultaneously. The ultimate goal is for all nodes to be bound with fact functions, with each node being bound to only one fact function.

Within each iteration, the compiler faces a crucial decision: to which subtree within the AST should which function be applied or bound? In the context of IR, this decision pertains to associating specific functions with particular sub-segments of the IR.

Given that the grounding state of each IR segment is independent of its preceding state, the success of converting a generated IR segment state during grounding is unaffected by any previous state. This characteristic renders each iteration of the grounding process as a distinct Markov Decision Problem. Consequently, the entire grounding procedure can be described as a sequence of Markov Decision Processes (MDPs)[57].

But in practice, four important issues make the use of MDPs less feasible:

Future Reward Calculation: Given the myriad of potential combinations of IR segments and functions, and the unfeasibility of remembering each one, the compiler can only determine possible actions for a subsequent state after observing the current IR segment's state. This makes it infeasible to calculate the cumulative reward for future actions.

Backtrack Mechanism Necessity: During the grounding process, it's not uncommon for an IR segment to reach an undesirable endpoint. Employing a backtrack mechanism can prevent repeated mistakes and conserve resources that would otherwise be wasted on re-grounding.

User Constraints: While states in grounding are inherently independent, constraints imposed by users on the application order, preferences, and aversions through PCPs indirectly induce sequences in specific grounding states. As a result, states with higher accumulated rewards from the initial phase, relative to those with lower rewards, tend to be nearer to the successful conclusion of the

grounding process or are more likely to complete the grounding with reduced resource consumption. Utilizing this information can hasten the identification of feasible solutions. However, it might challenge the fundamental Markovian structure of the process.

Step Skipping in Grounding Process: As indicated by principle 5.1, grounding in line with PCPs allows for step skipping. However, bypassing an essential step on occasion can make an IR state ungroundable. Simultaneously, jumping directly to a later step can sometimes produce a significantly larger reward compared to regular states. This can cause the compiler to allocate considerable resources, only to realize that the state is ultimately infeasible. Addressing this challenge necessitates an appropriate mechanism.

5.5.1 Bi-Direction Discounted Backtracking

To address these challenges effectively, the COOL compiler introduces the Bi-Direction Discounted Backtracking (BDDB) algorithm, an evolution from traditional MDPs. This algorithm takes cues from Eligibility Traces[58] and Offline Reinforce Learning[59] in terms of processing and modeling with historical data, but there are significant distinctions between them. In the algorithm's name, "Bi-Direction" emphasizes its consideration of both past and future states. "Discounted" points to the use of discount factors, and "Backtracking" indicates the algorithm's ability to revisit prior states. In the BDDB algorithm 5.3 presented, the notation $q(s, a)$ is employed to denote a value associated with a state-action pair. This is not to be confused with the Q-value from Q-learning.

5.5.2 Elucidation of the BDDB Algorithm

The BDDB algorithm provides a structured approach for problem-solving in scenarios characterized by a vast or infinite state space. The core components of the algorithm are:

State Space (S): Starting with the initial state s_0, this space expands to incorporate states identified during the algorithm's run. For COOL's grounding, a state symbolizes an IR segment, with s_0 being the original IR segment without any instruction bound to a fact function.

Action Space (A): Comprising valid actions a_i, which symbolize potential decisions in any state. In the context of COOL's grounding, an action signifies the act of applying or binding a function to an IR sub-segment, hence the viable actions are constrained by the state s.

Transition Function (T): This manages system development by mapping a state-action pair (s_t, a_t) to the succeeding state s_{t+1}. Within COOL's grounding framework, the Transition Function equates to the compiler.

Policy (π): This stochastically maps actions to states, steering the system's operations. In COOL's grounding, the system employs batch processing for

Algorithm 5.3 BDDB Algorithm

Require: $S, A, T, r, \pi, \gamma, \lambda, l, q_{base}$

Initialize:

State Space $S = \{s_0\}$ (initially, will expand as more states are discovered)

Action Space $A = \{a_1, \ldots, a_n\}$

Transition Function $s_{t+1} = T(s_t, a_t)$

Stochastic Policy $\pi(a_t|s_t)$

Reward Function $r(s_t, a_t, s_{t+1})$

Action-Value Function

$$q^{\pi}(s_t, a_t) = \begin{cases} \mathrm{E}_{\pi}\left[\sum_{k=0}^{l} \gamma^k r_{t+k+1} \mid s_t, a_t, k \leq l\right] + r(s_t, a_t, s_{t+1}) + \lambda q^{\pi}(s_{t-1}, a_{t-1}) & \text{if } t \geq 1 \\ r(s_0, a_0, s_1) + \lambda q_{base} & \text{if } t = 0 \end{cases} \tag{13}$$

Execute:

Action-Value Space $Q : \{q^{\pi}_{t,a_t} \rightarrow (s_t, a_t)\}$ initialized with $(q^{\pi}(s_0, a_0), (s_0, a_0))$

while exit conditions not met **do**

  Select (s_t, a_t) with highest q from Q

  Compute s_{t+1} via T and corresponding $q^{\pi}(s_{t+1}, a_{t+1})$

  if s_{t+1} meets success exit conditions **then**

    Output optimal action sequence $[a_0, \ldots, a_t]$ for starting state s_0

    Update dataset **D** for further modeling

    Terminate

  end if

  Remove $(q^{\pi}(s_t, a_t), (s_t, a_t))$ from Q and add all $(q^{\pi}(s_{t+1}, a_{t+1}), (s_{t+1}, a_{t+1}))$

end while

policy creation. Upon receiving an IR segment and a set of accessible domains **d**, which comprises accessible file and class names defined by domain specification prompts (DSPs) to constrain the feasible action range, the neural network agent responds with policies and two coefficients:

$$(\Pi, ac, ci) = \text{agent}(s, \mathbf{d}) \tag{14}$$

1. Policy array Π is defined as $\Pi = [\pi_1, \pi_2, \ldots, \pi_n]$, where n signifies the IR segment's length and π_i denotes the likelihood that a function can be applied or bound to the sub IR segment rooted at the node specified by the i^{th} IR segment instruction. Notably, function specifics at sub-segments are not determined at this stage, rendering π_i a general policy for all suitable potential actions.
2. ci represents the confidence with which the agent perceives the IR segment to be within its knowledge domain, specifically in relation to **d**.
3. ac denotes the agent's accuracy of neural network predictions, particularly in relation to the neural network's performance on the test set.

Compared to related frameworks, this processing approach considerably enhances efficiency. In works such as [11, 29], the neural network model must inspect the synthesized codes to evaluate them. Subsequently, it employs the neural network agent to iteratively score all produced codes. This method incurs significant computational overhead, with a complexity of $O(k \times n \times m)$, where k signifies the average actions executed for a grounding/code generation process, n represents node count, and m indicates average rules/operations applicable to a node. In contrast, COOL's neural network omits the need to inspect actual a_t and $s_{t+1} = T(s_t, a_t)$, leveraging s and $\mathbf{d}$ directly to produce π for each node of s in batches. Consequently, its complexity is a mere $O(k \times m)$.

Reward Function (r):

This function provides immediate feedback subsequent to the transition from state s_t to s_{t+1} post the execution of action a_t. Within COOL's grounding, the U2N mechanism in the compiler operationalizes the reward function. Based on this mechanism, immediate rewards are derived from both the user and neural network agent, transitioning from user-dominant to neural-dominant. The reward function in grounding is given as:

$$r(s, \pi, r_p) = (1 - ac * ci) * ((r_p - r_a) * (1 - ac * ci) + r_a) \tag{15}$$

$$+ ac * ci * \pi * r_a + o \tag{16}$$

with the following components:

1. r_p is the reward provided by the associated process control prompt (PCP).
2. r_a is the reward from the neural network agent, used to counteract the user's guidance effect on the grounding process and impose the neural network agent's guidance effect:

$$r_a = \max\{k_0 * |r_p|, r_{a_{base}}\}, \text{where } k_0 \in (0, 1], r_{a_{base}} \geq 0 \tag{17}$$

3. o is an offset value. It serves as a reward for advancement aligned with steps specified by the PCPs or a gradually increasing penalty for stagnation:

$$o = \begin{cases} k_0 & \text{if } t = 0 \\ -k_1 * k_2^t & \text{if } t > 0 \end{cases} \text{where } k_0, k_1, k_2 > 0 \tag{18}$$

Action-Value Function (q): Denotes the anticipated cumulative reward when selecting action a_t in state s_t and adhering to policy π subsequently. This function amalgamates immediate and future rewards and embodies the "memory" concept by accounting for the value of prior state-action pairs. Two constants, $\gamma \in [0, 1]$ and $\lambda \in [0, 1]$, serve as discount and decay factors, respectively. Specifically, γ attenuates the influence of future rewards, while λ progressively lessens the impact of past rewards. In the COOL's grounding context, $\gamma = 0$. This indicates that the compiler does not factor in future rewards, primarily because it cannot preemptively ascertain such rewards by

bypassing time steps. On the other hand, with $\lambda \in (0, 1]$, the compiler permits previously accrued rewards to inform and guide the grounding process. This approach harnesses the latent sequential information inherent in the grounding state. Moreover, by allowing the influence of past rewards to wane progressively, the method mitigates potential risks associated with skipping steps.

