Meta-Interpretive learning as Second Order Resolution

James Trewern[1], Stassa Patsantzis[1], and Alireza Tamaddoni Nezhad[1]

Department of Computer Science, University of Surrey, Guildford, UK
{jt00988, s.patsantzis, a.tamaddoni-nezhad}@surrey.ac.uk

Abstract. Meta-Interpretive Learning (MIL), a branch of Inductive Logic programming (ILP), has been shown to be an effective method for learning logic programs, through the use of higher-order 'meta-rules', from which first order clauses can be built using meta-substitutions. In this paper we propose a new framework for understanding MIL as Second Order SLD-Resolution which we prove is sound and complete for induction, as for deduction. We implement two new learning and reasoning engines for MIL, one called Vanilla implemented in Prolog, and the other, Prolog^2 implemented in Rust. We use Vanilla and Prolog^2 to implement Top Program Construction (TPC). TPC in Prolog^2 is a multi-threaded implementation, processing multiple instances of Second Order SLD-Resolution in parallel. Our results show that these systems can learn hundreds of clauses from over 600 examples in a matter of minutes for Vanilla-TPC, or Seconds in Prolog^2-TPC.

Keywords: Inductive Logic Programming · Meta-Interpretive Learning · SLD-Resolution · Top Program Construction · Second Order Resolution

1 Introduction

Meta-Interpretive Learning. Meta-Interpretive Learning (MIL) [14, 12] is an advanced form of Inductive Logic Programming (ILP) [13] where first-order logic hypotheses are learned by inductive generalisation from examples and a higher-order background theory comprising both first- and Second-Order definite clauses. Examples, background theory and learned hypotheses are all in the form of logic programs in a logic programming language, typically, Prolog.

Second-order SLD-Resolution. MIL systems are capable of learning, from a single positive example, complex logic programs with recursion and invented predicates (ones not included in the background theory, or examples [23]) and that generalise robustly to unseen instances. The present article formalises the understanding that this learning ability is the result of the use of SLD-Resolution [8, 10] as the learning procedure: by the soundness and completeness of SLD-Resolution [10, 15], given that an example is a logical consequence of the higher-order background theory, a first order program can always be constructed, by

applying to the clauses in the background theory substitutions of their Second-Order variables found by unification during SLD-Refutation of the example, and that program must necessarily entail the example with respect to the first-order background theory. Over-general hypotheses can be identified and rejected, or reformulated, by SLD-Refutation of negative examples. The inclusion of Second-Order definite clauses in the background theory raises SLD-Resolution to the second order of logic, where deduction and induction, learning and reasoning, become one.

Contributions. In this paper, we make the following contributions:

- **Theoretical results** A new recognition of MIL as Second-Order SLD-Resolution; proofs of *inductive* soundness and completeness of MIL.
- **Implementation** Two new learning and reasoning engines for MIL called *Vanilla* and *Prolog*^{2} informed by the new understanding of MIL as Second-Order SLD-Resolution; Vanilla-TPC and Prolog^2-TPC , new implementations, of state-of-the-art MIL algorithm Top Program Construction.

2 Related Work

Meta-Interpretive Learning. Previous descriptions of MIL in the literature fail to clearly identify it as Second-Order SLD-Resolution. Such descriptions point instead to the implementation of MIL by means of a "Prolog meta-interpreter" modified to "fetch higher-order clauses" named "metarules" (e.g. in [9, 4, 2]). It is possible, for the discerning reader, to recognise MIL as Second-Order SLD-Resolution if only by "reading between the lines" of such descriptions: a "Prolog meta-interpreter" is a coding, in Prolog, of Prolog, and, therefore, of SLD-Resolution; and "fetching metarules" indicates that the "metarules" are *resolved with* as definite clauses, rather than acting as mere templates or syntactic bias —as they are described e.g in [3], [7]). [17] define MIL as "metarule specialisation by SLD-Resolution"; which again fails to fully clarify the issue. In Section 3 we explicitly recognise MIL as Second-Order SLD-Resolution for the first time; and show, also for the first time, its soundness and completeness that follow from this new recognition.

