Machine learning manuscript No.
(will be inserted by the editor)

Learning higher-order logic programs

Andrew Cropper · Rolf Morel · Stephen Muggleton

the date of receipt and acceptance should be inserted later

Abstract A key feature of inductive logic programming (ILP) is its ability to learn first-order programs, which are intrinsically more expressive than propositional programs. In this paper, we introduce techniques to learn higher-order programs. Specifically, we extend meta-interpretive learning (MIL) to support learning higher-order programs by allowing for *higher-order definitions* to be used as background knowledge. Our theoretical results show that learning higher-order programs, rather than first-order programs, can reduce the textual complexity required to express programs which in turn reduces the size of the hypothesis space and sample complexity. We implement our idea in two new MIL systems: the Prolog system Metagol_{ho} and the ASP system HEXMIL_{ho}. Both systems support learning higher-order programs and higher-order predicate invention, such as inventing functions for `map/3` and conditions for `filter/3`. We conduct experiments on four domains (robot strategies, chess playing, list transformations, and string decryption) that compare learning first-order and higher-order programs. Our experimental results support our theoretical claims and show that, compared to learning first-order programs, learning higher-order programs can significantly improve predictive accuracies and reduce learning times.

1 Introduction

Suppose you have intercepted encrypted messages and you want to learn a general decryption program from them. Figure 1 shows such a scenario with three example encrypted/decrypted strings. In this scenario the underlying encryption algorithm is a simple

A. Cropper
University of Oxford, UK
E-mail: andrew.cropper@cs.ox.ac.uk

R. Morel
University of Oxford, UK
E-mail: rolf.morel@cs.ox.ac.uk

S. H. Muggleton
Imperial College London, UK
E-mail: s.muggleton@imperial.ac.uk

Caesar cipher with a shift of +1. Given these examples, most inductive logic programming (ILP) approaches, such as meta-interpretive learning (MIL) [34,35], would learn a recursive first-order program, such as the one shown in Figure 2a. Although correct, this first-order program is overly complex in that most of the program is concerned with manipulating the input and output, such as getting the head and tail elements. In this paper, we introduce techniques to learn higher-order programs that abstract away this boilerplate code. Specifically, we extend MIL to support learning higher-order programs that use higher-order constructs such as `map/3`, `until/4`, and `ifthenelse/5`. Using this new approach, we can learn an equivalent[1] yet smaller decryption program, such as the one shown in Figure 2b, which uses `map/3` to abstract away the recursion and list manipulation.

Encrypted	**Decrypted**
`joevdujwf`	`inductive`
`mphjd`	`logic`
`qsphsbnnjoh`	`programming`

Fig. 1: Example encrypted and decrypted messages.

```
decrypt(A,B):-
    empty(A),
    empty(B).
decrypt(A,B):-
    head(A,C),
    char_to_int(C,D),
    prec(D,E),
    int_to_char(E,F),
    head(B,F),
    tail(A,G),
    tail(B,H),
    decrypt(G,H).
```

(a)

```
decrypt(A,B):-
    map(A,B,decrypt1).
decrypt1(A,B):-
    char_to_int(A,C),
    prec(C,D),
    int_to_char(D,B).
```

(b)

Fig. 2: Decryption programs. Figure (a) shows a first-order program. Figure (b) shows a higher-order program, where `decrypt1/2` is an invented predicate symbol. The predicate `prec/2` represents *preceding/2*, i.e. the inverse of *successor/2*. The programs are success set equivalent when restricted to the target predicate `decrypt/2` but the higher-order program is much smaller and requires half the number of literals (6 vs 12).

We claim that, compared to learning first-order programs, learning higher-order programs can improve learning performance. We support our claim by showing that learning higher-order programs can reduce the textual complexity required to express programs which in turn reduces the size of the hypothesis space and sample complexity.

[1] Success set equivalent when restricted to the target predicate `decrypt/2`.

We implement our idea in Metagol$_{ho}$, which extends Metagol [9], a MIL implementation based on a Prolog meta-interpreter. Metagol$_{ho}$ extends Metagol to support *interpreted* BK (IBK). In this approach, meta-interpretation drives both the search for a hypothesis and predicate invention, allowing for higher-order arguments to be invented, such as the predicate `decrypt1/2` in Figure 2b. The key novelty of Metagol$_{ho}$ is the combination of *abstraction* (learning higher-order programs) and *invention* (predicate invention), i.e. inventions inside of abstractions. Metagol$_{ho}$ supports the invention of conditions and functions to an arbitrary depth, which goes beyond anything in the literature. We also introduce HEXMIL$_{ho}$, which likewise extends HEXMIL [23], an answer set programming (ASP) MIL implementation, to support learning higher-order programs. As far as we are aware, HEXMIL$_{ho}$ is the first ASP-based ILP system that has been demonstrated capable of learning higher-order programs.

We further support our claim that learning higher-order programs can improve learning performance by conducting experiments in four domains: robot strategies, chess playing, list transformations, and string decryption. The experiments compare the predictive accuracies and learning times when learning first and higher-order programs. In all cases learning higher-order programs leads to substantial increases in predictive accuracies and lower learning times in agreement with our theoretical results.

Our main contributions are:

- We extend the MIL framework to support learning higher-order programs by extending it to support higher-order definitions (Section 3.2).
- We show that the new higher-order approach can reduce the textual complexity of programs which in turn reduces the size of the hypothesis space and also sample complexity (Section 3.3).
- We introduce Metagol$_{ho}$ and HEXMIL$_{ho}$ which extend Metagol and HEXMIL respectively. Both systems support learning higher-order programs with higher-order predicate invention (Section 4).
- We show that the ASP-based HEXMIL and HEXMIL$_{ho}$ have an additional factor determining the size of their search space, namely the number of constants (Section 4.5).
- We conduct experiments in four domains which show that, compared to learning first-order programs, learning higher-order programs can substantially improve predictive accuracies and reduce learning times (Section 5).

