3-2024

Meta-Interpretive LEarning with Reuse

Rong WANG

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Cong TIAN

Zhenhua DUAN

Citation

WANG, Rong; SUN, Jun; TIAN, Cong; and DUAN, Zhenhua. Meta-Interpretive LEarning with Reuse. (2024). *Mathematics*. 12, (6), 1-20.
Available at: https://ink.library.smu.edu.sg/sis_research/8728

mathematics

MDPI

Article

Meta-Interpretive LEarning with Reuse

Rong Wang [1,2,*], Jun Sun [3], Cong Tian [1,2] and Zhenhua Duan [1,2]

1 School of Computer Science and Technology, Xidian University, Xi'an 710126, China; ctian@mail.xidian.edu.cn (C.T.); zhhduan@mail.xidian.edu.cn (Z.D.)
2 Institute of Computing Theory and Technology (ICTT), Xidian University, Xi'an 710126, China
3 School of Computing and Information Systems, Singapore Management University, Singapore 178902, Singapore; junsun@smu.edu.sg
* Correspondence: bilywr@163.com

Abstract: Inductive Logic Programming (ILP) is a research field at the intersection between machine learning and logic programming, focusing on developing a formal framework for inductively learning relational descriptions in the form of logic programs from examples and background knowledge. As an emerging method of ILP, Meta-Interpretive Learning (MIL) leverages the specialization of a set of higher-order metarules to learn logic programs. In MIL, the input includes a set of examples, background knowledge, and a set of metarules, while the output is a logic program. MIL executes a depth-first traversal search, where its program search space expands polynomially with the number of predicates in the provided background knowledge and exponentially with the number of clauses in the program, sometimes even leading to search collapse. To address this challenge, this study introduces a strategy that employs the concept of reuse, specifically through the integration of auxiliary predicates, to reduce the number of clauses in programs and improve the learning efficiency. This approach focuses on the proactive identification and reuse of common program patterns. To operationalize this strategy, we introduce MILER, a novel method integrating a predicate generator, program learner, and program evaluator. MILER leverages frequent subgraph mining techniques to detect common patterns from a limited dataset of training samples, subsequently embedding these patterns as auxiliary predicates into the background knowledge. In our experiments involving two Visual Question Answering (VQA) tasks and one program synthesis task, we assessed MILER's approach to utilizing reusable program patterns as auxiliary predicates. The results indicate that, by incorporating these patterns, MILER identifies reusable program patterns, reduces program clauses, and directly decreases the likelihood of timeouts compared to traditional MIL. This leads to improved learning success rates by optimizing computational efforts.

Keywords: meta-interpretive learning; program synthesis; inductive logic programming; frequent subgraph mining

MSC: 68T27

check for updates

Citation: Wang, R.; Sun, J.; Tian, C.; Duan, Z. Meta-Interpretive LEarning with Reuse. *Mathematics* **2024**, *12*, 916. https://doi.org/10.3390/math12060916

Academic Editor: Radu Tudor Ionescu

Received: 19 February 2024
Revised: 16 March 2024
Accepted: 19 March 2024
Published: 20 March 2024

1. Introduction

Symbolic artificial intelligence, which emerged in the 1950s and was probably the predominant approach until the late 1980s [1], aims to imitate human thinking by encoding knowledge in symbols and manipulating these symbols with logical rules to understand and solve complex problems. A symbolic AI system operates through logic-based steps on symbolic representations. These representations, often in the form of propositions, establish relationships between objects. Reasoning steps involve deriving new relations from existing ones, guided by a set of formal inference rules. Inductive logic programming (ILP) [2–4], is a subfield that synthesizes logic programs from observed data (usually a small number of examples) and empirical data (background knowledge). Meta-interpretive learning (MIL) is a newly proposed ILP technique capable of generating human-readable logic programs from a set of examples, a set of metarules (a restricted form of higher-order Horn clauses) and background knowledge. The novelty of MIL lies in its support for the predicate invention and learning of recursive programs. In practical applications, MIL has been successfully applied in areas such as synthetic biology [5], grammatical inference [6], text classification [7], and visual concept learning [8–10], demonstrating its powerful logical reasoning and pattern recognition capabilities.

MIL constrains the structure of the program to be synthesized using metarules as templates, with the MIL learner performing a depth-first traversal search. This process involves essentially generating programs by evaluating the different combinations of the primitive predicates, which in turn results in a vast potential program search space. This expansive search space is a direct consequence of MIL's inherent learning mechanism. Existing approaches to mitigate these problems heavily rely on significant human guidance in the form of metarules and specific domain knowledge related to the goal program. For instance, the work in [11] introduces layered bias reformulation in a multitask inductive programming setting. The work in [12] introduces higher-order operations involving an alternation between `abstraction` and `invention` to reduce the complexity of the learned programs. The work in [13] optimizes the search space by introducing types into predicates. These approaches primarily focus on enhancing the learning efficiency by introducing constraints or guidance during the learning phase. However, a more cost-effective strategy emerges from extensively mining existing examples before the learning process even begins.

Based on this, we propose a method called MILER (Meta-Interpretive Learning with Reuse), aimed at identifying frequently occurring programming patterns in existing examples and reusing them. This approach helps reduce the risk of crashes due to excessively large program search spaces, thereby increasing the efficiency and success rate of the learning process. However, extracting and utilizing these program patterns through symbolic machine learning techniques poses its own set of challenges, especially in some application areas where there may be a large amount of multimodal data. Inspired by how programmers solve coding problems in the real world, we designed a structured approach to problem-solving, actively seeking opportunities for reuse based on the structure of existing program patterns. MILER consists of three main components: a predicate generator based on frequent subgraph mining algorithms, a program learner as in MIL and a program evaluator. Specifically, MILER employs frequent subgraph mining technology to extract reusable program patterns from existing training examples and adds these patterns as new auxiliary predicates into the background knowledge of program learner. Once these auxiliary predicates are captured by the program learner, programs with fewer clauses can be generated, allowing the learning process to conclude at shallower depths during the depth-first search iteration. This approach enables MILER to traverse a smaller program search space, enhancing efficiency and effectiveness in program synthesis. To validate the effectiveness of MILER, we conducted experiments on two Visual Question Answering (VQA) tasks [14,15], as well as a program synthesis task [16]. The results show that, compared to the original MIL, MILER can effectively identify reusable program patterns, and these patterns as auxiliary predicates can significantly reduce the number of clauses in the learned programs, decrease the occurrence of timeouts, and thus enhance the success rate of learning. Moreover, it is important to emphasize that our method maintains robust learning efficiency, even with very few training examples (only dozens of samples). The main contributions of this study include the following:

- Utilizing frequent subgraph mining technology to extract reusable program patterns from a limited number of training examples;
- Enriching the existing information in the background knowledge by adding auxiliary predicates, thereby enhancing the extensibility of the background knowledge (BK);
- Reducing the occurrence of timeouts during the learning process and increasing the learning efficiency by enabling MILER to learn programs containing fewer clauses.

The rest of this paper is organized as follows. In Section 2, we discuss other relevant works in the field. In Section 3, we introduce the fundamental symbols, concepts, and principles that underpin our work. In Section 4, we begin with the motivation of this work, followed by an in-depth exposition of our algorithm, the improvements we have incorporated, and the overarching MILER framework. Section 5 presents an exploration of our experiments. Section 6 provides a comprehensive summary and outlines the future direction of our research.

2. Related Work

In this section, we study the existing research that mostly relate to ours.

2.1. Meta-Interpretive Learning

The origins of Inductive Logic Programming (ILP) can be credited to early significant contributions that laid the groundwork for its evolution. Pioneering efforts in the domain [17–20] provided a critical foundation, guiding the direction for future research in ILP. Initial ILP research concentrated on core theories like predicate invention and inverse resolution, leading to the creation of influential systems including FOIL [21], Golem [22], Progol [23], TILDE [24], and Aleph [25]. Meta-interpretive learning (MIL) is a newly proposed ILP method [6] for automatically synthesizing programs through optimized searching, which is further applied to learn higher-order dyadic datalog programs [26]. One core feature of MIL is that it supports user-specified metarules. Cropper et al. [27] show that the selection of metarules determines the structure of learnable programs and proves that the entailment reduction algorithm can be applied to automatically explore the irreducible sets of metarules. Later, Cropper et al. [28] proposed another reduction algorithm based on derivation to reduce the metarule sets of MIL, thus reducing the program space. Unlike these methods introducing constraints or guidance during the learning phase to enhance MIL's learning efficiency, our approach focuses on mining reusable program patterns from existing training examples before the MIL learning stage.