Recursion and Predicate Invention. Demonstrations of learning of recursive logic programs with invented predicates, some from single examples, abound in the MIL literature, e.g. [14, 12, 11]. By contrast, earlier ILP approaches are only capable of limited forms of learning recursion and predicate invention. A comprehensive review of such limited capabilities of earlier systems is found in [5]. In Section 3 we prove that MIL is capable of learning arbitrary logic programs with no restriction to the structure of recursive clauses thanks to the soundness and completeness of SLD-Resolution. We leave an extension of our proof to programs with invented predicates for future work.

3 Framework

In this Section we introduce a novel framework for MIL as Second-Order SLD Resolution and prove its soundness and completeness.

3.1 Logical Notation

We start with the logical notation used throughout the Section. Our notation follows closely from [10] and [15], which we extend with definitions for Second-Order definite clauses.

FOL language. We define a First-Order Logic (FOL) language as an alphabet consisting of sets of predicate symbols $\mathcal{P} = \{P, Q, R, ...\}$, function symbols $\mathcal{F} = \{f, g, h, ...\}$, constants $\mathcal{C} \subseteq \mathcal{F} = \{a, b, c, ...\}$, variables $\mathcal{V} = \{x, y, z, ...\}$, logical connectors $\neg, \wedge, \vee, \rightarrow, \Leftrightarrow$, quantifiers $\exists, \forall$ and punctuation symbols . , (and). Symbols have *arity*, a natural number; Constants have arity 0. *Terms* are defined inductively as follows: a variable is a term; a constant is a term; if f is a function symbol of arity n and $t_1, .., t_n$ are terms, $f(t_1, ..., t_n)$ is a term. *Formulae* are defined inductively as follows: if P is a predicate symbol of arity m and $t_1, ..., t_m$ are terms, $P(t_1, ..., t_m)$ is a formula, called an *atom*. If ϕ, ψ are formulae then $\neg\phi, \phi \wedge \psi, \phi \vee \psi, \phi \rightarrow \psi, \phi \Leftrightarrow \psi$ are formulae; if x is a variable in formula ϕ then $\exists x : \phi, \forall x : \phi$ are formulae. A formula is *ground* when it has no variables.

Logic Programming. A *literal* is an atom, or the negation of an atom; if A is an atom, A is a positive literal and $\neg A$ is a negative literal. A *clause* is a set of literals in disjunction: $\{L_1 \vee L_2 \vee L_3 \vee ...\}$; $\square$ is the empty clause. A clause is Horn if it has at most one positive literal and *Prolog* if all its variables are universally quantified; in which case we omit quantifiers. A Horn clause is a *definite program clause*, or simply *definite clause*, if it has exactly one positive literal, otherwise it is a Horn *goal*. A *definite program*, or *logic program*, is a set of definite clauses in conjunction: $\{C_1 \wedge C_2 \wedge C_3 \wedge ...\}$. By logic programming conventions we will write a definite clause, e.g., $\{P(x, y), \neg Q(x, z), \neg R(z, y)\}$ as an implication with the single positive literal first and the implication arrow reversed: $P(x, y) \leftarrow Q(x, z), R(z, y)$ under the standard semantics of disjunction where $A \vee \neg B \Leftrightarrow B \rightarrow A$. We will call the single consequent literal, $P(x, y)$, the "head" of the clause and all other literals, $Q(x, z), R(z, y)$, the "body" of the clause. We will denote a Horn goal $G = \{\neg L_1 \vee \neg L_2 \vee ...\}$ as $\leftarrow G$.

Higher-Order Datalog. A Horn clause is *Datalog* if it has no function symbols other than constants and each variable in the head of the clause is also found in a literal in the body of the clause; we say the head and body literal "share" the variable. A Datalog clause is a *Second-Order clause* if it has variables existentially quantified over $\mathcal{P}$ in place of predicate symbols. A Second-Order clause may have variables existentially or universally quantified over $\mathcal{C}, \mathcal{V}$. A set of Second-Order definite clauses is a *Second-Order definite program*. By MIL convention we will denote the quantification of variables in Second-Order clauses by

type-case: lower-case for universally quantified variables, $x, y, z, ...$, upper-case for existentially quantified variables $P, Q, R, X, Y, Z, ...$