2 Related work

2.1 Program induction

Program synthesis is the automatic generation of a computer program from a specification. Deductive approaches [27] *deduce* a program from a full specification which precisely states the requirements and behaviour of the desired program. By contrast, program induction approaches *induce* (learn) a program from an incomplete specification, usually input/output examples. Many program induction approaches learn specific classes of programs, such as string transformations [20]. By contrast, MIL is general-purpose, shown capable of grammar induction [34], learning robot strategies [7], and learning efficient algorithms [10]. In addition, MIL supports *predicate invention*, which has been repeatedly stated as an important challenge in ILP [32, 42, 33]. The idea behind predicate invention is for an ILP system to introduce new predicate symbols to improve

learning performance. In program induction, predicate invention can be seen as inventing auxiliary functions/predicates, as one does when manually writing a program, for example to reduce code duplication or to improve the readability of a program.

2.2 Inductive functional programming

Functional program induction approaches often support learning higher-order programs. MagicHaskeller [24] is a general-purpose system which learns Haskell functions by selecting and instantiating higher-order functions from a pre-defined vocabulary. Igor2 [25] also learns recursive Haskell programs and supports auxiliary function invention but is restricted in that it requires the first k examples of a target theory to generalise over a whole class. The *L2* system [16] synthesises recursive functional algorithms. The *MYTH* [36] and *MYTH2* [18] systems use type systems to synthesise programs. Frankle et al. [18] show how example-based specifications can be turned into type specifications. In this work we go beyond these approaches by (1) learning higher-order programs with invented predicates, (2) giving theoretical justifications and conditions for when learning higher-order programs can improve learning performance (Section 3.3), and (3) experimentally demonstrating that learning higher-order programs can improve learning performance.

2.3 Inductive logic programming

ILP systems, including the popular systems FOIL [37], Progol [31], ALEPH [41], and TILDE [1], usually learn first-order programs. Given appropriate mode declarations [31] for higher-order predicates such as `map/3`, Progol and Aleph could learn higher-order programs such as `f(A,B):-map(A,B,f1)`. However, because Progol and Aleph do not support predicate invention they would be unable to invent the predicate `f1/2` in the above example. Similarly, existing MIL implementations, such as Metagol, could learn a similar program to the one above when `map/3` is provided as background knowledge. However, even though Metagol supports predicate invention, it is unable to invent the predicate `f1/2` in the example above because Metagol deductively proves BK by delegating the proofs to Prolog. To overcome this limitation we introduce the notion of interpreted BK (IBK), where `map/3` can be defined as IBK. The new MIL system Metagol$_{ho}$ proves IBK through meta-interpretation, which allows for predicate arguments such as `f1/2` to be invented.

2.4 Meta-interpretive learning

MIL was originally based on a Prolog meta-interpreter, although the MIL problem has also been encoded as an ASP problem [23]. The key difference between a MIL learner and a standard Prolog meta-interpreter is that whereas a standard Prolog meta-interpreter attempts to prove a goal by repeatedly fetching first-order clauses whose heads unify with a given goal, a MIL learner additionally attempts to prove a goal by fetching higher-order existentially quantified formulas, called *metarules*, supplied as BK, whose heads unify with the goal. The resulting predicate substitutions are saved and can be reused later in the proof. Following the proof of a set of goals, a logic program is induced by

projecting the predicate substitutions onto their corresponding metarules. A key feature of MIL is the support for predicate invention. MIL uses predicate invention for automatic problem decomposition. As we will demonstrate, the combination of predicate invention and abstraction leads to compact representations of complex programs.

Cropper and Muggleton [8] introduced the idea of using MIL to learn higher-order programs by using IBK. This paper is an extended version of that paper. In addition, we go beyond that work in several ways. First, we generalise their preliminary theoretical results, principally in Section 3.3. We also provide more explanation as to why abstracted MIL can improve learning performance compared to unabstracted MIL (end of Section 3.3). Second, we introduce the HEXMIL$_{ho}$ system, which, as mentioned, extends HEXMIL to support learning higher-order programs with higher-order predicate invention. Our motivation for this extension is to show the generality of our work, i.e. to demonstrate that it is not specific to Metagol and Prolog. We also study the computational complexity of both Metagol$_{ho}$ and HEXMIL$_{ho}$. We show that the ASP approach is highly sensitive to the number of constant symbols, which leads to scalability issues. Furthermore, we corroborate the experimental results of Cropper and Muggleton by repeating the robot waiter, chess, and list transformation experiments with Metagol$_{ho}$. We provide additional experimental evidence by repeating the experiments with HEXMIL$_{ho}$. Finally, we add further evidence by conducting a new experiment on the string decryption problem mentioned in the introduction.

2.5 Higher-order logic

McCarthy [28] advocated using higher-order logic to represent knowledge. Similarly, Muggleton et al. [33] argued that using higher-order representations in ILP provides more flexible ways of representing BK. Lloyd [26] used higher-order logic in the learning process but the approach focused on learning functional programs and did not support predicate invention. Early work in ILP [17, 38, 14] used higher-order formulae to specify the overall form of programs to be learned, similar to how MIL uses metarules. However, these works did not consider learning higher-order programs. By contrast, we use higher-order logic as a learning representation and to represent learned hypotheses. Feng and Muggleton [15] investigated inductive generalisation in higher-order logic using a restricted form of lambda calculus. However, their approach does not support first-order nor higher-order predicate invention. By contrast, we introduce higher-order definitions which allow for invented predicate symbols to be used as arguments in literals.