Building upon early ILP methods utilized for concept learning from images [29,30], the development of the logical vision (LV) model [31], based on MIL, marks a significant advancement. LV extracts polygon edges as low-level symbols and learns high-level structured geometrical notions. Then, the work in [8,9] proposed a noise-robust version of LV. Unlike these methods, our approach uses frequent subgraph mining to generate auxiliary predicates as an extension of the background knowledge.

2.2. Visual Question Answering

Visual Question Answering (VQA) [32–34] is an active research area which aims to build a computer system to automatically answer questions (in natural language) regarding information that can be inferred from an image. VQA typically requires an understanding of vision, language and certain common knowledge (e.g., simple geometry). Popular VQA datasets, such as CLEVR [14], provide a set of primitive functions (i.e., the primitive predicates in MIL) which can be combined to answer the questions. Existing methods for VQA [15,35,36] work by training a neural network as a generator to produce a program, and then execute the program by an execution engine to answer the question. The approaches in this way often require a fairly large set of sample question-programs pairs as the training set, which hinders these approaches from being widely applied. Our approach aims to address the VQA problem by significantly reducing the size of the required training set. That is, by extracting frequent reusable program patterns from a small set of training samples, we are able to leverage the optimized searching capability of MIL to automatically synthesize the program. That is, MILER can achieve a similar performance to that of the state-of-the-art VQA methods with a much smaller training set and often generates programs that are smaller in size.

2.3. Program Pattern Reuse

The reuse of program patterns is an important aspect in the areas of code generation and program synthesis. Numerous research papers have dealt with this topic and explored various methods and technologies to improve the efficiency and quality of software development. CARLCS-CNN [30], for example, uses a co-attention mechanism (learning interdependent representations for embedded code and directly querying their semantic correlation) to improve the efficiency of the search process and the reuse of existing code from large code bases. SkCoder [37] is a sketch-based code generation approach that mimics the behavior of developers when reusing code. It searches for a similar code snippet for a given natural language requirement, extracts relevant parts as a code sketch, and reworks the sketch into the desired code. In contrast to these methods, which focus on the semantic correlation between code parts, our approach places more emphasis on the structure of reusable program patterns, utilizing them as predicates. This significantly enhances the scalability of logical programming.

3. Preliminaries

This section briefly presents the concepts and terminologies used throughout this paper.

3.1. Logical Notations

A *variable* is indicated by an uppercase letter followed by a sequence of lowercase letters and digits, whereas a *function* symbol and a *predicate* symbol are indicated by a lowercase letter followed by a sequence of lowercase letters and digits. Let $\mathcal{V}$, $\mathcal{F}$, and Π denote a set of variables, function symbols, and predicate symbols, respectively. An *atom* is of the form $p(t_1, \cdots, t_n)$ with $p/n \in \Pi$ and $t_i \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ (the term set constructed by using symbols from $\mathcal{F}$ and variables from $\mathcal{V}$) for $i \in \{1, \cdots, n\}$. Here, predicate p is called an n-arity predicate. The set of all predicate symbols is referred to as the *predicate signature*. A *constant is a function or predicate symbol with an arity of zero*. The set of all constants is referred to as the *constant signature* and denoted by $\mathcal{C}$. Both an atom A (referred to as a positive literal) and its negation $\neg A$ are literals. A *clause* is represented by a finite set of literals. A *Horn clause* is a clause that contains at most one positive literal. A *definite clause* has the form $head \leftarrow body$, where the *head* is a positive literal and the *body* is a conjunction of literals. In Prolog, a definite clause is written as `Head :- Body`. A collection of clauses is referred to as a *clausal theory*, which conveys the conjunction of its constituent clauses. A *fact* is a definite clause with an empty body which can be viewed as a ground atom. A *rule* is a definite clause with a non-empty body, and it can be written as $head \leftarrow body_1, body_2, \cdots, body_n$ where $body_1, body_2, \cdots, body_n$ are literals. A *logic program* is a finite set of facts and rules. A *ground instance* of a logic program is obtained by replacing each variable in the program with constants. A *goal* is a special type of clause that has an empty head and is written as $\leftarrow body$. The process of proving a goal from a given set of clauses is known as *resolution*.

A *Well-Formed Formula (WFF)* is a formula that adheres to the syntactic rules of a particular logical system. Literals, clauses, and clausal theories are all well-formed formula (wffs) in which the variables are assumed to be universally quantified. A wff is said to be *higher-order* whenever it contains at least one higher-order variable or a predicate symbol as an argument of a term. Let E be a wff. E is said to be *datalog* if it contains no function symbols other than constants. The set of all ground atoms constructed from $\mathcal{P}$ and $\mathcal{C}$ is called the *datalog Herbrand Base*. $\theta = \{v_1/t_1, \cdots, v_n/t_n\}$ is a substitution in the case that each v_i is a variable and each t_i is a term. $E\theta$ is formed by replacing each variable v_i from θ found in E by t_i. μ is called a unifying substitution for atoms A and B in the case of $A\mu = B\mu$. We say clause C θ-subsumes clause D or $C \succeq_\theta D$ whenever there exists a substitution θ such that $C\theta \subseteq D$.

3.2. Metarule

Definition 1 (Metarules). *A metarule is a higher-order wff*

$$\exists\sigma\forall\tau P(s_1, \cdots, s_m) \leftarrow \cdots, Q_i(t_1, \cdots, t_n), \cdots$$

where σ and τ are disjoint sets of variables, $P, Q_i \in \sigma \cup \tau \cup P$ and $s_1, \cdots, s_m$, $t_1, \cdots, t_n \in \sigma \cup \tau \cup \mathcal{C}$. Metarules are denoted concisely without quantifiers as

$$P(s_1, \cdots, s_m) \leftarrow \cdots, Q_i(t_1, \cdots, t_n), \cdots$$

Metarules play a central role in the MIL process. The *chain* metarule $P(x, y) \leftarrow Q(x, z), R(z, y)$ in Table 1, for example, includes *existentially quantified second-order variables*, denoted by the capital letters P, Q, and R, and *universally quantified first-order variables*, represented by the lowercase letters x, y, and z. An order constraint associated with each metarule ensures the termination of the proof process.

Table 1. Metarule examples.

Name	Metarules	Order Constraint
Curry3	$P(x, y) \leftarrow Q(x, y, R)$	$P \succ Q, P \succ R$
Precon	$P(x, y) \leftarrow Q(x), R(x, y)$	$P \succ Q, P \succ R$
Postcon	$P(x, y) \leftarrow Q(x, y), R(y)$	$P \succ Q, P \succ R$
Chain	$P(x, y) \leftarrow Q(x, z), R(z, y)$	$P \succ Q, P \succ R$
Tailrec	$P(x, y) \leftarrow Q(x, z), P(z, y)$	$P \succ Q, x \succ z \succ y$
Ifthenelse	$P(x, y) \leftarrow ifthenelse(x, y, Cond, Then, Else)$	$P \succ Cond, P \succ Then, P \succ Else$

3.3. Meta-Interpretive Learning

Definition 2 (MIL Problem [12]). *An MIL problem is a tuple $Input = \langle B, E^+, E^-, M \rangle$ where the background knowledge B is a set of primitive predicates; E^+ (and E^-) is a set of positive (and negative) examples; and M is a set of metarules. A solution to the MIL problem is a program H such that $B, H \models e^+$ for $\forall e^+ \in E^+$ and $B, H \not\models e^-$ for $\forall e^- \in E^-$.*

Figure 1 shows the framework of MIL. The MIL learner is based on an adapted Prolog meta-interpreter, which attempts to prove a goal by repeatedly fetching first-order clauses or higher-order metarules whose heads match the goal [38]. For an MIL learner, after proving a set of examples, a program is formed by applying the *meta-substitutions* onto the corresponding metarules. Table 2 shows a Prolog program generated by MIL.