Resolution. Resolution [19] is a deductive inference rule: given a conjunction of two clauses, $\neg\phi \vee \psi \wedge \phi \vee \chi$, eliminate the pair $\{\neg\phi, \phi\}$ found in contradiction and *derive* a new clause $\psi \vee \chi$. $\{\neg\phi, \phi\}$ form a *complementary pair*. If C_1, C_2 are clauses from which C_3 is derived by resolution, C_3 is the *resolvent* of C_1, C_2; and $C_1 \cup C_2 \vdash_R C_3$ is the *Resolution-derivation* of C_3 from C_1, C_2. Let L be a literal. A substitution, ϑ, of variables in L is a set $\{v_1/t_1, ..., v_n/t_n\}$ denoting the simultaneous substitution of each variable v_i in L with a term t_i. $L\vartheta$ denotes the application of ϑ to L. If ϕ, ψ are two literals and ϑ is a substitution of variables in both so that $\phi\vartheta = \psi\vartheta$, ϑ is a *unifier* for ϕ, ψ. Let $\neg\phi \vee \psi \wedge \xi \vee \chi$ be a conjunction of clauses. If there exists a unifier ϑ such that $\phi\vartheta = \xi\vartheta$ then $\{\neg\phi\vartheta, \xi\vartheta\}$ form a complementary pair and can be eliminated by Resolution.

SLD-Resolution. SLD-Resolution is a variant of Resolution restricted to Horn clauses. Let $\Pi, \leftarrow G$ be a definite program and a Horn goal. $\Pi \cup \{\leftarrow G\} \vdash_{SLD} \square$, the derivation of the empty clause from $\Pi, \leftarrow G$ by SLD-Resolution is an *SLD-Refutation* of $\leftarrow G$ by Π. Let A, B be two sets of formulae; $A \models B$, read "A entails B", if and only if B is true when A is true, under the standard rules of logic. If $A \models \square$, A is *unsatisfiable*, meaning there is no substitution of the variables in A that can make A true. SLD-Resolution is sound: if there exists ϑ s.t. $\Pi\vartheta \cup \{\leftarrow G\vartheta\} \vdash_{SLD} \square$ then $\Pi \wedge \neg G \models \square$. SLD-Resolution is refutation-complete: if $\Pi \wedge \neg G \models \square$ then there exists ϑ s.t. $\Pi\vartheta \cup \{\leftarrow G\} \vdash_{SLD} \square$. Proofs of the soundness and completeness of SLD-Resolution are detailed in [10] and [15].

3.2 Second-Order SLD-Resolution

Definition 1 (MIL Problem). *Let e^+ be an atom that we call a positive example, and $\mathcal{B} = \{B_{FO}, B_{SO}\}$ be a higher-order definite program where B_{FO} is a set of definite clauses and B_{SO} is a set of Second-Order datalog clauses chosen s.t. $\mathcal{B} \models e^+$ (*Consistency Assumption*) The* MIL Problem *is to find substitutions ϑ, Θ of the variables in $\mathcal{B}$ s.t. $H = B_{SO}\Theta$ and $B_{FO}\vartheta \wedge H \models e^+$ (*Target Theory*). We call $\{e^+, \mathcal{B}\}$ the* elements of the MIL problem *and H a* solution to the MIL problem*. We call H the learned, or induced,* hypothesis*. Note that H is a set of First-Order definite clauses.*

Theorem 1 (Inductive Soundness and Completeness of MIL). *Let $P_{MIL} = \{e^+, \mathcal{B}\}$. If, and only if, the Consistency Assumption holds, then a solution to P_{MIL} will be found by SLD-Refutation of e^+ by $\mathcal{B}$.*

Proof. By the soundness and refutation completeness of SLD-Resolution:
$\exists\vartheta, \Theta : B_{FO}\vartheta \wedge B_{SO}\Theta \models e^+$ (Target Theory)
$\Leftrightarrow \exists\vartheta, \Theta : B_{FO}\vartheta \wedge B_{SO}\Theta \wedge \neg e^+ \models \square$
$\Leftrightarrow \exists\vartheta, \Theta : B_{FO}\vartheta \cup B_{SO}\Theta \cup \{\leftarrow e^+\} \vdash_{SLD} \square$
$\Leftrightarrow \mathcal{B} \models e^+$ (Consistency Assumption).

Remark 1. Theorem 1 states that, given a higher-order background theory $\mathcal{B}$ and a single atomic example e^+ that is a logical consequence of $\mathcal{B}$, a logic program H can always be learned by MIL that entails e^+ with respect to the first-order component of $\mathcal{B}$, i.e. B_{FO}. Theorem 1 makes no assumption about the extent to which H generalises to unseen examples, including negative examples. Thus, MIL is *inductively* sound and complete for one-shot learning.