Action-Value Space (Q): This stores the relation between value and state-action pairs. It aids in pinpointing actions leading to maximum anticipated rewards and offers a mechanism to backtrack when a promising state-action pair leads to a less favorable outcome.

Execution: The algorithm, during its execution, explores the state-action space iteratively to uncover the optimal action sequence from the initial state s_0. The steps include:

1. Choosing the state-action pair in Q with the highest q value.
2. Ascertain the subsequent state via the transition function and calculate its associated action-value.
3. If state s_{t+1} meets predetermined success criteria, the algorithm renders the optimal action sequence from s_0. For COOL's grounding, the output is state s_{t+1}.
4. Update Q by excluding the currently evaluated state-action pair and incorporating the newly computed pairs.
5. Exploration creates a dataset, $\mathbf{D}$, which documents pertinent data from explored state-action pairs. This data can serve future modeling.

In the context of COOL's grounding process, the arrays

$$[(s_0, \text{root}(a_0), (\delta_\pi \mid (s_0, a_0))), \ldots, (s_n, \text{root}(a_0), (\delta_\pi \mid (s_n, a_n)))] \tag{19}$$

and

$$\mathbf{d}' \mid [a_0, ..., a_n] \tag{20}$$

are assembled as modeling data g. Here, the function root extracts the instruction position in s, where the instruction serves as the root of the sub-segment upon which the action operates. The policy error, denoted by

$$\delta_\pi = 1 - \pi \tag{21}$$

acts as a correction signal during modeling. The effective domain set $\mathbf{d}'$, a subset of $\mathbf{d}$, represents the union of domains encompassing functions employed in $[a_0, ..., a_n]$.

Specific Scenarios:

– For $\lambda = 0$, BDDB mimics MDPs.

- For $\lambda = 1$ and $\gamma = 0$, BDDB acts as Breadth-First Search (BFS) with consistent negative rewards and as Depth-First Search (DFS) with positive rewards.
- When $ci * ca = 0$, user dominance prevails in the grounding process.
- At $ci * ca = 1$, the grounding process only adheres to the steps prescribed by the PCPs based on principle 5.1, with all other guidance being provided by the neural network agent. In this case, it is recommended not to disable PCPs to retain the user's most basic control over the grounding process.

6 Neural Network Agent

This chapter details the neural network agent, including data collection by the COOL compiler, model creation, training, testing, updating, and the prediction process.

6.1 Data Collection

The data collection is required by the neural network agent but integrated into the COOL compiler. When data collection is enabled by default via the settings file, each grounding process becomes a potential data source.

6.1.1 Procedure

Data Generation: During each grounding process in the compilation, the compiler generates modeling data denoted as g.

Domain Grouping: Modeling data with the same effective domain set are grouped into the same knowledge domain. Both the effective domain set and corresponding knowledge domain are denoted as $\mathbf{d}'$.

File Storage: Data from the same knowledge domain, generated within a given collection cycle (e.g., a week), are stored in a single file. The file is defined as $\mathbf{F} = \{g \mid \mathbf{d}' \in g \text{ and all } g \text{ have the same value for } \mathbf{d}'\}$.

Indexing: Each file $\mathbf{F}$ is indexed by its associated $\mathbf{d}'$.

Table Updating: The relationship between these indices $\mathbf{d}'$ and their corresponding files is documented in table I. The table is defined by $I : \{\mathbf{d}' \mapsto \mathcal{F}\}$, where $\mathcal{F} = \{\mathbf{F}_t \mid t \text{ indicates the serial number of a collection cycle, } t \in \mathbb{N}\}$.

Notifying the Agent: After compilation, updates to the table I are relayed to the neural network agent for processing.

6.1.2 Considerations

Privacy: Users desiring confidentiality can modify settings to inhibit data gathering, offering them substantial discretion over their compilation datasets.

Data Integrity: Sourced directly from the compilation process, the compiler affirms the genuineness of the datasets. This direct sourcing obviates the necessity for supplementary verification post-collection.

Granular Recording: Datasets are meticulously categorized based on specific knowledge domains, contrasting with a broader classification by project. This differentiation ensures that varying problem-solving strategies are not conflated.

Data Storage: Datasets from distinct collection cycles are stored separately. Users possess the option to erase previously gathered data to conserve storage space.

Incremental Modeling: During multiple modeling sessions within a single collection cycle, only the most recent segment of the dataset is deemed as *new*. Previously utilized data within current or earlier cycle is considered *old* data, promoting a more streamlined and updated learning process.

6.2 Construction of Composite Datasets

Upon receiving the information about the generation of new data from compilations, the neural network agent undertakes the task of creating datasets for training and testing. Constructing datasets holds paramount significance for the neural network's performance. The dataset amalgamates new data with historical data. Concurrently, datasets corresponding to different knowledge fields are negatively sampled from each other for contrastive training [60].

In this section, unless explicitly stated, all sets are standard sets devoid of duplicated elements. When a multiset merges with another multiset or set, duplicate elements persist.

6.2.1 Construction of the Positive Sampling Training Dataset

A modeling dataset, $\mathbf{F}$, is partitioned into a training subset ($\mathbf{F}^{train}$) and a testing subset ($\mathbf{F}^{test}$) based on a predefined ratio during its initial utilization for modeling. The positive sampling training dataset, denoted as $\mathbf{G}_{\mathbf{d}'}^{train}$ (a multiset), is formulated as:

$$\mathbf{G}_{\mathbf{d}'}^{train} = (\mathbf{G}_{\mathbf{d}'}^{train,new} \cup \mathbf{G}_{\mathbf{d}'}^{train,old}) \mid \{\mathbf{F}_{\mathbf{d}'}^{train}\} \tag{22}$$

The constituents of this dataset include:

1. **New Training Dataset** ($\mathbf{G}_{\mathbf{d}'}^{train,new}$): This multiset includes the most recent data produced by the compiler. Its inclusion in the training process is crucial. On one hand, this fresh data can help improve the performance of models that previously lacked sufficient training. On the other hand, this

dataset reflects the most recent concept drift[61], encompassing updates in knowledge domains—such as the modification of corresponding files and classes—and addressing new queries stemming from emerging real-world requirements that the neural network must adjust to. The formulation of $\mathbf{G}_{\mathbf{d}'}^{train,new}$ incorporates a re-sampling method (specifically, oversampling) [62] to enhance data variability and representation. Notably, this re-sampling approach doesn't prioritize data by weight; instead, it's grounded in their policy error, δ_π.

For $(s, \text{root}(a), (\delta_\pi \mid (s,a))) \in \mathbf{F}_{\mathbf{d}'}^{new,train}$, the frequency of duplication is determined as:

$$n_{oversample} = \max\left\{\lceil n_{max} \times \frac{\delta_\pi - \delta_{tolerance}}{1 - \delta_{tolerance}} \rceil, 0\right\} \tag{23}$$

where:

n_{max} represents the maximum permissible oversampling count.

$\delta_{tolerance} \in [0, 1)$ is the paramount policy error that does not instigate duplication.

Further, the new training dataset is constructed as:

$$\begin{aligned}\mathbf{G}_{\mathbf{d}'}^{train,new} = \bigcup_{i=1}^{|\mathbf{F}_{\mathbf{d}'}^{new,train}|} & \{n_{oversample,i} \times (s_i, \text{root}(a_i), InDom_i) \\ & \mid (s_i, \text{root}(a_i), \pi_i) \in \mathbf{F}_{\mathbf{d}'}^{new,train}\}\end{aligned} \tag{24}$$

where $InDom_i$ signifies whether $(s_i, \text{root}(a_i))$ is within the knowledge domain $\mathbf{d}'$. Being a positive sampling process, $InDom_i$ is invariably true.