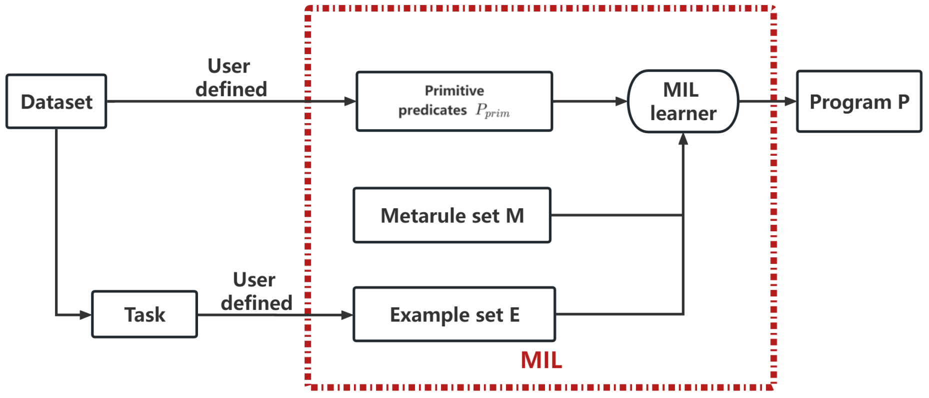


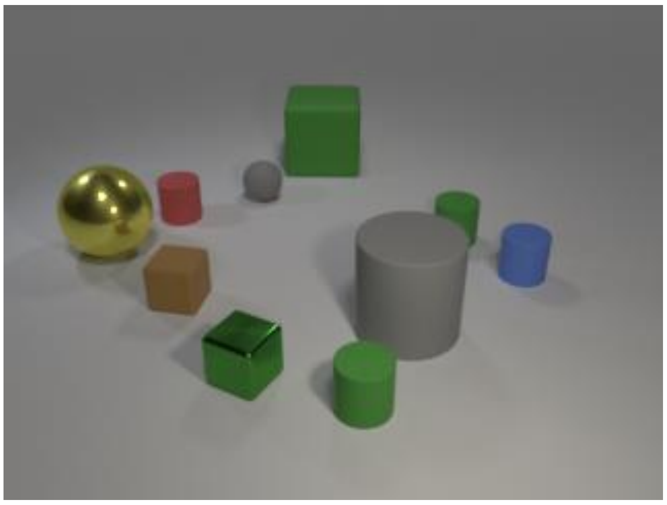
Figure 1. The framework of MIL.

Table 2. Prolog program of generalized meta-interpreter.

Generalized Meta-Interpreter

```
prove([],G,G).
prove(Atom | Atoms,G1,G2):-
   prove_aux(Atom,G1,G3), prove(Atoms,G3,G2).
prove_aux(Atom,G,G):- call(Atom).
prove_aux(Atom,G1,G4):-
   metarule(Name,MetaSub,(Atom:-Body)), OrderTest,
   save_subset(metasub(Name,MetaSub),G1,G3),
   prove(Body,G3,G2).
```

To illustrate the principle of MIL, a brief introduction to the application of a task is given. Figure 2 shows a task taken from the well-known CLEVR dataset [14]. There are some objects in the scene graph, a question that refers to the scene graph, an answer to the question, and a program that describes the reasoning process from the question to the answer. We now use the MIL method to accomplish this task.



Question:
What is the large gray cylinder made of?

Answer:
rubber

Program:
scene
filter_size
filter_color
filter_shape
unique
query_material

Figure 2. A VQA task from the CLEVR dataset.

Example 1. *For the VQA task shown in Figure 2, the MIL problem is a tuple* $input = \langle B, E^+, E^-, M\rangle$, *where* B, E^+, E^-, *and* M *are shown in Table 3. The solution to the MIL problem is to generate a program* H *that satisfies* $B, H \models e^+$ *for* $\forall e^+ \in E^+$ *and* $B, H \not\models e^-$ *for* $\forall e^- \in E^-$.

Table 3. MIL setting for Example 1.

Symbol	Name	MIL Setting
B	Background knowledge	scene/2, filter_color/2, filter_size/2, filter_shape, $\cdots$
$E = \langle E^+, E^-\rangle$	Example set	Positive example set $E^+ = \{f(Question, Answer)\}$ Negative example set $E^- = \emptyset$
M	Metarule set	*chain* metarule $P(X,Y)$:- $Q(X,Z), R(Z,Y)$. *ifthenelse* metarule $P(A,B)$:- $ifthenelse(A,B,Cond,Then,Else)$.

To accomplish this task, i.e., to generate a program that can account for how the *Answer* corresponding to the *Question* is derived, we first need to define the primitive predicates as the background knowledge, e.g., *filter_shape* and *filter_size*. With this basis, some examples in the form of $task(Question, Answer)$ about the task can be proven by MIL learner with the guidance of metarules, such as the *chain* metarule. Finally, these metarules will be instantiated into a logic program H.

MIL learners specifically use metarules to provide templates for learnable programs and leverage background knowledge to guide the hypothesis construction. The learning process is directed by SLD-resolution (selective linear definite clause resolution) [39], which implements a depth-first traversal search in the search space and iteratively refines hypotheses based on the provided positive and negative examples. Upon finding a program that satisfies the constraints, the MIL learner will halt and return the program it has learned. Table 4 demonstrates that the program generated by MIL consists of four clauses, where $f1$, $f2$, and $f3$ are invented predicates. Intuitively, with the capability of predicate invention, a Prolog program that answers the above question would first execute the predicates *filter_size*, *filter_color*, and *filter_shape* to filter all the large, grey, cylinder items in the picture. It will then check whether the object set is unique using the predicate *unique*. If so, the program will query and return the material of the target item using the predicate *query_material*. Given that the *scene* information has been encoded into the variables of the aforementioned predicates, no predicate named *scene* occurs in this program.

Table 4. Programs generated by MIL and MILER. Notes: for the programs generated by MILER, `aux_f1` and `aux_f2` are not predicates invented by the program learner, but rather auxiliary predicates captured by the predicate generator for reuse.

Program Learned by Original MIL
`f(A,B) :- filter_size(A,C), f1(C,B).`
`f1(A,B) :- filter_color(A,C),f2(C,B).`
`f2(A,B) :- filter_shape(A,C), f3(C,B).`
`f3(A,B) :- unique(A,C), query_material(C,B).`
Program Learned by MILER
`f(A,B) :- filter_size(A,C), f1(C,B).`
`f1(A,B) :- aux_f1(A,C), aux_f2(C,B).`
Auxiliary Predicates
`aux_f1(A,B) :- filter_color(A,C), filter_shape(C,B).`
`aux_f2(A,B) :-`
`    ifthenelse(unique(A,C),query_material(C,B),halt).`

3.4. Frequent Subgraph

Now, we will discuss how to encode programs into graphs and how to identify substructures within them. To do so, some useful definitions are presented.

Definition 3 (Substructures of programs [40])**.** *Let $P_s = \{p_i\}$ be a program set. Each program p_i can be encoded as a graph $G_i = (V_i, E_i)$ where the node set is composed of functions and edges represent the sequence between functions. A subgraph of G_i is defined as $G_s = (V_s, E_s)$ iff $V_s \subseteq V_i$ and $E_s \subseteq E_i \wedge (v_i, v_s) \in E_s \rightarrow v_i \in V_s$ and $v_s \in V_s$. If there is a path between any two nodes in G_s, we refer to it as a substructure of program p_i.*

Definition 4 (Reusable program patterns)**.** *Let P_s be a set of programs. A graph database is defined as $G = \{G_1, G_2, \cdots, G_n\}$ where each graph G_i is encoded according to the program $p_i \in P_s$. If there are two different programs p_i and p_j in the graph database G such that the graph G_s is a substructure of both programs p_i and p_j, it is said that the graph G_s is a reusable program pattern in the program set P_s.*

Definition 5 (Frequency of a program pattern)**.** *Let $G_S = \{G_s | G_s$ be a reusable program pattern in the program set $P_s\}$. For each graph G_s, the times program p_i (which satisfies that the graph G_s is a pattern of program p_i) occurs in P_s is defined as the frequency of G_s, denoted by $freq(G_s)$, in the program set P_s.*

4. Method

In this section, we introduce the motivation behind integrating frequent subgraph mining into MIL firstly, and then detail the framework and algorithm of MILER. Finally, we conduct a theoretical analysis of MILER.

4.1. Motivation

Considering the output program of Example 1 as shown in Table 4, although the invented predicate ($f1, f2, f3$) provides a hierarchical representation for MIL, they are still based on the primitive predicates *filter_size*, *filter_color*, *filter_shape*, *unique*, and *query_material*, which means that the learning cost associated with the invented predicates can be quite substantial. Especially when the number of primitive predicates is large, the program search space will exhibit exponential growth, and MIL will struggle in a vast search space and ultimately fail. In the VQA task discussed in Example 1, when the number of primitive predicates exceeds 10, the MIL will fail, while this task actually includes 22 primitive predicates.

An observable fact is that the conjunctions of predicates *filter_size* and *filter_color* resemble phrases, which frequently appear in other programs. With this basis, the intuitive idea is that we can actively look for opportunities to reuse recurring program patterns, namely *auxiliary predicates* which typically represent patterns that occur frequently (e.g., *aux_f1* and *aux_f2* shown in Table 4). This idea is not trivial, as many programs contain identifiable and reusable program patterns that can be reused, contributing to the scalability of MIL. It is natural that we can mine sub-structures allowed by the metarules from a small set of training samples for reuse and treat them as auxiliary predicates to expand the background knowledge of MIL.