4 Implementation

4.1 Vanilla-TPC

Vanilla. The results in the previous section inform our implementation of a new learning and reasoning engine for MIL that we call *Vanilla* [1], because it has the same structure as a "vanilla" Prolog meta-interpreter [25], but with added arguments for a) book-keeping (to collect the substitutions of existentially quantified variables in Second-Order clauses), b) filtering (to determine the symbols allowed in the heads of clauses in the learned program, possibly including invented predicate symbols) and, c), depth-limiting (to avoid infinite unifications between Second-Order definite clauses). Vanilla can be run in two modes. One mode uses Tabling (a.k.a. SLG-Resolution) [24] to control recursion, thus allowing left-recursive programs to be learned without going into infinite recursion. This mode also includes each new clause of a hypothesis to the background theory so that it can be resolved with itself, or other clauses in the hypothesis so-far. The second mode eschews tabling and instead controls recursion by a) constraints on the substitutions of existentially quantified variables in Second-Order clauses, to avoid construction of left-recursive clauses, and b) an upper limit on the number of resolution steps taken during an SLD-Refutation. Vanilla is developed in SWI-Prolog [26].

Vanilla-TPC. We use Vanilla to re-implement the MIL system Top Program Construction [16] and use this new version in comparisons with other systems in Section 5. We refer to this new system as Vanilla-TPC.

4.2 Prolog^2

Prolog^2 is a program for compiling and querying second order logic programs. This is achieved through implementing Second Order SLD-Resolution in the programming language Rust. The implementation is loosely inspired by the Warren Abstract Machine(WAM) [6], borrowing concepts such as heap term representation, argument registers, and a goal environment stack

Term Representation When compiling programs or queries the terms must be translated from a text form into a data structure. This is stored in a heap

0	Func	3
1	Con	p
2	Ref	2
3	Ref	3

(a)

0	Func	3
1	Con	p
2	Arg	0
3	Arg	1

(b)

0	STR	1
1	p/2	
2	Ref	2
3	Ref	3

(c)

Fig. 1: Heap representation of the term $p(x, y)$ as in a goal(1a), in a clause (1b), and as a WAM term(1c). Each row represents 2 consecutive memory addresses. Func : n declares that the next n terms form a functor structure including (function or predicate). STR : n declares a structure starting at address n

structure, similar to the WAM, where each cell on the heap contains a tag and value.

Figure 1 Shows the Heap cells which represent the term $P(x, y)$

One key difference from the WAM approach is to consider the predicate or function symbol as a term, which is part of a tuple of terms allowing variables to take the place of any functor. With this representation unification of second order terms becomes as straightforward as unifying tuples of terms in first order logic. This approach also differs from the WAM by storing clauses as compiled terms rather than as abstract machine instruction, necessitating the introduction of the Argument tag (ARG) to differentiate between query and clause variables.

Meta-Clauses With Second Order Resolution we can see two types of clauses, first order and second order clauses. If we want to learn new clauses from Second Order clauses then we need both a substitution and a meta-substitution, the first used to create new goals, and the latter used to build new clauses. But we may not want to learn from all Second Order clauses such as a map clause which contains a predicate as an argument, but is not meant to provide a template for new clauses. To distinguish the two Prolog[2] uses the concept of meta-clauses, which would normally be called meta-rules in standard MIL, denoted by a set of universally quantified variables at the end of the clause, which may be empty.

After unifying a goal with the head of a clause, two values are produced, a set of bindings and a set of argument registers assigned heap addresses. To build a new goal or clause we copy the literals and replace the argument cells with the value at the address in the respective argument register, or introduce a new unbound ref cell if the register is not set. To allow newly learnt clauses to contain variables we use the concept of meta-variables, with an Arg$\forall$ tag. Meta-variables are substituted when building new goals, but replaced by normal variables when building new clauses. In meta-clauses, we must then single out the universally quantified variables by providing these in a set at the end of the clause, all other symbols are either existentially quantified variables, or constants. The Prolog[2]