2. **Old Training Dataset** ($\mathbf{G}_{\mathbf{d}'}^{train,old}$): This dataset aggregates data samples from previous datasets in this knowledge domain. Incorporating this dataset pursues two essential objectives. Firstly, it aims to stabilize accuracy fluctuations. Relying exclusively on the new data could lead to significant variations in accuracy metrics, potentially compromising the agent's consistent performance. Drawing samples from historical data can mitigate such fluctuations. Secondly, it serves a crucial role in counteracting data drift[61]. While models focused on a narrow knowledge domain might be relatively immune to data drift, those spanning a broader or composite knowledge domain (a domain derived from multiple others) might be susceptible to biases when perpetually trained on unbalanced datasets. This ensures that the neural network sustains a balanced allegiance across the entirety of the knowledge domain. Additionally, the old training dataset integrates a temporal weighting re-sampling strategy, specifically undersampling, to recalibrate the representation of prior training data based on the elapsed time since their initial sampling and the present moment.

The old training dataset, $\mathbf{G}_{\mathbf{d}'}^{train,old}$, is defined as:

$$\mathbf{G}_{\mathbf{d}'}^{train,old} = \bigcup_{t=m-l}^{m-1} \mathscr{G}_{\mathbf{d}',t}^{train,old} \tag{25}$$

where:

m denotes the ordinal of the current collection cycle.

l represents the length of the sliding window, which is a positive integer indicating the span of permitted historical data.

$\mathscr{G}_{\mathbf{d}',t}^{train,old}$ is the dataset undersampled from $\mathbf{F}_t^{train,old}$.

The cardinality of $\mathscr{G}_{\mathbf{d}',t}^{train,old}$ follows:

$$\mid \mathscr{G}_{\mathbf{d}',t}^{train,old} \mid = \psi^{m-t} \min\{\mid \mathbf{F}_m^{train,new} \mid, \mid \mathbf{F}_t^{train,old} \mid\} \tag{26}$$

where $\psi \in [0, 1]$ stands for the Sequential Attrition Undersample Rate (SAUR). It quantifies the proportion of data retained from older datasets when merging them sequentially with the newest dataset.

6.2.2 Construction of the Negative Sampling Training Dataset

For two training datasets, $\mathbf{G}_{\mathbf{d}_1'}^{train}$ and $\mathbf{G}_{\mathbf{d}_2'}^{train}$, they are mutually negative when $\mathbf{d}_1' \cap \mathbf{d}_2' = \varnothing$.

The negative sampling training dataset, $\mathbf{G}_{\perp,\mathbf{d}'}^{train}$, adheres to:

$$\begin{cases} \mathbf{G}_{\perp,\mathbf{d}'}^{train} & = \{(s_i, InDom_i) \mid \exists (s_i, \text{root}(a_i), \text{True}) \in \text{dedup}\left(\bigcup \mathbf{G}_{\mathbf{d}'_\perp}^{train}\right) \\ & \quad \text{and } InDom_i = \text{False for negative sampling}\} \\ \mid \mathbf{G}_{\perp,\mathbf{d}'}^{train} \mid & = \phi \mid \mathbf{G}_{\mathbf{d}'}^{train} \mid \end{cases} \tag{27}$$

where:

dedup is the deduplication procedure.

$\mathbf{d}'_\perp$ delineates the mutually exclusive counterpart of $\mathbf{d}'$, signifying two distinct knowledge domains.

ϕ represents the negative sample rate, determining the negative sample proportion relative to positive samples.

Constructing a training dataset using negative sampling is essential for models that engage in multi-domain knowledge collaboration. This approach enhances the model's proficiency in differentiating tasks specific to its knowledge domain, thereby enabling it to concentrate more effectively on its own tasks during collaboration.

6.2.3 Finalization of the Training Dataset

The complete training dataset, denoted as $\mathcal{G}^{train}_{\mathbf{d}'}$, is a union of both the positive and negative sampling training datasets:

$$\mathcal{G}^{train}_{\mathbf{d}'} = \mathbf{G}^{train}_{\mathbf{d}'} \cup \mathbf{G}^{train}_{\perp,\mathbf{d}'} \tag{28}$$

6.2.4 Construction of the Testing Dataset

The process for constructing the testing dataset, $\mathcal{G}^{test}_{\mathbf{d}'}$, mirrors that of the training dataset with the following modifications:

1 Instead of $\mathbf{F}^{train}$, $\mathbf{F}^{test}$ is utilized, as indicated by the replacement of all superscripts labeled as *train* with *test*.

2 During the construction of $\mathbf{G}^{test,new}_{\mathbf{d}'}$, oversampling is not employed. Specifically, $\mathbf{G}^{test,new}_{\mathbf{d}'}$ is determined by the following:

$$\begin{aligned}\mathbf{G}^{test,new}_{\mathbf{d}'} = \bigcup_{i=1}^{|\mathbf{F}^{new,test}_{\mathbf{d}'}|} & \{(s_i, \text{root}(a_i), InDom_i) \\ & \mid (s_i, \text{root}(a_i), \pi_i) \in \mathbf{F}^{new,test}_{\mathbf{d}'} \text{ and } InDom_i = \text{True}\}\end{aligned} \tag{29}$$

6.3 Dynamic Training, Testing and Updating

The constant evolution of programming tasks necessitates the continuous refinement of the U2N neural network models. With the composite datasets, the neural network agent undertakes the task of training, testing, and updating the models in the background, ensuring that they are always in sync with the most recent programming challenges.

6.3.1 Training

As shown in figure 9, the training process in U2N involves multi-model training and multitasking, and it includes a contrastive training process contingent on the compositions of the dataset. Consider a trained neural network model $M_{\mathbf{d}',\theta} : e(s) \rightarrow (InDom, \Pi)$, where $\mathbf{d}'$ represents the knowledge domain to which the model is associated, $e(s)$ signifies the encoded form of instruction sequences, and s denote a state in the grounding process. The function $e(s) \rightarrow (InDom, \Pi)$ describes the dual objectives of the model: it takes a state s in the grounding process as input, and outputs $InDom$, the confidence level that the state is within the grounding process pertinent to knowledge domain $\mathbf{d}'$, and Π, a policy array. Each element in Π serves as an individual policy, indicating the probability of executing actions on sub Intermediate Representation (IR) segments that are based at a specific node.

Let the loss function for the first task be

$$L_{InDom}(InDom_{\text{output}}, InDom_{\text{label}}) = \text{BCE}(InDom_{\text{output}}, InDom_{\text{label}}) \tag{30}$$

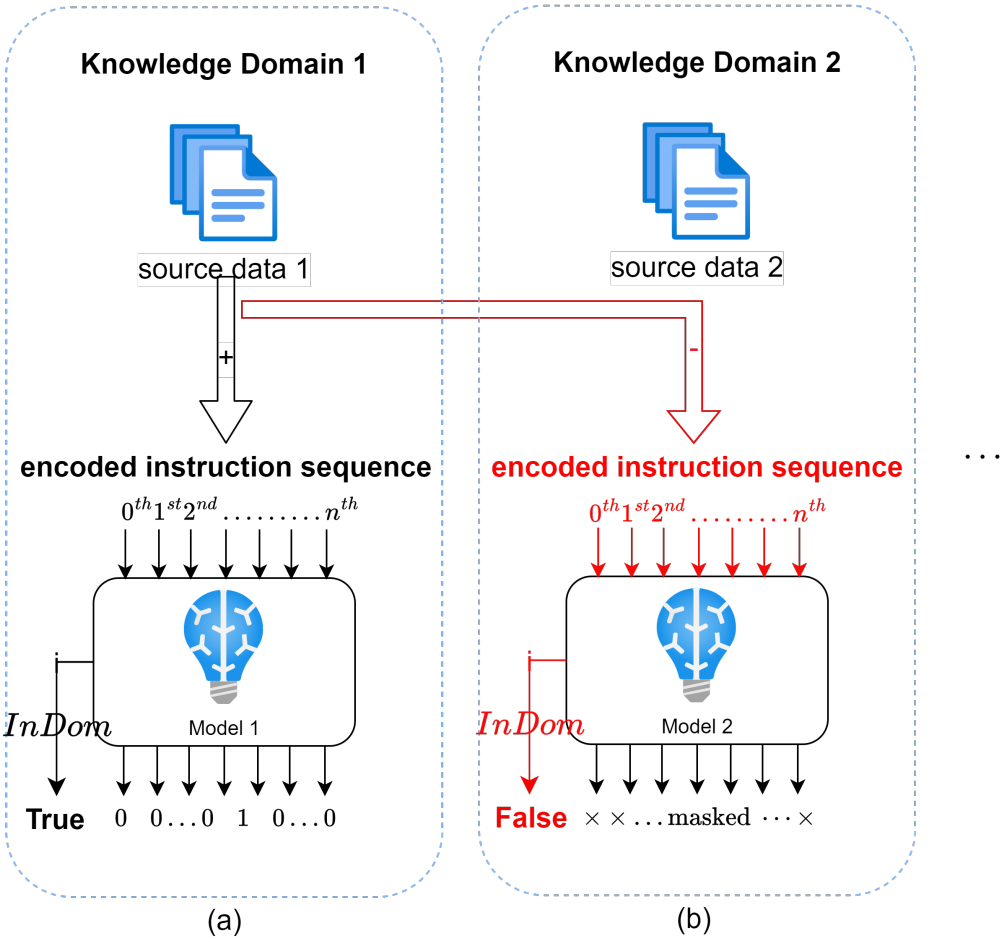

Figure 9: Training

where BCE denotes the Binary Cross-Entropy algorithm, and the loss function for the second task be