4.2. MILER Problem

We now define the problem of Meta-Interpretive Learning with Reuse (MILER Problem).

Definition 6 (MILER Problem)**.** *An MILER problem is defined as* $Input = \langle\langle B, E^+, E^-, M\rangle, \{(x_i, y_i)\}\rangle$*. Here,* $\langle E^+, E^-, M\rangle$ *is configured in the same way as in MIL problem and B is the background knowledge that additionally includes auxiliary predicates.* $\{(x_i, y_i)\}$ *refers to a collection of task-related examples which have been solved (often referred to as the training set). In this context,* y_i *represents the solution (program) to the case* x_i*. The task of MILER is to learn a program H such that* $B, H \models E^+$ *for* $\forall e^+ \in E^+$ *and* $B, H \not\models e^-$ *for* $\forall e^- \in E^-$*.*

Example 2. *For the VQA problem described in Example 1, the MILER setting is a pair* $\langle\langle B, E^+, E^-, M\rangle, \{(x_i, y_i)\}\rangle$*. Here,* E^+*,* E^-*, and M are the same as those in Example 1. B is the background knowledge that additionally includes auxiliary predicates* $\{aux\_f1, aux\_f2\}$*.* $\{(x_i, y_i)\} = \{((s_i, q_i, a_i), p_i)\}$ *is the training set used by the predicate generator of MILER to extract reusable programs. An example of* $((s_i, q_i, a_i), p_i)$ *is shown in Figure 2, where* s_i*,* q_i*,* a_i*, and* p_i *represent Scene, Question, Answer, and Program, respectively.*

For the given MILER problem, our strategy is two-fold. First, we aim to enhance the background knowledge B by automatically extracting frequent program patterns from the training set $((s_i, q_i, a_i), p_i)$. Then, we apply optimized searching to solve the MIL problem. However, the process of identifying reusable program structures presents a significant challenge due to the difficulty in determining which structures are potentially reusable. To address this issue, we utilize the frequent subgraph mining algorithm to construct auxiliary predicates. This approach provides a systematic method for enriching our background knowledge, thereby facilitating the solution of the MIL problem.

4.3. Framework of MILER

Figure 3 illustrates the framework of MILER, which is composed of three components: *predicate generator* is responsible for identifying reusable program patterns from a small set of training samples and preserving them in the form of auxiliary predicates; *program learner* is responsible for accepting the auxiliary predicates in addition to learning the program with the related example set, metarule set, and background knowledge; *program evaluator* is responsible for providing an estimate of the accuracy of the learned program by testing set.

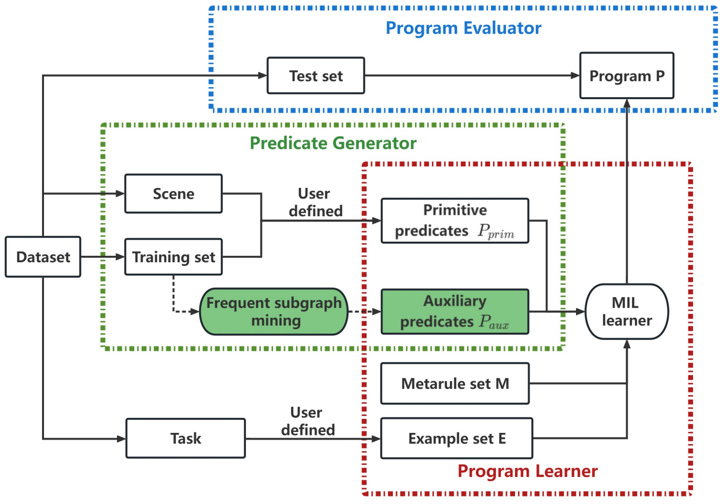


Figure 3. Framework of MILER.

4.3.1. Predicate Generator

Let $\{p_i|i = 1, 2, \cdots, n_{train}\}$ be a training program set. Then, we obtain an encoded graph set $G = \{g_i|i = 1, 2, \cdots, n_{train}\}$. By using frequent subgraph mining, we identify all the reusable program patterns G_s within the graph set G and their corresponding frequencies $freq(G_s)$. After sorting them based on the frequency of program patterns in descending order, we select the top 3 program patterns as the candidate set G_f to achieve a balance between efficiency and cost in MIL.

The elements of G_f, once further decoded into predicates known as auxiliary predicates, are then passed to the program learner.

Definition 7 (Auxiliary predicates)**.** *Let $G_f = (V_f, E_f)$ be a graph which describes a frequent program pattern. An auxiliary predicate can be defined based on the metarules as well as the functions $\{f_1, \cdots, f_k\}$ related to the set of nodes $V_f = \{v_1, \cdots, v_k\}$.*

Example 3 (Auxiliary predicates)**.** *Given a graph $G_f = (V_f, E_f)$ which describes an extracted frequent program pattern, where*

$$V_f = \{filter\_color, filter\_shape\}$$

and

$$E_f = \{(filter\_color, filter\_shape)\}.$$

An auxiliary predicate

$$\begin{aligned} aux\_f1(X, Y) :- filter\_color(X, Z), \\ filter\_shape(Z, U). \end{aligned}$$

can be extracted based on the chain metarule

$$P(X, Y) :- Q(X, Z), R(Z, Y).$$

Theorem 1 (Complexity of predicate generator)**.** *For a graph database $G = \{G_1, G_2, \cdots, G_n\}$, the maximum complexity of the predicate generator is $O(n * l^2) + O(n)$; here, l is the maximum number of nodes of a graph in G.*

Proof. The predicate generator includes two stages: mining all reusable program patterns and sorting them according to frequency. Let the number of programs in training set be n, and the size of the program (the number of functions) be l. Since the program database only contains two structures (sequence and tree), then for any program, the complexity for mining all of the reusable program patterns is $O(l^2)$. There are n programs in total, and thus, the maximum complexity of the predicate generator is $O(n * l^2) + O(n)$. □

4.3.2. Program Learner

In MILER, the program learner is essentially a basic MIL learner that accepts auxiliary predicates and adds them to the background knowledge B as *auxiliary background knowledge* B_{aux}.

Definition 8 (Auxiliary background knowledge)**.** *In the MILER problem,* $Input = \langle\langle B, E^+, E^-, M\rangle, \{(x_i, y_i)\}\rangle$*, the background knowledge* $B = \{Baux, Bcpl, Bint\}$ *consists of auxiliary background knowledge* $Baux$*, compiled background knowledge* $Bcpl$*, and interpreted background knowledge* $Bint$*.*

The compiled background knowledge $Bcpl$ (compiled Prolog code) and interpreted background knowledge $Bint$ (higher-order definitions) are defined in [12]. In MILER, the program learner will place auxiliary background knowledge $Baux$ in the prioritized position for searching the target program.

While the depth-first search mechanism of the MIL learner ensures that the program is minimal in terms of clause size, it does not guarantee that the program will be optimal in efficiency. Moreover, the learned program could even be incorrect, as it may only be valid for the in-sample data. Therefore, it is necessary to evaluate the learned program to demonstrate the effectiveness of our algorithm.

4.3.3. Program Evaluator

We utilize the case testing method, which is commonly used in the field of software engineering to evaluate the learned programs. Given that the learned programs of MILER are task-specific, aligning each program with the correct test set is critical to ensure the validity of the program evaluation. For each of the learned programs H, we collect test cases that match the program's task. We then check these cases against the definition of program H, subsequently calculating the proportion of test cases that conform to the definition. This proportion is used to measure the accuracy of the learned program.

4.4. Algorithm of MILER

To sum up, the pseudocode of the proposed algorithm MILER is illustrated in Algorithm 1. In the first phase, the predicate generator selects a number of programs from the training set T and encodes them into graphs, resulting in a set of graphs G. Subsequently, a frequent subgraph mining algorithm is applied to this set of graphs, returning the top 3 most frequently occurring program patterns top_G. These program patterns are then decoded into predicates, referred to as auxiliary predicates P_{aux}. The generated auxiliary predicates serve as broader background knowledge, enhancing the scalability of the MILER learner. During the program learning phase of MILER, these auxiliary predicates are assigned priorities. The learned program H will prioritize the use of auxiliary predicates if they are available; otherwise, it will continue to try other predicates. Once the program learner returns a program H, the program evaluator will match the corresponding test set T_{th} based on the *task* and calculate the accuracy of the learned program Acc_H.