[1] https://github.com/stassa/vanilla

code shown in Formula 1 represents the higher order clause in Formula 2

$$P(X,Y) : -Q(X,Y)\{X,Y\}. \tag{1}$$

$$\exists PQ \forall xy P(x,y) \leftarrow Q(x,y) \tag{2}$$

The Clause Table Prolog^2 stores all clauses within a clause table. Each clause in the clause table holds 2 key parts, the literals, stored as an array of heap addresses, and the clause type. The clause type can take 1 of 4 values, clause, body, meta, or hypothesis. The clause and body variants are both compiled before the execution of a query, may be either second order or first order clauses, and do not generate more clauses when matched with. The body clauses uniquely allow for goals with variable predicate symbols to match. The meta clauses as previously discussed are used to generate new clauses. finally, the hypothesis clauses which act as body clauses, are generated by matching with meta clauses. With these distinctions between clauses, Prolog^2 can handle the fetching of clauses when calling a goal, with 3 cases to consider, a goal with a variable predicate, a known constant predicate, or an unknown constant predicate.

Resolution Resolution in Prolog^2 is handled by a 'Proof Stack' which stores the environment for each goal, with a pointer variable to the current environment. When an environment is created the only thing stored is a pointer to the goal atom on the heap. When the goal is tried, if a match is made to the head of a clause, 4 things are added. Firstly the list of clause indexes left to be matched against for that goal. Secondly, the bindings created by matching to a clause. Thirdly whether or not a new clause was introduced when matching, and if this new clause has an invented predicate. Finally, the environment stores the number of child goals that are created. These child goal environments are inserted at the index after the current pointer in the proof stack. With this information, if any goal fails, the effects of attempting a goal can be undone whilst backtracking. At each successful step, the solver increments the pointer, once the pointer exceeds the length of the proof stack then a solution is found, if whilst backtracking the pointer would be decremented before the first goal then a query has resolved to false. This approach is an adaptation of the one described for the WAM[6]

Advantages By working directly with higher order terms, this implementation removes the need for a layer of meta-interpretation written on top of Prolog which can provide performance advantages. Additionally, this implementation benefits from simpler Meta-rule representation, allowing higher order predicate symbols to be recognised as variables if they start with a capital letter. With this, more complex or interesting meta-clauses can easily be included in the background knowledge, such as meta-clauses with both first order and higher order predicates, and predicate symbols in the arguments of literals. This can be extended to meta-clauses of higher order clauses allowing for the learning of

higher-order programs, which was once considered an extension to MIL [2], but within this framework, this is a natural result of Second-Order SLD-Resolution.

4.3 Prolog^2-TPC

This implementation does not differ fundamentally from Vanilla-TPC, except that each positive example is learned from in parallel, and each sub-hypothesis is tested against negative examples in parallel. When generalising the Top Program each example is taken as an independent goal to begin a query of the second order logic program. This query then produces several hypotheses found whilst solving for the goal. A set of these hypotheses is collected from each of the instances of Second Order Resolution. This set of sub-hypotheses can then be specialised in parallel to produce the Top Program. By testing each Hypothesis in conjunction with the background knowledge against each negative example with 1st order resolution. Together with the other efficiency gains from Prolog^2, we expect this to provide a significant speed up to the learning.

5 Experiments

By removing the layer of meta-interpretation implemented on top of Prolog for Meta-Interpretive learning and instead implementing native Second Order Resolution we expect to see some improvement in efficiency. More significantly, by parallelising top program construction, we expect to see significant reductions in the time taken to solve learning problems with a large number of examples.

5.1 Materials and Methods

To evaluate *Vanilla-TPC* and *Prolog^2-TPC* we use the 'Grid World' and 'coloured graph' datasets, first used to test Louise[18]. We trained both systems with train test splits from {0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9} with 0.1 having 10% of examples in the training split and 0.9 having 90%. For each split size, the experiment was run 10 times, randomising the split contents each time. For each experiment, we measure the time taken to learn a hypothesis, with a timeout set for 300 seconds, and the accuracy of the hypothesis against the remaining examples. If the experiment reaches the timeout, the accuracy was set to 0. [2]

Grid World. The grid world dataset is a challenge to learn a move/2 predicate, each example in the dataset is an example of a move from one position to another within a 4x4 grid world. The background knowledge includes 16 basic moves, with single steps in one of 4 directions and 12 double steps such as move right