$$L_{\Pi} = \mathrm{CCE}(\Pi_{\mathrm{output}}, \Pi_{\mathrm{label}}) \tag{31}$$

where CCE denotes the Categorical Cross-Entropy algorithm, and $\Pi_{\mathrm{label}} = [\pi_i \mid i \in \mathbb{N}, i < |\Pi_{\mathrm{output}}|, \pi_{i \neq \mathrm{root}(a)} = 0 \text{ and } \pi_{i=\mathrm{root}(a)} = 1]$.

In the training dataset $\mathcal{G}^{\mathrm{train}}_{\mathbf{d}'}$, a positive sample like $(s, \mathrm{root}(a), InDom = \mathrm{True})$ can train both tasks, while a negative sample like $(s, InDom = \mathrm{False})$ can only be used to train the first task. Therefore, the combined loss function is:

$$L = \begin{cases} \epsilon * L_{InDom} + (1-\epsilon) * L_{\Pi}, & \text{if } InDom_{\mathrm{label}} = \mathrm{True} \\ \epsilon * L_{InDom}, & \text{if } InDom_{\mathrm{label}} = \mathrm{False} \end{cases} \tag{32}$$

where $\epsilon \in (0, 1)$.

The training process is defined as:

$$\theta^* = \arg\min_{\theta} \bar{L} \mid M_\theta, \mathcal{G}_{\mathbf{d}'}^{\text{train}} \tag{33}$$

Here, $\bar{L}$ is the average loss and θ^* denotes the optimized neural network parameters.

6.3.2 Testing and Updating

The performance of the updated model $M_{\theta^*,\mathbf{d}'}$ is measured by its accuracies on two separate tasks: The accuracy of predicting on $InDom$ is defined as

$$A_{InDom} = \frac{\text{number of outputs where } |InDom_{\text{label}} - InDom_{\text{output}}| < 0.5}{\text{total number of outputs}} \tag{34}$$

and the accuracy of predicting on Π is defined as

$$A_{\Pi} = \frac{\text{number of outputs where } \pi_{\text{root}(a)} = \max(\Pi)}{\text{total number of outputs}} \tag{35}$$

Then, save the updated model along with its latest performance data A_{InDom} and A_{Π}. Both the updated model and the accuracy reflect the improvements from the recent training iteration and will be used in the further guidance process.

It is important to note that if the dataset $\mathcal{G}_{\mathbf{d}'}$ contains no negative samples, then A_{InDom} will not be calculated nor updated. If $M_{\mathbf{d}'}$ has never been trained with a dataset containing negative samples, its $InDom$ and A_{InDom} are considered invalid. When these two metrics are utilized, both are assigned the Jaccard similarity coefficient:

$$\frac{|\mathbf{d}' \cap \mathbf{d}|}{|\mathbf{d}' \cup \mathbf{d}|} \tag{36}$$

6.4 Expand New Models

When the neural network agent receives $\mathbf{F}_{\mathbf{d}'}$ from the compiler and a corresponding model $M_{\mathbf{d}'}$ is not found, it indicates that a new knowledge domain has emerged. This could be due to the introduction of new files and classes, or new methods of combining multi-domain knowledge to solve problems. In such cases, the neural network agent will attempt to create a new model $M_{\mathbf{d}'}$ to learn this new knowledge, following the procedure shown in Figure 10, depending on user-defined configurations in the settings file.

6.4.1 Preliminary Model Structures

Users can define their own strategies for choosing the structure of the neural network model through the settings file. By default, the architecture employed is a bi-directional LSTM (Long Short-Term Memory) network [63, 64].

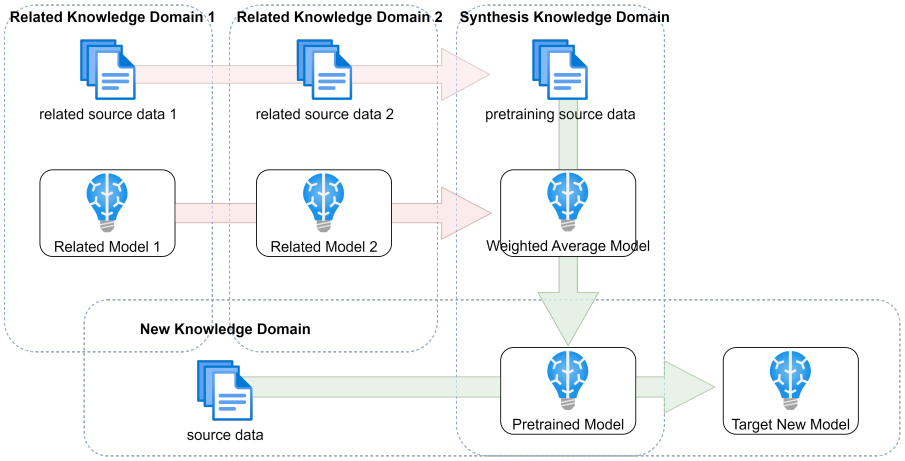

Figure 10: Expand New Model

6.4.2 Parameter Initialization

To initialize parameters, a set of existing knowledge domains is identified. This set, denoted as $\mathcal{D}'_{exist}$, comprises pre-created neural models that align with the preliminary model structures:

$$\mathcal{D}'_{exist} = \{\mathbf{d}'_{e,1}, \mathbf{d}'_{e,2}, \ldots, \mathbf{d}'_{e,n}\}. \tag{37}$$

The elements of this set should collectively cover the new domain's scope as comprehensively as possible and keep this set as small as possible:

$$\text{Priority 1:} \quad \min|\mathbf{d}' \oplus (\bigcup_i \mathbf{d}'_{e,i} \mid \mathbf{d}'_{e,i} \in \mathcal{D}'_{exist})| \tag{38}$$

$$\text{Priority 2:} \quad \min|\mathcal{D}'_{exist}| \tag{39}$$

where $\oplus$ denotes the symmetric difference operator.

If $\mathcal{D}'_{exist}$ is empty, the Xavier method [65] will be used for weight initialization. Otherwise, a shared weights strategy is applied for parameter initialization.

Then, the set $\mathcal{D}'_{exist}$ is sorted from high to low by the Jaccard similarity coefficient cj of its element and the target knowledge domain $\mathbf{d}'$:

$$cj = \frac{|\mathbf{d}'_e \cap \mathbf{d}'|}{|\mathbf{d}'_e \cup \mathbf{d}'|} \tag{40}$$

The initial parameters θ_{init} for $M_{\mathbf{d}'}$ are denoted as:

$$\theta_{init} = \frac{\Sigma_i cj_{\mathbf{d}'_{e,i}} * \theta_{\mathbf{d}'_{e,i}}}{\Sigma_i cj_{\mathbf{d}'_{e,i}}} \mid (i : \cos\langle \theta_{\mathbf{d}'_{e,i}}, \theta_{\mathbf{d}'_{e,1}} \rangle \geq \zeta) \tag{41}$$

where $\zeta \in [-1, 1]$ serves as the cosine similarity threshold to avoid parameters that differ too much from $\theta_{\mathbf{d}'_{e,1}}$ are included in the weighted average process.