2.6 Abstraction and invention

Predicate invention has been repeatedly stated as an important challenge in ILP [32, 42, 33]. Popular ILP systems, such as FOIL, Progol, and ALEPH, do not support predicate invention, nor do most program induction systems. Meta-level abduction [22] uses abduction and meta-level reasoning to invent predicates that represent propositions. By contrast, MIL uses abduction to invent predicates that represent relations, i.e. relations that are not in the initial BK nor in the examples. For instance, MIL was shown [35] able to invent a predicate corresponding to the *parent/2* relation when learning a *grandparent/2* relation. In this paper we extend MIL and the associated Metagol implementation to support higher-order predicate invention for use in higher-order constructs, such as

map/3, reduce/3, and fold/4. This approach supports a form of abstraction which goes beyond typical first-order predicate invention [39] in that the use of higher-order definitions combined with meta-interpretation drives both the search for a hypothesis and predicate invention, leading to more accurate and compact programs.

3 Theoretical framework

3.1 Preliminaries

We assume familiarity with logic programming. However, we restate key terminology. Note that we focus on learning function-free logic programs, so we ignore terminology to do with function symbols. We denote the predicate and constant signatures as $\mathscr{P}$ and $\mathscr{C}$ respectively. A variable is first-order if it can be bound to a constant symbol or another first-order variable. A variable is higher-order if it can be bound to a predicate symbol or another higher-order variable. We denote the sets of first-order and higher-order variables as $\mathscr{V}_1$ and $\mathscr{V}_2$ respectively. A term is a variable or a constant symbol. A term is ground if it contains no variables. An atom is a formula $p(t_1, \ldots, t_n)$, where p is a predicate symbol of arity n and each t_i is a term. An atom is ground if all of its terms are ground. A higher-order term is a higher-order variable or a predicate symbol. An atom is higher-order if it has at least one higher-order term. A literal is an atom A (a positive literal) or its negation $\neg A$ (a negative literal). A clause is a disjunction of literals. The variables in a clause are universally quantified. A Horn clause is a clause with at most one positive literal. A definite clause is a Horn clause with exactly one positive literal. A clause is higher-order if it contains at least one higher-order atom. A logic program is a set of Horn clauses. A logic program is higher-order if it contains at least one higher-order Horn clause.

3.2 Abstracted meta-interpretive learning

We extend MIL to the higher-order setting. We first restate metarules [6]:

Definition 1 (Metarule) A metarule is a higher-order formula of the form:

$$\exists \pi \forall \mu \;\; l_0 \leftarrow l_1, \ldots, l_m$$

where each l_i is a literal, $\pi \subseteq \mathscr{V}_1 \cup \mathscr{V}_2$, $\mu \subseteq \mathscr{V}_1 \cup \mathscr{V}_2$, and π and μ are disjoint.

In contrast to a higher-order Horn clause, in which all the variables are all universally quantified, the variables in a metarule can be quantified universally or existentially[2]. When describing metarules, we omit the quantifiers. Instead, we denote existentially quantified higher-order variables as uppercase letters starting from P and universally quantified first-order variables as uppercase letters starting from A. Figure 3 shows example metarules.

To extend MIL to support learning higher-order programs we introduce higher-order definitions:

[2] Existentially quantified first-order variables do not appear in this work, but do in existing work on MIL [11].

Name	Metarule
ident	$P(A,B) \leftarrow Q(A,B)$
precon	$P(A,B) \leftarrow Q(A),R(A,B)$
curry	$P(A,B) \leftarrow Q(A,B,R)$
chain	$P(A,B) \leftarrow Q(A,C),R(C,B)$

Fig. 3: Example metarules. The letters P, Q, and R denote existentially quantified higher-order variables. The letters A, B, and C denote universally quantified first-order variables.

Definition 2 (Higher-order definition) A higher-order definition is a set of higher-order Horn clauses where the head atoms have the same predicate symbol.

Three example higher-order definitions are:

Example 1 ***(Map definition)***

```
map([],[],F) ←
map([A|As],[B|Bs],F) ← F(A,B), map(As,Bs)
```

In Example 1 the symbol F is a universally quantified higher-order variable. The other variables are universally quantified first-order variables.

Example 2 ***(Until definition)***

```
until(A,A,Cond,F) ← Cond(A)
until(A,B,Cond,F) ← not(Cond(A)), F(A,C), until(C,B,Cond,F)
```

Example 3 ***(Fold definition)***

```
fold([],Acc,Acc,F) ←
fold([A|As],Acc1,B,F) ← F(A,Acc1,Acc2), fold(As,Acc2,B,F)
```

We frequently refer to *abstractions*. In computer science code abstraction [4] involves hiding complex code to provide a simpler interface. In this work, we define an *abstraction* as a higher-order Horn clause that contains at least one atom which takes a predicate symbol an argument. In the following abstraction example, the final argument of $map/3$ is ground to the predicate symbol $succ/2$:

Example 4 ***(Abstraction)*** f(A,B) ← map(A,B,succ)

Likewise, in the higher-order decryption program in the introduction (Figure 2b), the final argument of `map/3` is ground to the predicate symbol `decrypt1/2`.

We define the abstracted MIL input, which extends a standard MIL input [6] (and problem) to support higher-order definitions:

Definition 3 (Abstracted MIL input) An abstracted MIL input is a tuple (B,E^+,E^-,M) where:

- $B = B_C \cup B_I$ where B_C is a set of Horn clauses and B_I is (the union of) a set of higher-order definitions
- E^+ and E^- are disjoint sets of ground atoms representing positive and negative examples respectively
- M is a set of metarules.

There is little declarative difference between B_C and B_I. There is, however, a procedural difference between the two. We therefore call B_C *compiled* BK and B_I *interpreted* BK (IBK). The procedural distinction between B_C and B_I is that whereas a clause from B_C is proved deductively (by calling Prolog), a clause from B_I is proved through meta-interpretation, which allows for predicate invention to be combined with abstractions to invent higher-order predicates. The distinction between B_I and M is that the clauses in B_I are all universally quantified, whereas the metarules in M contain existentially quantified variables whose substitutions form the induced program. We discuss these distinctions in more detail in Section 4 when we describe the MIL implementations.