[2] Experiment code and datasets:
`https://github.com/JamesTrewern/louise-Testing/tree/Second-Order-Resolution-Tests`
`https://github.com/JamesTrewern/prolog2/tree/RunTests`

```
double_move(M,Ss,Gs):- move(M), M(Ss,Ss_1), M(Ss_1,Gs).
triple_move(M,Ss,Gs):-move(M), M(Ss,Ss_1), double_move(M,Ss_1,Gs).
quadruple_move(M,Ss,Gs):- move(M), double_move(M,Ss,Ss_1), double_move(M,Ss_1,Gs).
```

Fig. 2: Higher order move predicates

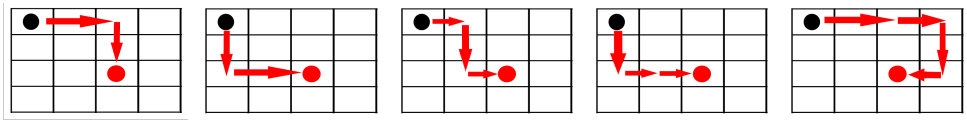


Fig. 3: Example of possible steps taken to go from 0,0 to 2,2 in 4x4 grid world

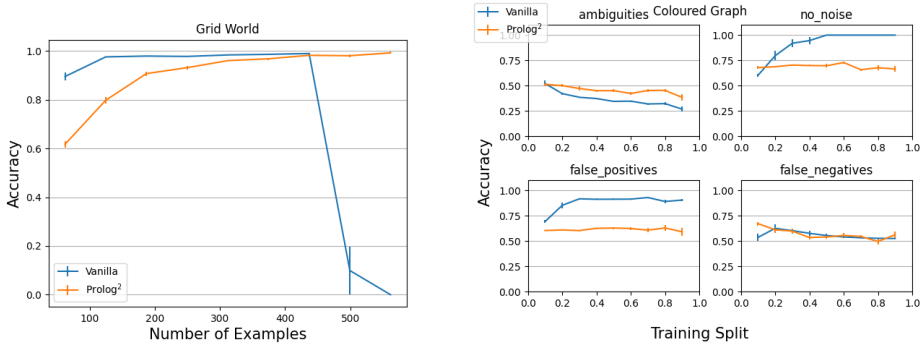
twice, or move right then up. Additionally a set of higher-order moves, listed in Figure 2, are included. Given how many methods there are to move from one location to another this problem, by design, is combinatorially complex, as illustrated by Figure 3, due to the large size of the hypothesis space, which increases exponentially with each example added to the set. This property makes the grid world experiment effective for measuring the efficiency of a learning system, as the cost to search for a hypothesis is so large.

Coloured Graph. The coloured graph dataset describes two coloured graphs with the objective of learning the connected/2 predicate. This dataset includes the 4 variants: no noise, false positives, false negatives, and ambiguities. Each of these subsets describes the problem with differing types of noise in the examples, where ambiguities combines both false positives and negatives. Whilst this dataset is less expensive to learn from than the grid world example, it is effective at measuring the noise tolerance of a learner

5.2 Results

Accuracy As shown in figures 4a and 4b, both implementations of TPC achieved similar accuracy though Prolog^2 under performed in accuracy for the coloured graph experiments with no noise and false positives. The major exception is in the final two splits when learning from 80% and 90% of the examples. Vanilla-TPC began to hit the 300-second threshold for the experiment with these sample sizes, leading to an accuracy of 0 in the very last split as there was no learnt hypothesis.

Time Figures 5a and 5b, as expected, demonstrate that Prolog^2-TPC handled the largest split sizes significantly quicker than Vanilla-TPC. This was likely a result of the parallel nature. More surprisingly, even when handling the smallest split, there is still a large difference in the time taken to build the Top Program, which shows that the native implementation of second order resolution can achieve some significant gains in efficiency.