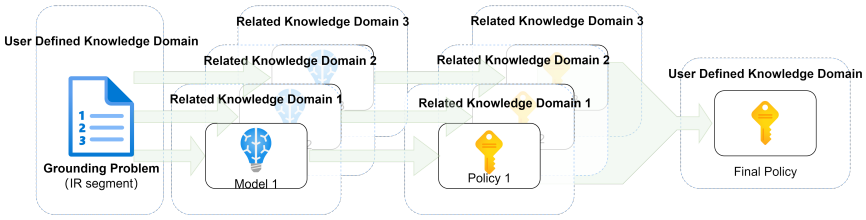

Figure 11: Policy-Making Process

6.4.3 Pretraining

After parameter initialization, the model $M_{\mathbf{d}'}$ undergoes a pretraining phase. During this phase, a pretraining dataset $\mathcal{G}_{\mathbf{d}'}^{pretrain}$, which is constructed by sampling from the feature sets $\mathcal{F}_{\mathbf{d}'_e}$ where $\mathbf{d}'_e \subseteq \mathbf{d}'$, is utilized. This pretraining allows the network to learn the features of problem-solving strategies inherent to the contributing knowledge domains, thus mitigating the impact of potential training data scarcity. Consequently, the pretraining results in $M_{\mathbf{d}'}$ becoming an amalgamation of the models $\{M_{\mathbf{d}'_e}\}$. However, the problems $M_{\mathbf{d}'}$ is now prepared to tackle may not align with those of the target knowledge domain $\mathbf{d}'$.

6.4.4 Transfer Learning

For $M_{\mathbf{d}'}$ to effectively address the grounding problems specific to the knowledge domain $\mathbf{d}'$, it undergoes further training, testing, and updating with the dataset $\mathbf{F}_{\mathbf{d}'}$, following the procedures outlined in Section 6.3.

6.4.5 Pretraining

After parameter initialization, a pre-training dataset $\mathcal{G}_{\mathbf{d}'}^{pretrain}$ sampling from all $\mathcal{F}_{\mathbf{d}'_e|\mathbf{d}'_e\subseteq\mathbf{d}'}$ is used to enable the network learn the features of problem-solving strategies of the knowledge domains deriving it, which weakens the effect of lack of training data. After this phase, $M_{\mathbf{d}'}$ is essentially a synthesis of $\{M_{\mathbf{d}'_e}\}$, therefore the grounding problems it can tackle are not perfectly consistent with the target knowledge domain $\mathbf{d}'$.

6.4.6 Transfer Learning

To solve grounding problems of knowledge domain $\mathbf{d}'$, the $M_{\mathbf{d}'}$ is then performed to further training, testing and updating with $\mathbf{F}_{\mathbf{d}'}$ just as section 6.3.

6.5 Policy-Making Process

The neural network agent predicts by providing policy array Π, in domain confidence ci, and accuracy ca to the compiler. As shown in figure 11, the policy made by the neural network agent is the result of collaboration from models of different knowledge domains. The critical phase unfolds as follows:

6.5.1 Selecting Collaborating Models

When the neural network receives the grounding state s and knowledge domain $\mathbf{d}$ as defined by domain specification prompts (DSPs), it identifies a subset of knowledge domains with pre-established neural models, denoted as $\mathcal{D}'co = \mathbf{d}co, 0', \mathbf{d}co, 1', \ldots, \mathbf{d}co, n'$, for the policy-making process. The union of these models should cover the full scope of d effectively, ensuring that each model is a subset of d and that the collective set of models is kept as small as possible:

$$\text{Priority 1:} \quad \min \mid \mathbf{d} \setminus (\bigcup_i \mathbf{d}'_{co,i} \mid \mathbf{d}'_{co,i} \in \mathcal{D}'_{co}) \mid \tag{42}$$

$$\text{Priority 2:} \quad \max \frac{\mid \{\mathbf{d}_{co,i} \mid \mathbf{d}'_{co,i} \in \mathcal{D}'_{co} \text{ and } \mathbf{d}'_{co,i} \subseteq \mathbf{d}\} \mid}{\mid \mathcal{D}'_{co} \mid} \tag{43}$$

$$\text{Priority 3:} \quad \min \mid \mathcal{D}'_{co} \mid \tag{44}$$

If $\mathbf{d}'_{co}$ is empty, the neural network agent will directly return an empty policy array with both ac and ci set to zero. Otherwise, $\mathcal{M}_{\mathbf{d}'_{co}}$, which contains models corresponding to the elements of $\mathbf{D}'_{co}$, will be loaded for prediction.

6.5.2 Predicting and Eliminating the Outliers

In this phase, each element in $\mathcal{M}_{\mathbf{d}'_{co}}$ makes prediction and both result and related coefficients are stored in

$$\mathbf{\Pi} = \{(\mathbf{d}'_{co,i}, \Pi_i, A_{\Pi,i}, InDom_i, A_{InDom,i}) \mid i \in \mathbb{N}, i <\mid \mathcal{M}_{\mathbf{d}'_{co}} \mid\} \tag{44}$$

Then eliminate the outliers by following the steps:

1 Sort $\mathbf{\Pi}$ based on $InDom * A_{InDom}$ from high to low (in descending order), conserve the top $\eta \in (0, 1]$ fraction elements. This step is used to eliminate the result generated by models not suitable to make policy for such kind of grounding state indicated by $InDom$.

2 Calculate weighted average policy array:

$$\overline{\Pi} = \frac{\sum_{i=1}^{|\mathbf{\Pi}|}(A_{\Pi,i} * InDom_i * A_{InDom,i}) * \Pi_i}{\sum_{i=1}^{|\mathbf{\Pi}|}(A_{\Pi,i} * InDom_i * A_{InDom,i})} \tag{44}$$

3 then, for each element in $\mathbf{\Pi}$, if $\mathbf{d}'_{co,i} \not\subseteq \mathbf{d}$, calculate the Symmetric Kullback-Leibler divergence between Π_i and $\overline{\Pi}$:

$$\text{Symmetric KL}(\Pi_i, \overline{\Pi}) = \frac{1}{2}\left(\mathrm{D_{KL}}(\Pi_i \| \overline{\Pi}) + \mathrm{D_{KL}}(\overline{\Pi} \| \Pi_i)\right) \tag{44}$$

where $\mathrm{D_{KL}}(\mathrm{p}, \mathrm{q})$ means the Kullback-Leibler divergence [66] between distributions p and q, and eliminate the elements satisfying Symmetric $\mathrm{KL}(\Pi_i, \overline{\Pi}) > SKL_{max}$,

where SKL_{max} is the threshold of the tolerant Symmetric Kullback-Leibler divergence. Those elements beyond this constraint are highly likely to misguide the compiler to apply or bind an uninvokable function belonging to knowledge domain $\mathbf{d}'_{co,i} \setminus \mathbf{d}$. Finally, denote the ultimate $\mathbf{\Pi}$ with no outliers as $\mathbf{\Pi}^*$

6.5.3 Predicting and Eliminating the Outliers

In this phase, each element within $\mathcal{M}_{\mathbf{d}'_{co}}$ makes prediction. The predicted policy arrays and related coefficients are stored in

$$\mathbf{\Pi} = \{(\mathbf{d}'_{co,i}, \Pi_i, A_{\Pi,i}, InDom_i, A_{InDom,i}) \mid i \in \mathbb{N}, i < |\mathcal{M}_{\mathbf{d}'_{co}}|\} \tag{44}$$

Subsequently, outliers are eliminated by following these steps:

1. Sort $\mathbf{\Pi}$ based on the product $InDom_i * A_{InDom,i}$ in descending order and retain the top $\eta \in (0, 1]$ fraction of elements. This step filters out results from models that are not adequately suited for making policy for state s outside their knowledge domain, as indicated by $InDom_i$.

2. Compute the weighted average policy array

$$\overline{\Pi} = \frac{\sum_{i=1}^{|\mathbf{\Pi}|}(A_{\Pi,i} * InDom_i * A_{InDom,i}) * \Pi_i}{\sum_{i=1}^{|\mathbf{\Pi}|}(A_{\Pi,i} * InDom_i * A_{InDom,i})} \tag{44}$$

3. For each element in $\mathbf{\Pi}$, if $\mathbf{d}'_{co,i} \not\subseteq \mathbf{d}$, calculate the Symmetric Kullback-Leibler divergence between Π_i and $\overline{\Pi}$:

$$\text{Symmetric KL}(\Pi_i, \overline{\Pi}) = \frac{1}{2}\left(\mathrm{D_{KL}}(\Pi_i \| \overline{\Pi}) + \mathrm{D_{KL}}(\overline{\Pi} \| \Pi_i)\right) \tag{44}$$

   where $\mathrm{D_{KL}}(\mathrm{p} \| \mathrm{q})$ denotes the Kullback-Leibler divergence between distributions p and q[66]. Eliminate elements where Symmetric $\mathrm{KL}(\Pi_i, \overline{\Pi}) > SKL_{max}$, with SKL_{max} representing the maximum tolerable Symmetric Kullback-Leibler divergence. Elements exceeding this threshold are likely to misdirect the compiler into applying or binding an inapplicable function belonging to the knowledge domain $\mathbf{d}'_{co,i} \setminus \mathbf{d}$.