Algorithm 1: Algorithm for MILER

```
Initialize: Example set $E = \langle E^+, E^- \rangle$; metarule set $M$, compiled background knowledge
            $Bcpl$, interpreted background knowledge $Bint$, $Baux = \emptyset$
Input     : Background knowledge $B$, example set $E$, metarule set $M$, training set
            $T = \{(x_i, y_i)|i = 1, \cdots, n_{train}\}$, test set $T_t = \{(x'_i, y'_i)|i = 1, \cdots, n_{test}\}$.
Output    : Program $H$.
begin
    // Predicate generator.
    $T = \{(x_i, y_i)\} = \{((s_i, q_i, a_i), p_i)\}$;
    $B_{aux} = \emptyset, G = \emptyset, top = 3$;
    for $p_i \in subset(T)$ do
        $G_i = graph(p_i)$;
        $G = G \cup G_i$;
    end
    $top_G = frequentSubgraph(G, top)$;
    $P_{aux} = predicate(top_G)$;
    $B_{aux} = P_{aux}$;
    $B = B_{aux} \cup B_{cpl} \cup B_{int}$;
    // Program learner.
    for $e^- \in E^-$ do
        for $e^+ \in E^+$ do
            $B, H \models E^+$;
        end
        $B, H \not\models e^-$;
    end
    // Program evaluator.
    if $H$ exists then
        $T_t = \{(x'_i, y'_i)\} = \{((s'_i, q'_i, a'_i), p'_i)\}$;
        $T_{th} = matchset(H, T_t)$;
        $Count = 0$;
        for $t \in T_{th}$ do
            if $instance(t, H)$ then
                Count = Count+1
            end
        end
        $Acc_H = Count/|T_{th}|$;
    end
    return $H, Acc_H$;
end
Function $frequentSubgraph(G, top)$:
    $G = \{g_i\}$;
    $G_S = \emptyset$;
    for each pair $(g_i, g_j) \in G$ do
        $G_s = subgraph(g_i, g_j)$;
        if $reusable(G_s)$ then
            $G_S = G_S \cup \{G_s\}$;
        end
    end
    $t(G) = \{G_s : freq(G_s)\}$;
    $top_G = sortbyfreq(t(G), top)$;
    return $top_G$;
end
```

4.5. Theoretical Analysis of MILER

In this section, we analyze the advantages of the MILER system theoretically.

Lemma 1 (MIL program space [12])**.** *Given p predicate symbols and m metarules in the fragment M^i_j, a metarule is in M^i_j if it has at most j literals in the body and each literal has at most i arities. The number of programs expressible with n clauses is at most $(mp^{j+1})^n$.*

Compared with the original MIL system, our MILER system can reduce the number of clauses required in the program by adding t auxiliary predicates and reducing the range of primitive predicates according to the examples in the training set. Since MIL performs the depth-first traversal search, MILER is more likely to learn the program with fewer clauses than the MIL system. Based on Lemma 1, suppose that the MILER system can reduce k clauses, then the program space of MILER will follow Lemma 2.

Lemma 2 (MILER program space)**.** *Given $p' + t$ predicate symbols and m metarules in the fragment M^i_j, a metarule is in M^i_j if it has at most j literals in the body and each literal has at most i arities. The number of programs expressible with $n - k$ clauses is at most $(m(p' + t)^{j+1})^{n-k}$.*

Example 4 (Program space of MIL and MILER)**.** *Following Example 1, given the information of the image (shown in Figure 2), question, answer, program, and scene, we can define the input as $Input = \langle B, E^+, E^-, M\rangle$ for the MIL setting, where $E = \langle E^+, E^-\rangle$ denotes the example set, and M denotes the set of metarules. The background knowledge of the MIL setting is defined as $B_{MIL} = \{$ scene/2, filter_color/2,filter_shape/2, filter_size/2, filter_material/2, $\cdots\}$, and the background knowledge of the MILER setting is defined as $B_{MILER} = P_{aux} \cup B_{MIL}$, where $P_{aux} = \{aux\_f1, aux\_f2\}$.*

We can see that the programs learned by MIL and MILER are presented in Table 4. The number of primitive predicates is $p = 19$ and the number of auxiliary predicates is $t = 3$. Here, we utilize the *chain* metarule and $ifthenelse$ metarule, with $m = 2$, $j = 3$, and $i = 2$. By Lemma 1, the program space of MIL with $n = 4$ clauses is $(mp^{j+1})^n = 2 * 19^4)^4$ at most. By Lemma 2 and $p' \leq p$, the program space of MILER with $n - k = 2$ clauses is at most $(m(p' + t)^{j+1})^{n-k} \leq (2 * (22)^4)^2$. Obviously, the program space of MIL is much larger than that of MILER.

5. Experiments

In order to evaluate the proposed approach MILER, we use the original MIL system as the baseline and measure the performance improvement of MILER through the availability ratio and the clause reduction ratio. These experiments were conducted using SWI-Prolog (v8.4.1 for x64-win64), a widely used software tool in the field, to aid our analysis. We evaluate MILER on two VQA tasks (i.e., those used in [14,15]) and one string program synthesis task [16]. The experiments are carried out on a Windows system with Intel processors running at 2.80 GHz and 32 GB RAM.

For the three sets of experiments, we designed 1000, 500, and 40 tasks for them to learn programs, respectively. When the number of training samples is set to 50, the number (percentage) of tasks successfully learned by MIL is 584 (58.4%), 122 (24.4%), and 26 (65%), while the number (percentage) of tasks successfully learned by MILER is 779 (77.9%), 255 (51.0%) and 34 (85%).

We now define the evaluation metrics: *availability ratio* and *clause reduction ratio*. Note that these metrics are computed without considering timeout cases.

$$availability\ ratio = \frac{N_p^{MILER_{aux}}}{N_p^{MIL}} \tag{1}$$

Here, $N_p^{MILER_{aux}}$ denotes the number of programs learned by MILER with auxiliary predicates, and N_p^{MIL} denotes the number of programs learned by MIL.

$$\textit{clause reduction ratio} = \frac{N_c^{MIL} - N_c^{MILER}}{N_c^{MIL}} \tag{2}$$

Here, N_c^{MIL} and N_c^{MILER} indicate the number of clauses contained in the programs learned by MIL and MILER, respectively.

5.1. CLEVR Dataset

CLEVR [14,41] is a dataset comprising synthetic images of 3D primitives with multiple attributes: shape, color, material, size, and 3D coordinates. Each image has a set of questions, each corresponding to a program (a set of symbolic modules) generated by machines based on 90 logic templates.

We randomly select x samples from 150,000 program samples, where x ranges from 20 to 400, to form the training set. We extract the top 3 auxiliary predicates from the training set by the subgraph mining algorithm. Subsequently, we choose 100 related programs as the test set to evaluate the accuracy of the learned programs. We set a timeout termination of 60 s and repeat the experiment 10 times to calculate the average values of the availability ratio, the clause reduction ratio of MILER, and the average accuracy of the program learned by MILER.

Figure 4 demonstrates the availability ratio and clause reduction ratio of MILER for CLEVR. It can be seen from the results that when the sample size of the training set is around 100, MILER can achieve a stable availability rate of over 70% and a clause reduction ratio around 10% when the auxiliary predicates can be reused. The higher availability ratio suggests that MILER indeed identifies reusable program patterns from a small dataset. The clause reduction ratio implies that the program learned by MIL possesses fewer clauses compared to the original MIL. Figure 5 shows that the program accuracy rate ranges from 87% to 98%, which is consistent with the original MIL method. This is because the programs that contain auxiliary predicates learned by MILER are logically equivalent to the programs learned by MIL.

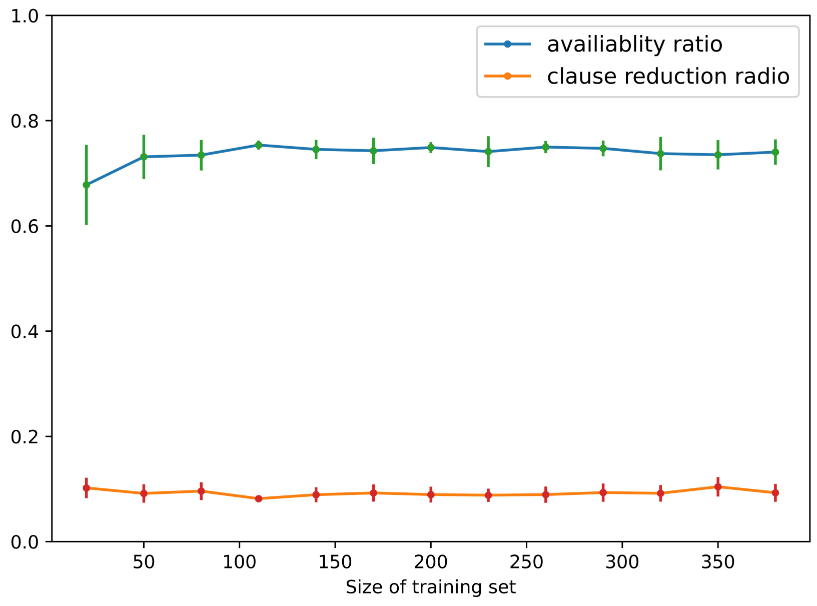


Figure 4. Availability ratio and clause reduction ratio of MILER for CLEVR.