(a) (b)

Fig. 4: Average accuracy per number of examples for grid world (4a), and per training size ratio for coloured graph (4b). With vertical lines for standard error

6 Conclusions And Future Work

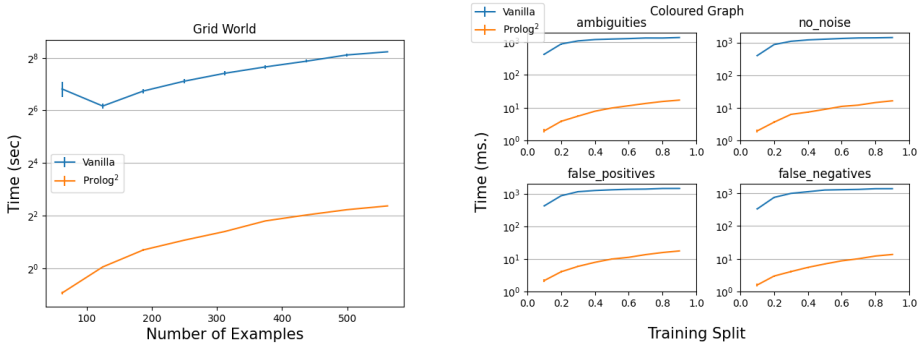
6.1 Conclusions

Our work shows that considering MIL as a form of Second Order Resolution proves soundness and completeness and provides a useful framework for the understanding and implementation of MIL systems such as Vanilla and Prolog^2. We have also demonstrated that Prolog^2-TPC can learn from large numbers of examples within seconds where other MIL systems would take minutes, due to the parallel nature of its execution, and the lack of meta-interpretation needed to achieve Second Order Resolution.

6.2 Future Work

First Order Predicates in Meta-Rules. easy inclusion of first order predicate symbols in the Meta-Rules is an interesting result of considering MIL as Second Order Resolution. Similarly to the concept of interpreted background knowledge[2], the separation of Meta-Rules and background knowledge is not required when considering MIL as Second Order Resolution. This concept could potentially allow domain-specific experts help to inform learning. If applied properly this could bolster MIL's advantages in the areas of human-like computing, and understandable AI.

Alternative Approaches to Resolution. The goal of Prolog^2 and Prolog^2-TPC is to create an efficient MIL engine that could learn from large datasets in reasonable amounts of time. One approach to further increase the efficiency would be to try different approaches to Resolution. One approach is SLG-Resolution also called tabling[1, 20], which can prevent left recursion loops by suspending certain



(a) (b)

Fig. 5: Average time taken to learn hypothesis per number of examples for grid world (4a), and per training size ratio for coloured graph (4b). With vertical lines for standard error

goals, and increase efficiency by memoizing or tabling the results of goals. Another alternative approach to resolution is Multi-SLD-Resolution [21, 22], where multiple binding environments exist concurrently, which could allow for Prolog^2 to utilise parallelism, even when learning from small numbers of examples. There are many approaches to Resolution with different advantages and disadvantages, though these have only been shown for First Order Resolution, so further work is needed to evaluate these approaches for Second Order Resolution, and expand the efficacy of this framework.

Acknowledgments. J. Trewern's PhD is supported by the Surrey Institute for People-Centred AI. The authors acknowledge the EPSRC grant on human-machine learning of ambiguities. The third author acknowledges the EPSRC Network Plus grant on Human-Like Computing (HLC).

References

1. Chen, W., Warren, D.S.: Tabled evaluation with delaying for general logic programs. J. ACM **43**(1), 20–74 (jan 1996). https://doi.org/10.1145/227595.227597, `https://doi.org/10.1145/227595.227597`
2. Cropper, A., Muggleton, S.: Learning higher-order logic programs through abstraction and invention. In: Proceedings of the 25th International Joint Conference Artificial Intelligence (IJCAI 2016). pp. 1418–1424. IJCAI (2016), `http://www.doc.ic.ac.uk/\~shm/Papers/metafunc.pdf`
3. Cropper, A.: Playgol: learning programs through play. CoRR **abs/1904.08993** (2019), `http://arxiv.org/abs/1904.08993`
4. Cropper, A., Muggleton, S.H.: Learning efficient logical robot strategies involving composable objects. In: Yang, Q., Wooldridge, M.J. (eds.) Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015. pp. 3423–3429. AAAI Press (2015), `http://ijcai.org/Abstract/15/482`