Denote the ultimate $\mathbf{\Pi}$ with no outliers as $\mathbf{\Pi}^*$.

6.5.4 Synthesizing the Policy

Once outliers have been removed, synthesize the final reply policy and coefficients based on $\mathbf{\Pi}^*$:

$$\begin{cases} \Pi^* = \dfrac{\sum_{i=1}^{|\mathbf{\Pi}^*|}(A_{\Pi,i} * InDom_i * A_{InDom,i}) * \Pi_i}{\sum_{i=1}^{|\mathbf{\Pi}^*|}(A_{\Pi,i} * InDom_i * A_{InDom,i})} \\ ci^* = \dfrac{\sum_{i=1}^{|\mathbf{\Pi}^*|}(InDom_i * A_{InDom,i})}{\sum_{i=1}^{|\mathbf{\Pi}^*|}(A_{InDom,i})} \\ ac^* = \dfrac{\sum_{i=1}^{|\mathbf{\Pi}^*|}(A_{\Pi,i} * InDom_i * A_{InDom,i})}{\sum_{i=1}^{|\mathbf{\Pi}^*|}(InDom_i * A_{InDom,i})} \end{cases} \tag{44}$$

where $(\mathbf{d}'_{co,i}, \Pi_i, A_{\Pi,i}, InDom_i, A_{InDom,i}) \in \mathbf{\Pi}^*$.

Send the (Π^*, ac^*, ci^*) to the compiler to guide the grounding process together with process control prompts (PCPs).

7 Advanced Research Directions

In the era of rapid development of artificial intelligence, the COOL and relative concepts proposed in this article may have outstanding prospects in the following aspects:

7.1 Infrastructure for Edge Computing and Federated Learning

The COOL, and the U2N mechanism, are intrinsically suited for edge computing and federated learning[67]. As edge devices proliferate, localized data processing and decision-making become vital. By augmenting COOL's network support for edge environments, it can serve as a platform that facilitates seamless user interaction with edge computing and federated learning ecosystems. This integration is promising as it addresses pivotal concerns such as data privacy and the hinder of model training and iterative updates. Moreover, it enhance the inter-device collaboration, thereby enhancing the overall intelligence of distributed networks.

7.2 Mitigating the Dependency on Large Language Models' Scale through Neural-Symbolic Integration

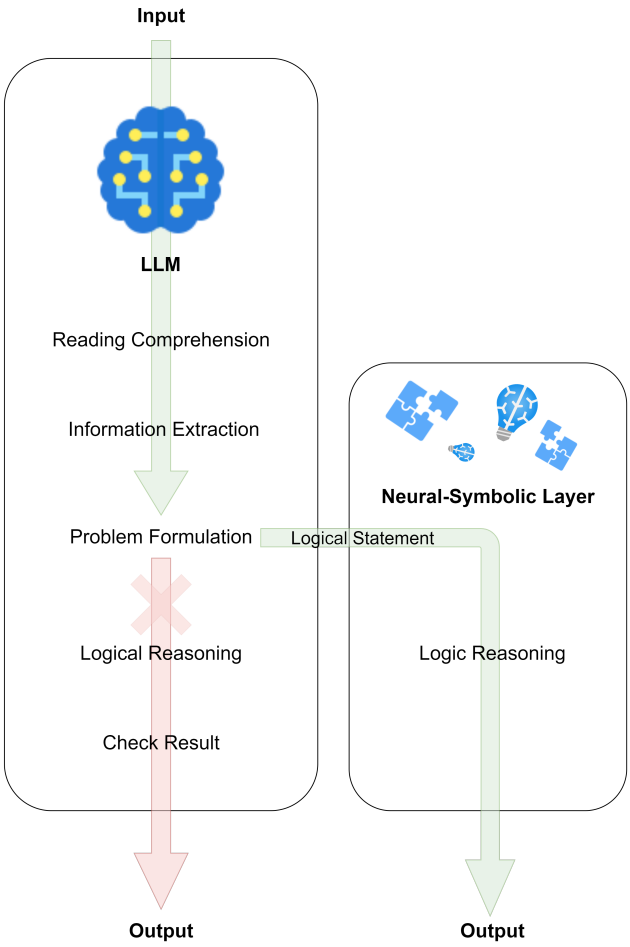

Figure 12: LLM with a Neural-Symbolic Layer

Built upon the Transformer architecture [68], Large Language Models (LLMs) have shown remarkable performance in natural language processing [69] and complex reasoning [70]. Despite their impressive capabilities, these models exhibit a concerning trend: their reasoning capacity is heavily reliant on the scale of the model, this relationship is characterized by rapidly diminishing returns. Consequently, each incremental improvement in reasoning ability results in considerably higher resource consumption compared to previous advancements. This pattern of growth is unsustainable, presenting both a developmental bottleneck and an environmental challenge for the ongoing deployment of LLMs.

As depicted in the Figure 12, a novel approach to this challenge involves the incorporation of neural-logic layers into the LLM framework to replace the logical reasoning process. In this configuration, the LLM specializes in tasks such as reading comprehension, information extraction, and problem formulation. Subsequently, the reasoning process is handled by the neural-symbolic layers, which are adept at performing logical reasoning tasks and have strong scalability. This layered structure promises to alleviate the constraints imposed by the physical scale of LLM networks, offering a pathway to enhance inferential capabilities without the associated resource intensity of traditional scaling methods.

8 Key Implementation Details

8.1 Hierarchical Address

In the IR segment, a hierarchical structure is adopted to denote the addresses in the IR.

$$A = \{a_1, a_2, \ldots, a_n\}, \text{ where } a_i \in \mathcal{N} \text{ and } 1 \leq i \leq n \tag{44}$$

This allows any number of addresses between the two addresses, ensuring that when the IR is partially replaced in the grounding process, the newly inserted IR segment always has enough addresses while not modifying other addresses.

8.2 Model Pool

For efficient storage and reuse of neural network models, a model pool with a Least Recently Used (LRU) strategy [71] is employed. As the model pool reaches its capacity, this strategy helps to identify and discard the least accessed models, thus prioritizing the retention of the most relevant and frequently used models. Moreover, a protection mechanism is in place for models that have been recently added to the pool, preventing their premature elimination and affording them a grace period to demonstrate their utility and relevance. These provisions are designed to optimize the use of computational resources.

9 Design of Experiment

9.1 Basic Experiments

9.1.1 Effect of Process Control Prompt

Purpose: To evaluate the impact of Process Control Prompts (PCPs) on the grounding process, and to observe how PCPs influence the quality of the generated Intermediate Representation (IR) as reflected by the execution speed of the resulting compiled program.

Control Variables: Grounding algorithm (BDDB), neural network agent disabled.

Independent Variables: PCPs enabled, PCPs disabled (with a setting that causes BDDB to operate like Breadth-First Search (BFS)), PCPs disabled (with a setting that causes BDDB to operate like Depth-First Search (DFS))

Dependent Variables: Success rate of compilation (p_{suc}), the average number of grounding states generated, average time taken for grounding and average execution time of the compiled program.

Materials: COOL source code for solving math problems from https://math-drills.com and similar sites.

Procedure: Compile the COOL source code with different configurations and execute the resulting program.

9.1.2 Effect of BDDB algorithm

Purpose: To evaluate the effect of BDDB algorithm on the grounding process and enhancing the quality of the generated IR.

Independent Variables: BDDB algorithm (with $\gamma = 0$ and $\lambda = 0$, which simplifies the approach to an approximation of greedy-policy Markov Decision Processes (MDPs) with the capability of backtracking), BDDB algorithm (with $\gamma = 0$ and a fine-tuned λ)

Dependent Variables: Success rate of compilation (p_{suc}), the average number of grounding states generated, average time taken for grounding and average execution time of the compiled program.

Materials: COOL source code for solving math problems.

Procedure: Compile the COOL source code with different configurations and execute the resulting program.