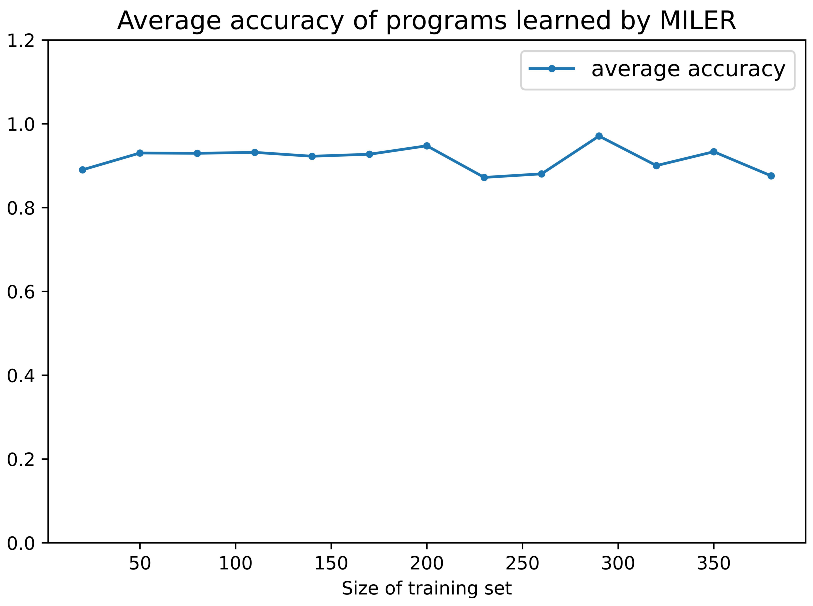


Figure 5. Average accuracy of programs learned by MILER in CLEVR.

We also explore cases where the MIL learning failed (due to timeouts), while MILER learning succeeded, facilitated by the introduction of auxiliary predicates. Table 5 displays some of the cases generated by MIL and MILER. For the CLEVR dataset, the first example in the table No. 1 shows that, when MIL fails (timeout) to learn a program due to a vast search space, MILER can still generate the program using auxiliary predicates. Other examples (No. 2∼5) indicate that the MILER system can generate more compact programs than MIL. On the other hand, according to our observation, even if the extracted auxiliary predicates are unavailable for a program, the MILER system can still generate the same program as the original MIL system, as demonstrated in the last case No. 6 of Table 5. In this experiment, as illustrated in No. 2 of Table 5, it shows that, with the addition of auxiliary predicates, MILER may incur extra time overhead.

5.2. *Minecraft World*

We now discuss another VQA dataset that features objects and scenes from Minecraft World. To generate the dataset, we use the dataset generation tool provided by [42] to render 600 Minecraft World scenes, building upon the Malmo interface [43]. We utilize the same configuration details as suggested by Wu et al., except that we do not render the images, which are not part of the input of MIL. Each scene consists of 5–10 objects, and each object is sampled from a set of 12 entities. Our structural representation has the following fields for each object: category (12-dim), position in the 2D plane (2-dim, $\{x, z\}$), and the direction the object faces front, back, left, right.

The questions and programs associated with each Minecraft World image are generated based on the objects' categorical and spatial attributes (position, direction). We use 600 scenes with 1800 question–program pairs, and randomly select $x \in (20,300)$ samples as the training set, and another 600 programs from the remaining program samples as the test set to calculate the availability ratio and clause reduction ratio of the generated auxiliary predicates. We set a timeout termination of 60 seconds and obtain the results after repeating the experiment 10 times.

Table 5. Partial cases of the learning time and size of programs generated by MIL and MILER.

No.	**CLEVR**				**Minecraft World**				**String Program Synthesis**			
	Learning Time (Seconds)		**Program Size**		**Learning Time (Seconds)**		**Program Size**		**Learning Time (Seconds)**		**Program Size**	
	MIL	**MILER**	**MIL**	**MILER**	**MIL**	**MILER**	**MIL**	**MILER**	**MIL**	**MILER**	**MIL**	**MILER**
1	**Timeout**	0.046875	-	2	**Timeout**	0.015625	-	2	**Timeout**	0.0625	-	2
2	0.375	0.0625	5	3	0.0625	0.046875	3	2	0.0625	0.046875	2	1
3	0.046875	0.0625	3	1	0.078125	0.0625	3	2	0.078125	0.046875	2	1
4	0.03125	0.046875	3	1	0.0625	0.046875	2	1	0.0675	0.046875	2	1
5	0.078125	0.0625	2	1	0.03125	0.046875	3	1	0.078125	0.046875	3	2
6	0.0625	0.0625	4	4	0.0625	0.046875	3	2	0.0625	0.046875	2	1

Figure 6 shows that, when the size of the training set is 20∼300, MILER can achieve a stable availability ratio of auxiliary predicates and a clause reduction ratio of approximately 50% and 10%, respectively. The mediocre availability ratio is attributed to the diversity of entities and attributes contained in this dataset, which complicates the task of mining reusable program patterns. Figure 7 shows the average accuracy of programs learned by MILER in Minecraft, and the accuracy of the programs ranges from 93% to 98.9%, consistently with the original MIL method.

For Minecraft World dataset, we also explored cases where MIL learning failed while MILER learning succeeded. For example, in Table 5, the first case $No.1$ shows the timeout of MIL and the success of MILER. As for the cases where both methods succeeded in learning, like No. 2∼6, they indicate that MILER can typically reduce the clause from $\{2,3\}$ to $\{1,2\}$, aligning with our theoretical clause reduction ratio.

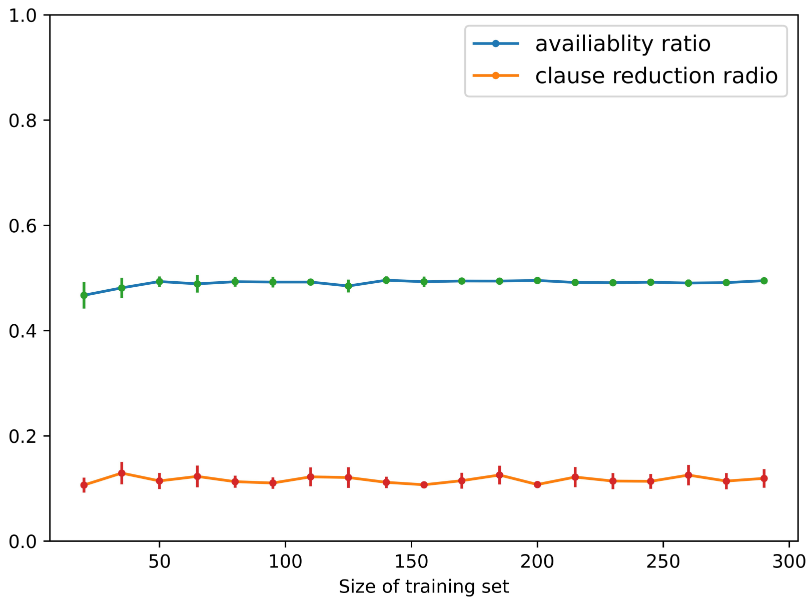


Figure 6. Availability ratio and clause reduction ratio of MILER for Minecraft World.

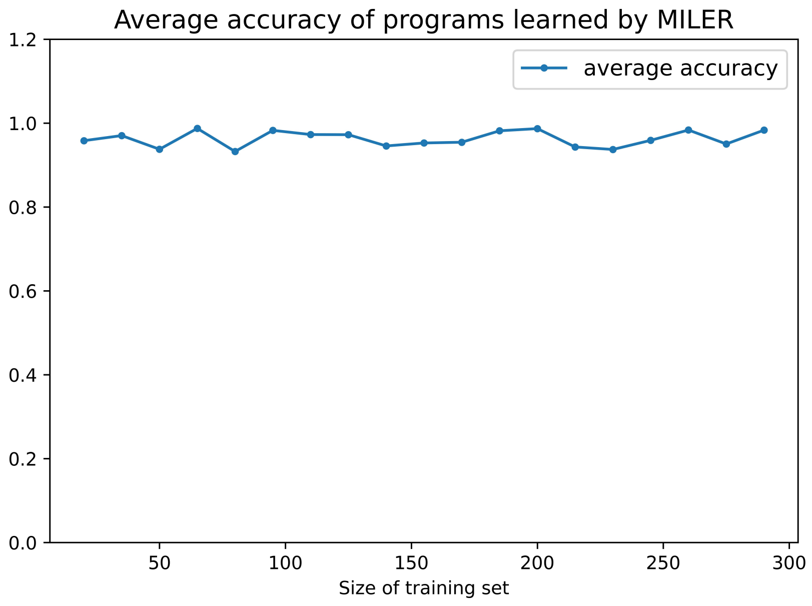


Figure 7. Average accuracy of programs learned by MILER in Minecraft.