5. Flener, P., Yilmaz, S.: Inductive synthesis of recursive logic programs: achievements and prospects. The Journal of Logic Programming **41**(2), 141 – 195 (1999). https://doi.org/https://doi.org/10.1016/S0743-1066(99)00028-X, http://www.sciencedirect.com/science/article/pii/S074310669900028X
6. Hassan, A.K.: Warren's abstract machine: A tutorial reconstruction (1991)
7. Hocquette, C., Muggleton, S.: How much can experimental cost be reduced in active learning of agent strategies? In: Riguzzi, F., Bellodi, E., Zese, R. (eds.) Inductive Logic Programming. pp. 38–53. Springer International Publishing, Cham (2018)
8. Kowalski, R.A.: Predicate logic as programming language. In: IFIP Congress (1974), https://api.semanticscholar.org/CorpusID:10850205
9. Lin, D., Dechter, E., Ellis, K., Tenenbaum, J., Muggleton, S., Dwight, M.: Bias reformulation for one-shot function induction. In: Proceedings of the 23rd European Conference on Artificial Intelligence. pp. 525–530 (2014). https://doi.org/10.3233/978-1-61499-419-0-525
10. Lloyd, J.W.: Foundations of Logic Programming. Symbolic Computation, Springer Berlin Heidelberg, Berlin, Heidelberg (1987). https://doi.org/10.1007/978-3-642-83189-8
11. Muggleton, S., Dai, W.Z., Sammut, C., Tamaddoni-Nezhad, A., Wen, J., Zhou, Z.H.: Meta-interpretive learning from noisy images. Machine Learning **107**(7), 1097–1118 (Jul 2018). https://doi.org/10.1007/s10994-018-5710-8, https://doi.org/10.1007/s10994-018-5710-8
12. Muggleton, S., Lin, D.: Meta-Interpretive Learning of Higher-Order Dyadic Datalog : Predicate Invention Revisited. Machine Learning **100**(1), 49–73 (2015)
13. Muggleton, S., de Raedt, L.: Inductive Logic Programming: Theory and methods. The Journal of Logic Programming **19-20**(SUPPL. 1), 629–679 (1994), https://doi.org/10.1016/0743-1066(94)90035-3
14. Muggleton, S.H., Lin, D., Pahlavi, N., Tamaddoni-Nezhad, A.: Meta-interpretive learning: Application to grammatical inference. Machine Learning **94**(1), 25–49 (2014), https://doi.org/10.1007/s10994-013-5358-3
15. Nienhuys-Cheng, S.H., de Wolf, R.: Foundations of Inductive Logic programming. Springer-Verlag, Berlin (1997)
16. Patsantzis, S., Muggleton, S.H.: Louise system (2019), https://github.com/stassa/louise
17. Patsantzis, S., Muggleton, S.H.: Meta-interpretive learning as metarule specialisation. CoRR **abs/2106.07464** (2021), https://arxiv.org/abs/2106.07464
18. Patsantzis, S., Muggleton, S.H.: Top program construction and reduction for polynomial time meta-interpretive learning. Machine Learning **110**(4), 755–778 (2021)
19. Robinson, J.A.: A machine-oriented logic based on the resolution principle. J. ACM **12**(1), 23–41 (Jan 1965), https://doi.org/10.1145/321250.321253
20. Sagonas, K., Swift, T.: An abstract machine for tabled execution of fixed-order stratified logic programs. ACM Transactions on Programming Languages and Systems (TOPLAS) **20**(3), 586–634 (1998)
21. Smith, D.A.: Multilog and data or-parallelism. The Journal of Logic Programming **29**(1), 195–244 (1996). https://doi.org/https://doi.org/10.1016/S0743-1066(96)00067-2, https://www.sciencedirect.com/science/article/pii/S0743106696000672, high-Performance Implementations of Logic Programming Systems
22. Smith, D.A., Hickey, T.J.: Multi-sld resolution. In: Pfenning, F. (ed.) Logic Programming and Automated Reasoning. pp. 260–274. Springer Berlin Heidelberg, Berlin, Heidelberg (1994)

23. Stahl, I.: Predicate invention in ILP — an overview. In: Brazdil, P.B. (ed.) Machine Learning: ECML-93. pp. 311–322. Springer Berlin Heidelberg, Berlin, Heidelberg (1993)
24. Tamaki, H., Sato, T.: Old resolution with tabulation. In: Shapiro, E. (ed.) Third International Conference on Logic Programming. pp. 84–98. Springer Berlin Heidelberg, Berlin, Heidelberg (1986)
25. Triska, M.: The power of Prolog. https://www.metalevel.at/prolog (2005), `https://www.metalevel.at/prolog`
26. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming **12**(1-2), 67–96 (2012)