9.1.3 Effect of U2N on Single Knowledge Domain Problem Solving

Purpose: To assess the U2N algorithm's impact on the grounding process and the quality of IR for single-domain problem-solving.

Control Variables: Neural network agent (prediction enabled and initially without prior training), BDDB (with $\gamma = 0$ and a fine-tuned λ).

Independent Variables: The quantity of different COOL source codes of compiled with the compiler with neural network agent learning mode enabled.

Dependent Variables: The accuracy of the neural network's predictions, the success rate of compilation (p_{suc}), the average number of grounding states generated, and the average grounding and average execution times post-compilation.

Materials: COOL source code samples that address math problems within a specific knowledge domain.

Procedure: Compile the COOL source code with the neural network agent's learning mode enabled and subsequently, for evaluation purposes, with the learning mode disabled.

Analysis: Draw graphs of the dependent variables as functions of the independent variable to evaluate the learning ability of the neural network agent. and draw graphs of (the percentage of compilation completed successfully, average number of grounding states created, and average grounding time as functions of the accuracy of neural network models, and average program execution time after compilation) to evaluate the effect of neural network agents taking a bigger and bigger proportion in policy making. Generate graphical representations of the dependent variables in relation to the independent variable to analyze the learning efficacy of the neural network agent. Also, graph the relationship between neural network model's accuracy (policy-making influence) in terms of successful compilation rates, grounding states, and execution timings.

9.1.4 Effect of U2N on Multi Knowledge Domain Problem Solving

Purpose: To evaluate the capability of the U2N mechanism to handle multiple knowledge domains during the grounding process and to assess the quality of the generated IR.

Control Variables: BDDB algorithm (with $\gamma = 0$ and a fine-tuned λ).

Independent Variables: Neural network agent (Predicting enabled, learning mode disabled, well trained for handling problems of knowledge domain $\mathbf{d}'_1$ and $\mathbf{d}'_2$), Neural network agent disabled, different knowledge domains in the COOL source code being compiled.

Dependent Variables: The accuracy of neural network predictions, the success rate of compilation, the average number of grounding states generated, and the average grounding time.

Materials: COOL source code representing mathematical problems from the combined knowledge domains $\mathbf{d}_1$ and $\mathbf{d}_2$, where $\mathbf{d}_1$ includes domains $\mathbf{d}'_1$ and $\mathbf{d}'_2$, and $\mathbf{d}_2$ includes $\mathbf{d}'_1$, $\mathbf{d}'_2$, and $\mathbf{d}'_3$.

Procedure: Sequentially compile the COOL source code with the neural network agent both enabled and disabled.

9.2 Advanced Experiments

9.2.1 Effect of combination of LLM with neural-symbolic layer

Background and Purpose: As the increase in model size for large language models (LLMs) such as GPT series[72] leads to diminishing returns, this experiment seeks to determine whether combining a smaller LLM with a neural-symbolic layer can match or surpass the logical reasoning abilities of a larger LLM alone, providing an alternative approach for enhancing AI reasoning.

Independent Variables: Comparison of GPT-2[73] and GPT-3[74], each augmented with a neural-symbolic layer, against a baseline version of a larger GPT model (e.g., GPT-4)

Dependent Variables: The reasoning accuracy, efficiency, and complexity of problems that can be solved by each framework.

Materials: Math word problems varying in complexity.

Procedure: Conduct problem-solving sessions using each framework to solve the dataset of math word problems.

9.2.2 Comparison of reasoning ability of different LLM-based frameworks

Background: Research like VisProg [75] and Program-of-Thoughts[76] has shown improvements in reasoning ability when LLMs are used to generate Python code. However, these benefits often rely on the LLM's ability to formulate a correct Chain-of-Thought [77] (CoT). Complex problems that exceed an LLM's reasoning capacity may be unsolvable within this paradigm. Such limitations may be overcome by integrating a neural-symbolic layer, which can take over the complex logical reasoning process after the problem formulation stage, as illustrated in Figure 12.

Purpose: To evaluate how adding a neural-symbolic layer to an LLM affects its reasoning capabilities, especially for complex problems where chain-of-thought approaches may fail.

Control Variables: Consistent GPT version across all frameworks.

Independent Variables: Three frameworks are compared: GPT augmented with chain-of-thought prompting (CoT), GPT enhanced with chain-of-prompts (CoP), and GPT integrated with a neural-symbolic layer.

Dependent Variables: The accuracy and efficiency of problem-solving.

Materials: Math word problems varying in complexity.

Procedure: Carry out problem-solving tasks using each framework to solve the dataset of math word problems.

9.3 Long-Term Experiment

9.3.1 Application of COOL

Purpose: To evaluate the COOL programming language's readiness for production use by assessing its features, performance, and user satisfaction over a series of tests.

Independent Variables: Language features, algorithm optimization, execution environment, documentation.

Dependent Variables: Application areas, performance, user feedback.

References

[1] Lei Cai, Jingyang Gao, and Di Zhao. A review of the application of deep learning in medical image classification and segmentation. *Annals of translational medicine*, 8(11), 2020.

[2] Alex Graves and Navdeep Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *International conference on machine learning*, pages 1764–1772. PMLR, 2014.

[3] Yonatan Belinkov and James Glass. Analysis methods in neural language processing: A survey. *Transactions of the Association for Computational Linguistics*, 7:49–72, 2019.

[4] Bryan Lim and Stefan Zohren. Time-series forecasting with deep learning: a survey. *Philosophical Transactions of the Royal Society A*, 379(2194):20200209, 2021.

[5] Mehar Vijh, Deeksha Chandola, Vinay Anand Tikkiwal, and Arun Kumar. Stock closing price prediction using machine learning techniques. *Procedia computer science*, 167:599–606, 2020.

[6] Iria Santos, Luz Castro, Nereida Rodriguez-Fernandez, Alvaro Torrente-Patino, and Adrian Carballal. Artificial neural networks and deep learning in the visual arts: A review. *Neural Computing and Applications*, 33:121–157, 2021.

[7] Shulei Ji, Jing Luo, and Xinyu Yang. A comprehensive survey on deep music generation: Multi-level representations, algorithms, evaluations, and future directions. *arXiv preprint arXiv:2011.06801*, 2020.

[8] Artur d'Avila Garcez, Sebastian Bader, Howard Bowman, Luis C Lamb, Leo de Penning, BV Illuminoo, Hoifung Poon, and COPPE Gerson Zaverucha. Neural-symbolic learning and reasoning: A survey and interpretation. *Neuro-Symbolic Artificial Intelligence: The State of the Art*, 342(1):327, 2022.

[9] Krishan Kumar and Sonal Dahiya. Programming languages: A survey. *International Journal on Recent and Innovation Trends in Computing and Communication*, 5(5):307–313, 2017.

[10] Kevin Ellis, Lucas Morales, Mathias Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. Learning libraries of subroutines for neurally-guided bayesian program induction. 31.

[11] Lisa Zhang, Gregory Rosenblatt, Ethan Fetaya, Renjie Liao, William Byrd, Matthew Might, Raquel Urtasun, and Richard Zemel. Neural guided constraint logic programming for program synthesis. 31.

[12] Honghua Dong, Jiayuan Mao, Tian Lin, Chong Wang, Lihong Li, and Denny Zhou. NEURAL LOGIC MACHINES.

[13] Zhiting Hu, Xuezhe Ma, Zhengzhong Liu, Eduard Hovy, and Eric Xing. Harnessing Deep Neural Networks with Logic Rules. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2410–2420, Berlin, Germany, August 2016. Association for Computational Linguistics.

[14] Wenya Wang and Sinno Jialin Pan. Integrating deep learning with logic fusion for information extraction. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 9225–9232, 2020. Issue: 05.

[15] Thomas Winters, Giuseppe Marra, Robin Manhaeve, and Luc De Raedt. Deepstochlog: Neural stochastic logic programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 10090–10100, 2022. Issue: 9.

[16] Robin Manhaeve, Sebastijan Dumančić, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Neural probabilistic logic programming in DeepProbLog. *Artificial Intelligence*, 298:103504, 2021. Publisher: Elsevier.

[17] Qing Li, Siyuan Huang, Yining Hong, Yixin Chen, Ying Nian Wu, and Song-Chun Zhu. Closed Loop Neural-Symbolic Learning via Integrating Neural Perception, Grammar Parsing, and Symbolic Reasoning. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 5884–5894. PMLR, July 2020.