5.3. String Program Synthesis

String program synthesis is a task in the field of program synthesis and artificial intelligence, where the goal is to automatically generate a program that can correctly process and manipulate string data based on the given specifications.

We use the dataset from Euphony [16]. The String benchmarks comprise 205 tasks, divided into two parts: 108 problems from the 2017 SyGus competition PBE string track (involving transformations from string to string) and 97 problems from StackOverflow and Exceljet (either have a non-string parameter or the synthesis of a numeric or Boolean expression). Table 6 shows two examples of string programs from the dataset. In the MILER setting

$$\langle \langle B, E^+, E^-, M \rangle, \{((s_i, q_i, a_i), p_i)\} \rangle,$$

where $\{((s_i, q_i, a_i), p_i)\} = \{(([], input\_string\_list, out\_string\_list), string\_program)\}$.

Table 6. Examples of string program from the dataset.

(**define-fun** f((name String)) String (**str.substr** name 0 (- (str.len name) 3)))
(**define-fun** f((_arg_0 String) (_arg_1 String) (_arg_2 String) (_arg_3 String)) Bool (**str.contains** _arg_0 _arg_3))

In this string program synthesis experiment, we divide the programs into a training set and a learning set according to a ratio of 4:1. In order to convert the programs in the training set into medium-scale graphs, we reserve the position of variables in the string program and replace them with constants. After 10 rounds of repetition and with a timeout termination set at 60 s, we obtained the results regarding the availability ratio and clause reduction ratio.

In Figure 8, the results show that, when the training size is around 20, MILER can achieve the best availability ratio and clause reduction ratio of auxiliary predicates in the field of string program synthesis, e.g., 60% and 18.3%. Table 7 provides the examples of the reusable auxiliary predicates extracted by MILER and the program learned by MILER contains an auxiliary predicate aux_f1. Figure 9 shows the average accuracy of programs learned by MILER in String Program Synthesis. The average accuracy of programs reaches 100%, which is consistent with the original MIL method.

Table 7. Examples of the extracted auxiliary predicates and program learned by MILER for String Program Synthesis.

Extracted Reusable Structure

`define f arg_zero int string int_to_str arg_zero`
`define f arg_zero string bool str_contains arg_zero`

Auxiliary Predicates Generated by MILER
aux_f1`(A,B) :- int(A,C), string(C,B).` **aux_f2**`(A,B) :- string(A,C), str_contains(C,B).`
Program Learned by MILER
`f(A,B) : - aux_f1(A,C), int_to_str(C,B).`

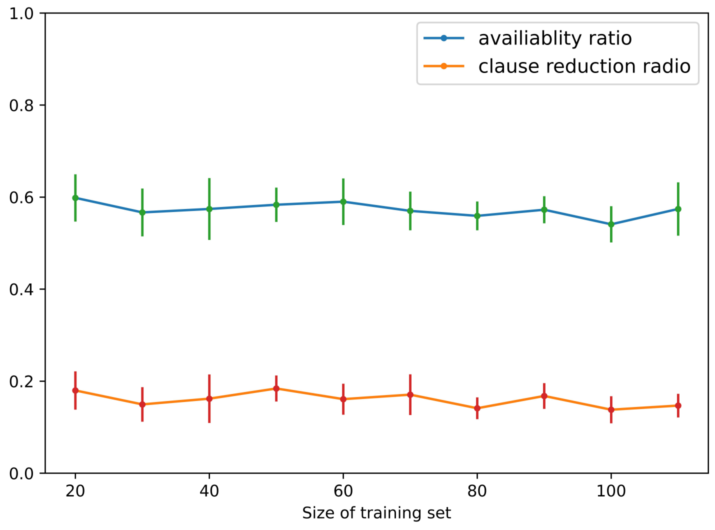


Figure 8. Availability ratio and clause reduction ratio of MILER for String Program Synthesis.

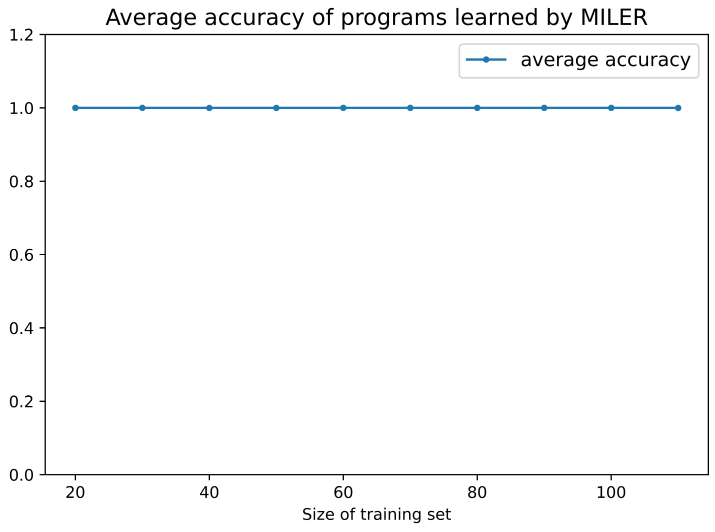


Figure 9. Average accuracy of programs learned by MILER in String Program Synthesis.

For the results of String Program Synthesis in Table 5, the first case No. 1 exhibits MIL learning fails, whereas MILER can generate a program with the help of auxiliary predicates. According to the observation, such as the examples No. 2∼6, MILER can typically reduce the program by 1∼2 clauses.

6. Conclusions

In this paper, we introduce a new approach, MILER, to enhance the scalability of MIL. MILER first uses subgraph mining to search for reusable program patterns from the training set, which is composed of program sequences, and then sorts them according to the frequency of these program patterns in the training set. It selects the top three program patterns and represents them in the form of auxiliary predicates allowed by metarules. After incorporating the auxiliary predicates into the background knowledge, MILER will try to learn more compact programs with the assistance of auxiliary predicates. We use test samples to evaluate the accuracy of learned programs. We also theoretically demonstrate that the addition of auxiliary predicates can reduce the complexity of the program space. To evaluate the effectiveness of our approach, we conducted three experiments: two VQA tasks and a string program synthesis task. The experimental results show that MILER is able to mine the structural information in the training set and represent it in the form of auxiliary predicates. MILER can learn from a small number of training data and extract useful auxiliary predicates that can be used to optimize the learning process. Additionally, MILER can learn a more compact program with fewer clauses.

Our approach has two main limitations. First, the auxiliary predicates introduced depend on a small set of training examples, which may lead to a lack of robustness, especially on different tasks where the performance may vary significantly. Second, the number of auxiliary predicates introduced is based on empirical settings, which requires a balance between efficiency and cost, making it a challenging task. Future efforts will focus on refining the form of metarules, such as introducing the specialized metarules better aligned with auxiliary predicates, to enhance the MIL learner's ability to prioritize searches within these predicates.

Author Contributions: Methodology, R.W.; supervision, J.S., C.T. and Z.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research is supported by the National Natural Science Foundation of China with Grant No. 62192734.