We define the abstracted MIL problem:

Definition 4 (Abstracted MIL problem) Given an abstracted MIL input (B, E^+, E^-, M), the abstracted MIL problem is to return a logic program hypothesis H such that:

- $\forall h \in H, \exists m \in M$ such that $h = m\theta$, where θ is a substitution that grounds all the existentially quantified variables in m
- $H \cup B \models E^+$
- $H \cup B \not\models E^-$

We call H a solution to the MIL problem.

The first condition ensures that a logic program hypothesis is an instance of the given metarules. It is this condition that enforces the strong inductive bias in MIL.

MIL supports inventions:

Definition 5 (Invention) Let (B, E^+, E^-, M) be a MIL input and H be a solution to the MIL problem. Then a predicate symbol p/a is an *invention* if and only if it is in the predicate signature (i.e. the set of all predicate symbols with their associated arities) of H and not in the predicate signature of $B \cup E^+ \cup E^-$.

A MIL learner uses abstractions to generate inventions:

Example 5 ***(Invention)***

```
f(A,B) ← map(A,B,f1)
f1(A,B) ← succ(A,C),succ(C,B)
```

In this program, a MIL learner has invented the predicate `f1/2` for use in a `map/3` definition. Likewise, in the higher-order decryption program in the introduction (Figure 2b), the final argument of `map/3` is ground to the invented predicate symbol `decrypt1/2`.

3.3 Language classes, expressivity, and complexity

We claim that increasing the expressivity of MIL from learning first-order programs to learning higher-order programs can improve learning performance. We support this claim by showing that learning higher-order programs can reduce the size of the hypothesis space which in turn reduces sample complexity and expected error. In MIL the size of the hypothesis space is a function of the number of metarules m and their form, the number of background predicate symbols p, and the maximum program size n (the maximum number of clauses allowed in a program). We restrict metarules by their body size and literal arity:

Definition 6 (Metarule fragment $\mathcal{M}^i_j$) A metarule is in the fragment $\mathcal{M}^i_j$ if it has at most j literals in the body and each literal has arity at most i.

For instance, the *chain* metarule in Figure 3 restricts clauses to be definite with two body literals of arity two, i.e. is in the fragment $\mathcal{M}^2_2$. By restricting the form of metarules we can calculate the size of a MIL hypothesis space. The following result is essentially the same as in [12]. The only difference is that we drop the redundant Big O notation:

Proposition 1 (MIL hypothesis space) *Given p predicate symbols and m metarules in $\mathcal{M}^i_j$, the number of programs expressible with n clauses is at most $(mp^{j+1})^n$.*

Proof The number of clauses which can be constructed from a $\mathcal{M}^i_j$ metarule given p predicate symbols is at most p^{j+1} because for a given metarule there are at most $j+1$ predicate variables with at most p^{j+1} possible substitutions. Therefore the number of clauses that can be formed from m distinct metarules in $\mathcal{M}^i_j$ using p predicate symbols is at most mp^{j+1}. It follows that the number of programs which can be formed from a selection of n such clauses is at most $(mp^{j+1})^n$. □

Proposition 1 shows that the MIL hypothesis space grows exponentially both in the size of the target program and the number of body literals in a clause. For instance, for the $\mathcal{M}^2_2$ fragment, the MIL hypothesis space contains at most $(mp^3)^n$ programs, where m is the number of metarules and n is the number of clauses in the target program.

We update this bound for the abstracted MIL framework:

Proposition 2 (Number of abstracted $\mathcal{M}^i_j$ programs) *Given p predicate symbols and m metarules in $\mathcal{M}^i_j$ with at most k additional existentially quantified higher-order variables, the number of abstracted $\mathcal{M}^i_j$ programs expressible with n clauses is at most $(mp^{j+1+k})^n$.*

Proof As with Proposition 1, the number of clauses which can be constructed from a $\mathcal{M}^i_j$ metarule given p predicate symbols is at most p^{j+1} because for a given metarule there are at most $j+1$ predicate variables with at most p^{j+1} possible substitutions. Given a metarule in $\mathcal{M}^i_j$ with at most k additional existentially quantified higher-order variables there are therefore potentially $j+1+k$ predicate variables with p^{j+1+k} possible substitutions. Therefore the number of clauses expressible with m such metarules is at most mp^{j+1+k}. By the same reasoning as for Proposition 1, it follows that the number of programs which can be formed from a selection of n such clauses is at most $(mp^{j+1+k})^n$. □

We use this result to develop sample complexity [29] results for unabstracted MIL:

Proposition 3 (Sample complexity of unabstracted MIL) *Given p predicate symbols, m metarules in $\mathcal{M}^i_j$, and a maximum program size n_u, unabstracted MIL has sample complexity:*

$$s_u \geq \frac{1}{\epsilon}(n_u \ \ln(m) + (j+1)n_u \ \ln(p) + \ln(\frac{1}{\delta}))$$

Proof According to the Blumer bound, which appears as a reformulation of Lemma 2.1 in [2], the error of consistent hypotheses is bounded by ϵ with probability at least $(1-\delta)$ once $s_u \geq \frac{1}{\epsilon}(\ln(|H|) + \ln(\frac{1}{\delta}))$, where $|H|$ is the size of the hypothesis space. From Proposition 1, $|H| = (mp^{j+1})^{n_u}$ for unabstracted MIL. Substituting and applying logs gives:

$$s_u \geq \frac{1}{\epsilon}(n_u \ \ln(m) + (j+1)n_u \ \ln(p) + \ln(\frac{1}{\delta}))$$

□

We likewise develop sample complexity results for abstracted MIL:

Proposition 4 (Sample complexity of abstracted MIL) *Given p predicate symbols, m metarules in $\mathcal{M}^i_j$ augmented with at most k higher-order variables, and a maximum program size n_a, abstracted MIL has sample complexity:*

$$s_a \geq \frac{1}{\epsilon}(n_a \ \ln(m) + (j+1+k)n_a \ \ln(p) + \ln(\frac{1}{\delta}))$$