[18] Marc Fischer, Mislav Balunovic, Dana Drachsler-Cohen, Timon Gehr, Ce Zhang, and Martin Vechev. DL2: Training and Querying Neural Networks with Logic. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 1931–1941. PMLR, June 2019.

[19] Prithviraj Sen, Breno WSR de Carvalho, Ryan Riegel, and Alexander Gray. Neuro-symbolic inductive logic programming with logical neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 8212–8219, 2022. Issue: 8.

[20] Ute Schmid. Inductive Programming as Approach to Comprehensible Machine Learning. In *DKB/KIK@ KI*, pages 4–12, 2018.

[21] Andrew Cropper, Rolf Morel, and Stephen Muggleton. Learning higher-order logic programs. *Machine Learning*, 109:1289–1322, 2020. Publisher: Springer.

[22] Hikaru Shindo, Masaaki Nishino, and Akihiro Yamamoto. Differentiable inductive logic programming for structured examples. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 5034–5041, 2021. Issue: 6.

[23] Kun Gao, Hanpin Wang, Yongzhi Cao, and Katsumi Inoue. Learning from interpretation transition using differentiable logic programming semantics. *Machine Learning*, pages 1–23, 2022. Publisher: Springer.

[24] Leon Weber, Pasquale Minervini, Ulf Leser, and Tim Rocktäschel. NLProlog: Reasoning with Weak Unification for Natural Language Question Answering. 2018.

[25] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. Knowledge distillation: A survey. *International Journal of Computer Vision*, 129:1789–1819, 2021.

[26] Xingjiao Wu, Luwei Xiao, Yixuan Sun, Junhang Zhang, Tianlong Ma, and Liang He. A survey of human-in-the-loop for machine learning. *Future Generation Computer Systems*, 135:364–381, 2022.

[27] Ivan Bratko and Stephen Muggleton. Applications of Inductive Logic Programming. *Commun. ACM*, 38(11):65–70, November 1995. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[28] Thomas Winters, Giuseppe Marra, Robin Manhaeve, and Luc De Raedt. Deepstochlog: Neural stochastic logic programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 10090–10100, 2022.

[29] Ashwin Kalyan, Dhruv Batra, Abhishek Mohta, Prateek Jain, Oleksandr Polozov, and Sumit Gulwani. NEURAL-GUIDED DEDUCTIVE SEARCH FOR REAL-TIME PROGRAM SYNTHESIS FROM EXAMPLES. *ICLR*.

[30] Illia Polosukhin and Alexander Skidanov. Neural Program Search: Solving Programming Tasks from Description and Examples. 2018.

[31] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

[32] Henry Lieberman. Programming by example (introduction). *Communications of the ACM*, 43(3):72–74, 2000.

[33] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *International Conference on Machine Learning*, pages 187–195. PMLR, 2013.

[34] Luc De Raedt, Angelika Kimmig, Hannu Toivonen, and M Veloso. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI 2007, Proceedings of the 20th international joint conference on artificial intelligence*, pages 2462–2467. IJCAI-INT JOINT CONF ARTIF INTELL, 2007.

[35] William E Byrd. *Relational programming in miniKanren: techniques, applications, and implementations*. PhD thesis, Indiana University, 2009.

[36] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126, 2015.

[37] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

[38] Sumit Gulwani. Programming by examples. *Dependable Software Systems Engineering*, 45(137):3–15, 2016.

[39] JAN WIELEMAKER, TOM SCHRIJVERS, MARKUS TRISKA, and TORBJÖRN LAGER. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012. Publisher: Cambridge University Press.

[40] Tim A Majchrzak and Herbert Kuchen. Logic java: combining object-oriented and logic programming. In *International Workshop on Functional and Constraint Logic Programming*, pages 122–137. Springer, 2011.

[41] Miguel Calejo. InterProlog: Towards a declarative embedding of logic programming in Java. In *Logics in Artificial Intelligence: 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004. Proceedings 9*, pages 714–717. Springer, 2004.

[42] Farhad Pourpanah, Moloud Abdar, Yuxuan Luo, Xinlei Zhou, Ran Wang, Chee Peng Lim, Xi-Zhao Wang, and QM Jonathan Wu. A review of generalized zero-shot learning methods. *IEEE transactions on pattern analysis and machine intelligence*, 2022.

[43] Ashish Jaiswal, Ashwin Ramesh Babu, Mohammad Zaki Zadeh, Debapriya Banerjee, and Fillia Makedon. A survey on contrastive self-supervised learning. *Technologies*, 9(1):2, 2020.

[44] Gopalan Nadathur and Dale Miller. Higher-order logic programming. *Handbook of logic in artificial intelligence and logic programming*, 5:499–590, 1998.

[45] Python Software Foundation. Python documentation, 2023. Python Language Reference 6.16.

[46] Sharon Sickel. Invertibility of logic programs. In *4th Workshop on Automated Deduction*, pages 103–109, 1979.

[47] Sergei Abramov and Robert Glück. The universal resolving algorithm and its correctness: inverse computation in a functional language. *Science of Computer Programming*, 43(2-3):193–229, 2002.

[48] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Principles of a reversible programming language. In *Proceedings of the 5th Conference on Computing Frontiers*, pages 43–54, 2008.

[49] Eduardo Costa. Visual prolog for tyros. Pdc, 2008.

[50] Joxan Jaffar, Spiro Michaylov, Peter J Stuckey, and Roland HC Yap. The clp (r) language and system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(3):339–395, 1992.

[51] V Sekovanić and S Lovrenčić. Challenges in teaching logic programming. In *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 594–598. IEEE, 2022.

[52] Roland N Bol, Krzysztof R Apt, and Jan Willem Klop. An analysis of loop checking mechanisms for logic programs. *Theoretical computer science*, 86(1):35–79, 1991.

[53] JF Reiser. Compiling three-address code for c programs. *The Bell System Technical Journal*, 60(2):159–166, 1981.

[54] Goetz Graefe and William J McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of IEEE 9th international conference on data engineering*, pages 209–218. IEEE, 1993.

[55] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics (TOG)*, 31(4):1–12, 2012.

[56] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer set solving in practice*. Springer Nature, 2022.

[57] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[58] Satinder P Singh and Richard S Sutton. Reinforcement learning with replacing eligibility traces. *Machine learning*, 22:123–158, 1996. Publisher: Springer.

[59] Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643*, 2020.

[60] Ashish Jaiswal, Ashwin Ramesh Babu, Mohammad Zaki Zadeh, Debapriya Banerjee, and Fillia Makedon. A survey on contrastive self-supervised learning. *Technologies*, 9(1):2, 2020.

[61] Jie Lu, Anjin Liu, Fan Dong, Feng Gu, Joao Gama, and Guangquan Zhang. Learning under concept drift: A review. *IEEE transactions on knowledge and data engineering*, 31(12):2346–2363, 2018.

[62] Ajinkya More. Survey of resampling techniques for improving classification performance in unbalanced datasets. *arXiv preprint arXiv:1608.06048*, 2016.

[63] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681, 1997.

[64] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

[65] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[66] John R Hershey and Peder A Olsen. Approximating the kullback leibler divergence between gaussian mixture models. In *2007 IEEE International Conference on Acoustics, Speech and Signal Processing-ICASSP'07*, volume 4, pages IV–317. IEEE, 2007.

[67] Haftay Gebreslasie Abreha, Mohammad Hayajneh, and Mohamed Adel Serhani. Federated learning in edge computing: a systematic survey. *Sensors*, 22(2):450, 2022.

[68] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[69] Bonan Min, Hayley Ross, Elior Sulem, Amir Pouran Ben Veyseh, Thien Huu Nguyen, Oscar Sainz, Eneko Agirre, Ilana Heintz, and Dan Roth. Recent advances in natural language processing via large pre-trained language models: A survey. *ACM Computing Surveys*, 56(2):1–40, 2023.

[70] Jie Huang and Kevin Chen-Chuan Chang. Towards reasoning in large language models: A survey. *arXiv preprint arXiv:2212.10403*, 2022.

[71] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, 50(12):1352–1361, 2001.

[72] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[73] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[74] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[75] 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.

[76] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

[77] 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.

10 *About the Author

We are exploring new research areas and rely on thorough experiments to support our work. We've started detailed experiments to test and improve our proposed ideas.

These experiments need significant computing power and expertise. We see the value in our work and know collaboration can enhance its impact.

I'm looking for a Ph.D. program that aligns with my research interests.

Meanwhile, we welcome the community's input, advice, and partnership offers to advance our field together.