Data Availability Statement: Publicly available datasets were analyzed in this study. This data can be found here: https://cs.stanford.edu/people/jcjohns/clevr/ and https://github.com/jiajunwu/nsd (accessed on 19 February 2024).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Garnelo, M.; Shanahan, M. Reconciling deep learning with symbolic artificial intelligence: Representing objects and relations. *Curr. Opin. Behav. Sci.* **2019**, *29*, 17–23. [CrossRef]
2. Muggleton, S.; De Raedt, L. Inductive logic programming: Theory and methods. *J. Log. Program.* **1994**, *19*, 629–679. [CrossRef]
3. Cropper, A.; Dumančić, S.; Muggleton, S.H. Turning 30: New ideas in inductive logic programming. *arXiv* **2020**, arXiv:2002.11002.
4. Muggleton, S.; De Raedt, L.; Poole, D.; Bratko, I.; Flach, P.; Inoue, K.; Srinivasan, A. ILP turns 20. *Mach. Learn.* **2012**, *86*, 3–23. [CrossRef]
5. Dai, W.Z.; Hallett, L.; Muggleton, S.H.; Baldwin, G.S. Automated Biodesign Engineering by Abductive Meta-Interpretive Learning. *arXiv* **2021**, arXiv:2105.07758.
6. Muggleton, S.H.; Lin, D.; Pahlavi, N.; Tamaddoni-Nezhad, A. Meta-interpretive learning: Application to grammatical inference. *Mach. Learn.* **2014**, *94*, 25–49. [CrossRef]
7. Milani, G.A.; Cyrus, D.; Tamaddoni-Nezhad, A. Towards One-Shot Learning for Text Classification using Inductive Logic Programming. *arXiv* **2023**, arXiv:2308.15885.
8. Muggleton, S.; Dai, W.Z.; Sammut, C.; Tamaddoni-Nezhad, A.; Wen, J.; Zhou, Z.H. Meta-interpretive learning from noisy images. *Mach. Learn.* **2018**, *107*, 1097–1118. [CrossRef]
9. Dai, W.Z.; Muggleton, S.; Wen, J.; Tamaddoni-Nezhad, A.; Zhou, Z.H. Logical vision: One-shot meta-interpretive learning from real images. In Proceedings of the International Conference on Inductive Logic Programming, Bari, Italy, 13–15 November 2017; pp. 46–62.
10. Cyrus, D.; Trewern, J.; Tamaddoni-Nezhad, A. Meta-interpretive Learning from Fractal Images. In Proceedings of the International Conference on Inductive Logic Programming, Bari, Italy, 13–15 November 2017; pp. 166–174.
11. Lin, D.; Dechter, E.; Ellis, K.; Tenenbaum, J.B.; Muggleton, S.H. Bias reformulation for one-shot function induction. *Front. Artif. Intell. Appl.* **2014**, *263* , 525–530. . [CrossRef]
12. Cropper, A.; Muggleton, S.H. Learning Higher-Order Logic Programs through Abstraction and Invention. In Proceedings of the IJCAI, New York, NY, USA, 9–15 July 2016; pp. 1418–1424.
13. Morel, R.; Cropper, A.; Ong, C.H.L. Typed meta-interpretive learning of logic programs. In Proceedings of the Logics in Artificial Intelligence: 16th European Conference, JELIA 2019, Rende, Italy, 7–11 May 2019; pp. 198–213.
14. Johnson, J.; Hariharan, B.; Van Der Maaten, L.; Fei-Fei, L.; Lawrence Zitnick, C.; Girshick, R. Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Honolulu, HI, USA, 21–26 July 2017; pp. 2901–2910.
15. Yi, K.; Wu, J.; Gan, C.; Torralba, A.; Kohli, P.; Tenenbaum, J. Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. *Adv. Neural Inf. Process. Syst.* **2018**, *31*, 1.
16. Lee, W.; Heo, K.; Alur, R.; Naik, M. Accelerating search-based program synthesis using learned probabilistic models. *ACM SIGPLAN Not.* **2018**, *53*, 436–449. [CrossRef]
17. Plotkin, G.D. A note on inductive generalization. *Mach. Intell.* **1970**, *5*, 153–163.
18. Plotkin, G. Automatic Methods of Inductive Inference. Ph.D. Thesis, Univerzitet u Edinburghu, Edinburgh, UK, 1972.

19. Shapiro, S. Conservativeness and incompleteness. *J. Philos.* **1983**, *80*, 521–531. [CrossRef]
20. Sammut, C.; Banerji, R.B. Learning concepts by asking questions. *Mach. Learn. Artif. Intell. Approach* **1986**, *2*, 167–192.
21. Quinlan, J.R.; Cameron-Jones, R.M. Induction of logic programs: FOIL and related systems. *New Gener. Comput.* **1995**, *13*, 287–312. [CrossRef]
22. Muggleton, S.H.; Feng, C. *Efficient Induction of Logic Programs*; Turing Institute: London, UK, 1990.
23. Muggleton, S. Inverse entailment and Progol. *New Gener. Comput.* **1995**, *13*, 245–286. [CrossRef]
24. Blockeel, H.; De Raedt, L.; Ramon, J. Top-down induction of clustering trees. *arXiv* **2000**, arXiv:cs/0011032.
25. Srinivasan, A. The aleph manual. **2001**, *1*, 1–66.
26. Muggleton, S.H.; Lin, D.; Tamaddoni-Nezhad, A. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Mach. Learn.* **2015**, *100*, 49–73. [CrossRef]
27. Cropper, A.; Muggleton, S.H. Logical Minimisation of Meta-Rules Within Meta-Interpretive Learning. In Proceedings of the Revised Selected Papers of the 24th International Conference on Inductive Logic Programming, Nancy, France, 14–16 September 2014; pp. 62–75. [CrossRef]
28. Cropper, A.; Tourret, S. Derivation reduction of metarules in meta-interpretive learning. In Proceedings of the International Conference on Inductive Logic Programming, Ferrara, Italy, 1 September 2018; pp. 1–21. [CrossRef]
29. Cohn, A.G.; Hogg, D.C.; Bennett, B.; Devin, V.; Galata, A.; Magee, D.R.; Needham, C.; Santos, P. Cognitive vision: Integrating symbolic qualitative representations with computer vision. In *Cognitive Vision Systems*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 221–246.
30. Antanas, L.; Van Otterlo, M.; Mogrovejo, J.O.; Tuytelaars, T.; De Raedt, L. There are plenty of places like home: Using relational representations in hierarchies for distance-based image understanding. *Neurocomputing* **2014**, *123*, 75–85. [CrossRef]
31. Dai, W.Z.; Muggleton, S.H.; Zhou, Z.H. Logical vision: Meta-interpretive learning for simple geometrical concepts. In Proceedings of the ILP (Late Breaking Papers), Kyoto, Japan, 20–22 August 2015.
32. Antol, S.; Agrawal, A.; Lu, J.; Mitchell, M.; Batra, D.; Zitnick, C.L.; Parikh, D. Vqa: Visual question answering. In Proceedings of the IEEE international Conference on Computer Vision, Santiago, Chile, 7–13 December 2015; pp. 2425–2433.
33. Anderson, P.; He, X.; Buehler, C.; Teney, D.; Johnson, M.; Gould, S.; Zhang, L. Bottom-up and top-down attention for image captioning and visual question answering. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 18–22 June 2018; pp. 6077–6086.
34. Tan, H.; Bansal, M. Lxmert: Learning cross-modality encoder representations from transformers. *arXiv* **2019**, arXiv:1908.07490.
35. Johnson, J.; Hariharan, B.; Van Der Maaten, L.; Hoffman, J.; Fei-Fei, L.; Lawrence Zitnick, C.; Girshick, R. Inferring and executing programs for visual reasoning. In Proceedings of the IEEE International Conference on Computer Vision, Venice, Italy, 22–29 October 2017; pp. 2989–2998.
36. Vani, A.; Schwarzer, M.; Lu, Y.; Dhekane, E.; Courville, A. Iterated learning for emergent systematicity in vqa. *arXiv* **2021**, arXiv:2105.01119.
37. Li, J.; Li, Y.; Li, G.; Jin, Z.; Hao, Y.; Hu, X. Skcoder: A sketch-based approach for automatic code generation. *arXiv* **2023**, arXiv:2302.06144.
38. Cropper, A.; Tamaddoni-Nezhad, A.; Muggleton, S. Meta-Interpretive Learning of Data Transformation Programs. In Proceedings of the Inductive Logic Programming, Bordeaux, France, 5–8 July 2016; pp. 46–59. [CrossRef]
39. Mukaidono, M.; Kikuchi, H. Foundations of fuzzy logic programming. In *Between Mind and Computer: Fuzzy Science and Engineering*; World Scientific: Singapore, 1993; pp. 225–244.
40. Elseidy, M.; Abdelhamid, E.; Skiadopoulos, S.; Kalnis, P. Grami: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow.* **2014**, *7*, 517–528. [CrossRef]
41. Bahdanau, D.; de Vries, H.; O'Donnell, T.J.; Murty, S.; Beaudoin, P.; Bengio, Y.; Courville, A. Closure: Assessing systematic generalization of clevr models. *arXiv* **2019**, arXiv:1912.05783.
42. Wu, J.; Tenenbaum, J.B.; Kohli, P. Neural scene de-rendering. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Honolulu, HI, USA, 21–26 July 2017; pp. 699–707.
43. Johnson, M.; Hofmann, K.; Hutton, T.; Bignell, D. The Malmo Platform for Artificial Intelligence Experimentation. In Proceedings of the IJCAI, New York, NY, USA, 9–15 July 2016; pp. 4246–4247.