Proof Analogous to Proposition 3 using Proposition 2. □

We compare these bounds:

Theorem 1 (Unabstracted and abstracted bounds) *Let m be the number of $\mathcal{M}^i_j$ metarules, n_u and n_a be the minimum numbers of clauses necessary to express a target theory with unabstracted and abstracted MIL respectively, s_u and s_a be the bounds on the number of training examples required to achieve error less than ϵ with probability at least $1-\delta$ with unabstracted and abstracted MIL respectively, and $k \geq 1$ be number of additional higher-order variables used by abstracted MIL. Then $s_u > s_a$ when:*

$$n_u - n_a > \frac{k}{j+1} n_a$$

Proof From Proposition 3 it holds that:

$$s_u \geq \frac{1}{\epsilon}(n_u \ \ln(m) + (j+1)n_u \ \ln(p) + \ln(\frac{1}{\delta}))$$

From Proposition 4 it holds that:

$$s_a \geq \frac{1}{\epsilon}(n_a \ \ln(m) + (j+1+k)n_a \ \ln(p) + \ln\frac{1}{\delta})$$

If we cancel $\frac{1}{\epsilon}$ then $s_u > s_a$ follows from:

$$n_u \ln(m) + (j+1)n_u \ln(p) > n_a \ln(m) + (j+1+k)n_a \ln(p)$$

Because $k \geq 1$, the inequality $s_u > s_a$ holds when:

$$n_u \ln(m) > n_a \ln(m) \quad (1)$$

and:

$$(j+1)n_u \ln(p) > (j+1+k)n_a \ln(p) \quad (2)$$

Because $k \geq 1$ the inequality (2) implies the inequality (1). The inequality (2) holds when $(j+1)n_u > (j+1+k)n_a$. Therefore $s_u > s_a$ follows from $(j+1)n_u > (j+1+k)n_a$. Rearranging terms leads to $s_u > s_a$ when $n_u - n_a > \frac{k}{j+1} n_a$. □

The results from this section motivate the use of abstracted MIL, and help explain the experimental results (Section 5). To illustrate these theoretical results, reconsider the decryption programs shown in Figure 2. Consider representing these programs in $\mathcal{M}^2_2$. Figure 4a shows that the first-order program would require seven clauses. By contrast, Figure 4b shows that the higher-order program requires only three clauses and one extra higher-order variable. Let $m_u = 4$, $p_u = 6$, and $n_u = 7$ be the number of metarules, background relations, and clauses needed to express the first-order program shown in Figure

4a. Plugging these values into the formula in Proposition 1 shows that the hypothesis space of unabstracted MIL contains approximately 10^{21} programs. By contrast, suppose we allow an abstracted MIL learner to additionally use the higher-order definition `map/3` and the corresponding *curry* metarule $P(A,B) \leftarrow Q(A,B,R)$. Therefore $m_a = m_u + 1$, $p_a = p_u + 1$, $n_a = 3$, and $k = 1$, where k is the number of additional higher-order variables used in the curry metarule. Then plugging these values into the formula from Proposition 2 shows that the hypothesis space of abstracted MIL contains approximately 10^{13} programs, which is substantially smaller than the first-order hypothesis space, despite using more metarules and more background relations. The Blumer bound [2] says that given two hypothesis spaces of different sizes, then searching the smaller space will result in less error compared to the larger space, assuming that the target hypothesis is in both spaces. In this example, the target hypothesis, or a hypothesis that is equivalent[3] to the target hypothesis, is in both hypothesis spaces but the abstracted MIL space is smaller. Therefore, our results imply that in this scenario, given a fixed number of examples, abstracted MIL should improve predictive accuracies compared to unabstracted MIL. In Section 5.5 we experimentally explore whether this result holds.

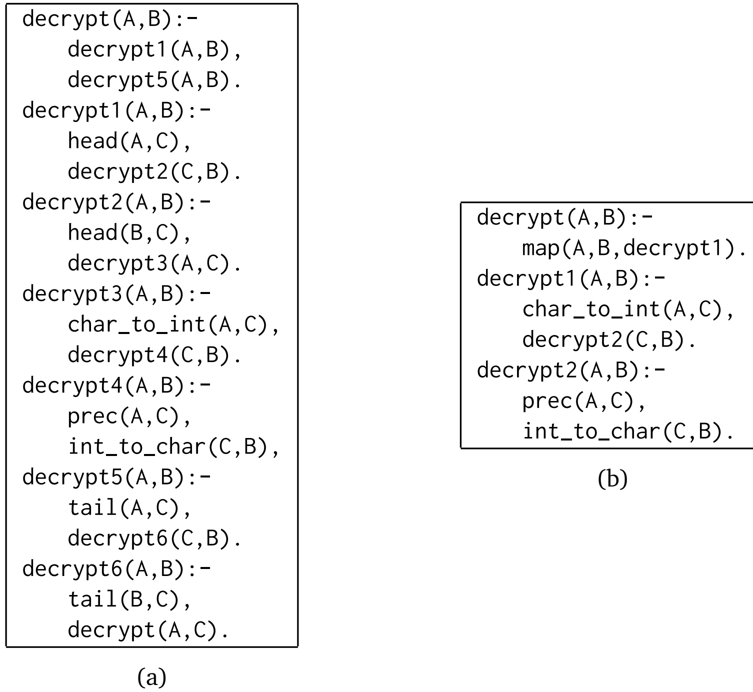


Fig. 4: Decryption programs. Figure (a) shows a first-order program represented in $\mathscr{M}^2_2$. Figure (b) shows a higher-order program represented in $\mathscr{M}^2_2$ with one extra higher-order variable (the third argument of `map/3`).

[3] Success set equivalent when restricted to the target predicate `decrypt/2`. The success set of a logic program P is the set of ground atoms $\{A \in hb(P) | P \cup \{\neg A\}$ has a SLD-refutation$\}$, where $hb(P)$ represents the Herband base of the logic program P. The success set restricted to a specific predicate symbol p is the subset of the success set restricted to atoms of the predicate symbol p.

4 Algorithms

We now introduce Metagol$_{ho}$ and HEXMIL$_{ho}$, both of which implement abstracted MIL and which extend Metagol and HEXMIL respectively. For self-containment, we also describe Metagol and HEXMIL.

4.1 Metagol

Metagol [9] is a MIL learner based on a Prolog meta-interpreter. Figure 5 shows Metagol's learning procedure described using Prolog. Metagol works as follows. Given a set of atoms representing positive examples, Metagol tries to prove each atom in turn. Metagol first tries to deductively prove an atom using compiled BK by delegating the proof to Prolog (`call(Atom)`), where the compiled BK contains standard Prolog definitions. Metagol uses *prim* statements to allow a user to specify what predicates are part of the compiled BK. Prim statements of the form `prim(P/A)`, where `P` is a predicate symbol and `A` is the associated arity, and are similar to determinations used by Aleph [41], except that Metagol only requires prim statements for predicates that may appear in the body. If this deductive step fails, Metagol tries to unify the atom with the head of a metarule (`metarule(Name,Subs,(Atom:-Body))`) and tries to bind the existentially quantified higher-order variables in a metarule to symbols in the predicate signature, where `Subs` contains the substitutions. Metagol saves the resulting substitutions and tries to prove the body of the metarule. After proving all atoms, a Prolog program is formed by projecting the substitutions onto their corresponding metarules. Metagol checks the consistency of the learned program with the negative examples. If the program is inconsistent, then Metagol backtracks to explore different branches of the SLD-tree.

Metagol uses iterative deepening to ensure that the first consistent hypothesis returned has the minimal number of clauses. The search starts at depth 1. At depth d the search returns a consistent hypothesis with at most d clauses if one exists; otherwise it continues to depth $d+1$. At each depth d, Metagol introduces $d-1$ new predicate symbols[4].

4.2 Metagol$_{ho}$

Figure 6 shows the Prolog code for Metagol$_{ho}$. The key difference between Metagol$_{ho}$ and Metagol is the introduction of the second `prove_aux/3` clause in the meta-interpreter, denoted in boldface. This clause allows Metagol$_{ho}$ to prove an atom by fetching a clause from the IBK (such as `map/3`) whose head unifies with a given atom. The distinction between compiled and interpreted BK is that whereas a clause from the compiled BK is proved deductively by calling Prolog, a clause from the IBK is proved through meta-interpretation. Meta-interpretation allows for predicate invention to be driven by the proof of conditions (as in `filter/3`) and functions (as in `map/3`). IBK is different to metarules because the clauses are all universally quantified and, importantly, does not

[4] Metagol forms new predicate symbols by taking the name of the task and adding underscores and numbers. For example, if the task is `f` and the depth is 4 then Metagol will add the predicate symbols `f_3`, `f_2`, and `f_1` to the predicate signature. Note that in this paper we remove the underscore symbols from any learned programs to save space, but the experimental code contains the original underscore symbols.

```
learn(Pos,Neg,Prog):-
    prove(Pos,[],Prog),
    \+ prove(Neg,Prog,Prog).
prove([],Prog,Prog).
prove([Atom|Atoms],Prog1,Prog2):-
    prove_aux(Atom,Prog1,Prog3),
    prove(Atoms,Prog3,Prog2).
prove_aux(Atom,Prog,Prog):-
    prim(Atom),!,
    call(Atom).
prove_aux(Atom,Prog1,Prog2):-
    member(sub(Name,Subs),Prog1),
    metarule(Name,Subs,(Atom:-Body)),
    prove(Body,Prog1,Prog2).
prove_aux(Atom,Prog1,Prog2):-
    metarule(Name,Subs,(Atom:-Body)),
    prove(Body,[sub(Name,Subs)|Prog1],Prog2).
```

Fig. 5: Metagol's learning procedure described using Prolog. Note that this code is the barebones code for Metagol and the actual code differs. The actual code has slightly different syntax and includes more code, such as code to perform the iterative deepening and code to invent new predicate symbols. For instance, in this Figure `prim(Atom)` is not of the form `prim(P/A)`, as described in the text.

require any substitutions. By contrast, metarules contain existentially quantified variables whose substitutions form the hypothesised program. Figure 7 shows examples of the three forms of BK used by Metagol_{ho}.

Metagol_{ho} works in the same way as Metagol except for the use of IBK. Metagol_{ho} first tries to prove an atom deductively using compiled BK by delegating the proof to Prolog (`call(Atom)`), exactly how Metagol works. If this step fails, Metagol_{ho} tries to unify the atom with the head of a clause in the IBK (`ibk((Atom:-Body))`) and tries to prove the body of the matched definition. Metagol does not perform this additional step. Failing this, Metagol_{ho} continues to work in the same way as Metagol. Metagol_{ho} uses negation as failure [5] to negate predicates in the compiled BK. Negation of invented predicates is unsupported and is left for future work[5].

To illustrate the difference between Metagol and Metagol_{ho}, suppose you have compiled BK containing the `succ/2`, `int_to_char/2`, and `map/3` predicates and the *curry1* ($P(A,B) \leftarrow Q(A,B,R)$) and *chain* ($P(A,B) \leftarrow Q(A,C),R(C,B)$) metarules. Suppose you are given the examples `f([1,2,3],[c,d,e])` and `f([1,2,1],[c,d,c])` where the underlying target hypothesis is to add two to each element of the list and find the corresponding letter in an a-z index. Given these examples Metagol would try to prove each atom in turn. Metagol cannot prove any example using only the compiled BK so it would need to use a metarule. Suppose it unifies the atom `f([1,2,3],[c,d,e])` with the *curry* metarule. Then the new atom to prove would be `Q([1,2,3],[c,d,e],R)`.

[5] Metagol could support the negation of invented predicates but it is non-trivial to efficiently negate an invented predicate that itself contains an invented predicate. This limitation could be addressed by allowing Metagol to alternatively perform a generate and test approach. However, a generate-and-test approach is impractical for non-trivial situations.

```
learn(Pos,Neg,Prog):-
    prove(Pos,[],Prog),
    \+ prove(Neg,Prog,Prog).
prove([],Prog,Prog).
prove([Atom|Atoms],Prog1,Prog2):-
    prove_aux(Atom,Prog1,Prog3),
    prove(Atoms,Prog3,Prog2).
prove_aux(Atom,Prog,Prog):-
    prim(Atom),!,
    call(Atom).
prove_aux(Atom,Prog1,Prog2):-
    ibk((Atom:-Body)),
    prove(Body,Prog1,Prog2).
prove_aux(Atom,Prog1,Prog2):-
    member(sub(Name,Subs),Prog1),
    metarule(Name,Subs,(Atom:-Body)),
    prove(Body,Prog1,Prog2).
prove_aux(Atom,Prog1,Prog2):-
    metarule(Name,Subs,(Atom:-Body)),
    prove(Body,[sub(Name,Subs)|Prog1],Prog2).
```

Fig. 6: Prolog code for Metagol$_{ho}$.

To prove this atom Metagol could unify `map/3` with `Q` and then try to prove the atom `map([1,2,3],[c,d,e],R)`. However, the proof of `map([1,2,3],[c,d,e],R)` would fail because there is no suitable substitution for `R`. The only possible substitution for R is `succ/2`, which will clearly not allow the proof to succeed. The only way Metagol can learn a consistent hypothesis is by successively chaining calls to `map(A,B,succ)` and `map(A,B,int_to_char)` using the *chain* metarule to learn:

```
f(A,B):-f1(A,C),f3(C,B)
f1(A,B):-f2(A,C),f2(C,B).
f2(A,B):-map(A,B,succ).
f3(A,B):-map(A,B,int_to_char).
```

By contrast, suppose we had the same setup for Metagol$_{ho}$ but we allowed `map/3` to be defined as IBK. In this case, Metagol$_{ho}$ would unify the atom `f([1,2,3],[c,d,e])` with the *curry1* metarule. The new atom to prove would therefore be `Q([1,2,3],[c,d,e],R)`. In contrast to Metagol, Metagol$_{ho}$ can unify this atom with `map/3` defined as IBK. Metagol$_{ho}$ will then try to prove `map([1,2,3],[c,d,e],R)` through meta-interpretation. This step would result in a sequence of new atoms to prove `R(1,c), R(2,d), R(3,e)`. These new atoms can also be proven though meta-interpretation which allows for Metagol$_{ho}$ to invent and define the suitable symbol for `R`. Therefore, in this scenario, Metagol would learn:

```
f(A,B):-map(A,B,f1).
f1(A,B):-succ(A,C),f2(C,B).
f2(A,B):-succ(A,C),int_to_char(C,B).
```

```
empty([]).
head([H|_],H).
tail([_|T],T).
last([A],A):-!.
last([_|T],A):-last(T,A).
```

Compiled BK

```
ibk(([map,[],[],F]:-[])).
ibk(([map,[A|As],[B|Bs],F]:-[[F,A,B],[map,As,Bs,F]])).
ibk(([fold,[],Acc,Acc,F]:-[])).
ibk(([fold,[A|As],Acc1,B,F]:-[[F,A,Acc1,Acc2],[fold,As,Acc2,B,F]])).
ibk([ifthenelse,A,B,Cond,Then,_],[[Cond,A],[Then,A,B]]).
ibk([ifthenelse,A,B,Cond,_,Else],[[not,Cond,A],[Else,A,B]]).
```

Interpreted BK

```
metarule(monadic,[P,Q],([P,A,A]:-[[Q,A]])).
metarule(identity,[P,Q],([P,A,B]:-[[Q,A,B]])).
metarule(inverse,[P,Q],([P,A,B]:-[[Q,B,A]])).
metarule(didentity,[P,Q],([P,A,B]:-[[Q,A,B],[R,A,B]])).
metarule(precon,[P,Q,R],([P,A,B]:-[[Q,A],[R,A,B]])).
metarule(postcon,[P,Q,R],([P,A,B]:-[[Q,A,B],[R,B]])).
metarule(curry1,[P,Q,R],([P,A,B]:-[[Q,A,B,R]])).
metarule(curry2,[P,Q,R,S],([P,A,B]:-[[Q,A,B,R,S]])).
metarule(curry3,[P,Q,R,S,T],([P,A,B]:-[[Q,A,B,R,S,T]])).
metarule(chain,[P,Q,R],([P,A,B]:-[[Q,A,C],[R,C,B]])).
metarule(tailrec,[P,Q],([P,A,B]:-[[Q,A,C],[P,C,B]])).
```

Metarules

Fig. 7: Three forms of BK used by Metagol$_{ho}$ described in Prolog syntax. The curry rules are slightly unusual but are necessary to use the interpreted BK (e.g. *curry1* allows us to use the `map/3` definition).

As this scenario illustrates, the real power and novelty of Metagol$_{ho}$ is the combination of abstraction (learning higher-order programs) and invention (predicate invention). In this scenario, abstraction has allowed the atom `Q([1,2,3],[c,d,e],R)` to be decomposed into the sub-problems `R(1,c), R(2,d), R(3,e)`. Further abstraction and invention allows for Metagol$_{ho}$ to solve these sub-problems by inventing and defining the necessary predicate for `R`. By successively interleaving these two steps, Metagol$_{ho}$ supports the invention of conditions and functions to an arbitrary depth, which goes beyond anything in the literature.

4.3 HEXMIL

Before describing HEXMIL$_{ho}$, which supports learning higher-order logic programs, first we discuss HEXMIL, on which HEXMIL$_{ho}$is based.

HEXMIL is an answer set programming (ASP) encoding of MIL introduced by Kaminski et al. [23]. Whereas Metagol searches for a proof (and thus a program) using a meta-interpreter and SLD-resolution, HEXMIL searches for a proof by encoding the MIL problem as an ASP problem. As argued by Kaminski et al., an ASP implementation can be more efficient than a Prolog implementation because ASP solvers employ efficient conflict propagation, which is important for detecting the derivability of negative examples early during ASP search.

The HEXMIL encoding specifies constraints on possible hypotheses derived from the examples, in addition to rules specifying the available BK. An ASP solver performs a combinatorial search for solutions satisfying these constraints. ASP solvers typically work in two phases: (1) a grounding phase, where rules are grounded, and (2) a solving phase, where reasoning on (propositional) rules leads to answer sets [19]. A straightforward ASP encoding of the MIL problem is infeasible in many cases, for reasons such as the grounding bottleneck of ASP and the difficulty in manipulating complex structures such as lists [23]. To mitigate these difficulties HEXMIL uses the HEX formalism [13] which allows ASP programs to interface with *external sources*. External sources are predicate definitions given by programs outside of the ASP language. For instance, HEXMIL interfaces with external sources described as a Python program. HEX programs can access these definitions via *external atoms*. HEXMIL benefits from external atoms by allowing for arbitrary encodings of complex structures (e.g. we encode lists as strings, thereby reducing the number of variables needed in the encoding). Another benefit is that external atoms allow for the incremental introduction of new constants (i.e. symbols not in the initial ASP program).

To improve efficiency, Kaminski et al. introduced a *forward-chained* HEXMIL-encoding which requires forward-chained metarules:

Definition 7 (Forward-chained metarule) A metarule is *forward-chained* when it can be written in the form:

$$P(A,B) \leftarrow Q_1(A,C_1), Q_2(C_1,C_2), \ldots, Q_i(C_{i-1},B), R_1(D_1), \ldots, R_j(D_j)$$

where $D_1, \ldots, D_j$ are all contained in $\{A, C_1, \ldots, C_{i-1}, B\}$.

In the forward-chained HEXMIL encoding, compiled (first-order) BK is encoded using the external atoms `&bkUnary[P,A]()` and `&bkBinary[P,A](B)`. These two atoms represent all BK predicates of the form `P(A)` and `P(A,B)`, where `P` and `A` are input arguments to the external source and `B` is an output argument. Using the input/output ordering of the external binary atoms, grounding of variables in forward-chained metarules occurs from left to right. HEXMIL uses the forward-chained encoding:

deduced(P,A) ← &bkUnary[P,A](), state(A)
deduced(P,A,B) ← &bkBinary[P,A](B), state(A)
state(A) ← for each P(A,B) $\in E^+ \cup E^-$
state(B) ← deduced(P,A,B)

HEXMIL uses the *deduced* predicate to represent facts that hypotheses could entail. In this encoding, the import of BK is guarded by the predicate *state/1*. A solution for MIL problem (Definition 4) must entail all positive examples (i.e ground atoms). Therefore, in HEXMIL, every positive examples must appear in the head of a grounded metarule. It follows that ground terms in atoms can be seen as the states that can be reached from the examples. Therefore, HEXMIL initially marks the ground terms that appear in the

examples as *state*. As new ground terms are introduced by the external atoms, HEXMIL marks these values as *state* as well.

To support metarules HEXMIL employs two encoding rules. The first rule encodes the possible instantiations of a metarule. Let mr be the name of an arbitrary forward-chained metarule (Def. 7), then for each such metarule, the first encoding rule is:

$$\begin{aligned}&meta(mr,P,Q_1,\ldots,Q_i,R_1,\ldots,R_j) \lor neg_meta(mr,P,Q_1,\ldots,Q_i,R_1,\ldots,R_j) \leftarrow\\&\quad sig(P),sig(Q_1),\ldots,sig(Q_i),sig(R_1),\ldots,sig(R_j),\\&\quad ord(P,Q_1),\ldots,ord(P,Q_i),ord(P,R_1),\ldots,ord(P,R_j),\\&\quad deduced(Q_1,A,C_1),\ldots,deduced(Q_i,C_{i-1},B),\\&\quad deduced(R_1,D_1),\ldots,deduced(R_j,D_j)\end{aligned}$$

Note that the head in this rule allows for choosing whether to deduce the metarule instantiation. Also note that the disjunction in the head means that this is not a Horn clause, yet it encodes a Horn clause metarule. This encoding rule relies on two other rules:

$$\begin{aligned}&sig(p) \leftarrow \textit{for each } p \in \mathscr{P}\\&ord(p,q) \leftarrow \textit{for all } p,q \in \mathscr{P} \textit{ s.t. } p \preceq q\end{aligned}$$

The sig relation denotes predicate symbols available, both invented and given as part of the BK. The ord relation denotes an ordering $\preceq$ over the predicate symbols. This ordering disallows certain instantiations[6], e.g. recursive instantiations.

The second metarule encoding allows for metarule instantiations to be generated in order to derive facts:

$$\begin{aligned}&deduced(P,A,B) \leftarrow\\&\quad meta(mr,P,Q_1,\ldots,Q_i,R_1,\ldots,R_j),\\&\quad deduced(Q_1,A,C_1),\ldots,deduced(Q_i,C_{i-1},B),\\&\quad deduced(R_1,D_1),\ldots,deduced(R_j,D_j)\end{aligned}$$

The generation of metarule instantiations are then checked by the solver for consistency with the examples. This checking step relies on constraints derived from positive and negative examples:

$$\begin{aligned}&\leftarrow not\ deduce(P,A,B) \quad \text{for each } P(A,B) \in E^+\\&\leftarrow deduce(P,A,B) \quad \text{for each } P(A,B) \in E^-\end{aligned}$$

Similar to Metagol, HEXMIL searches for solutions using iterative deepening on the number of allowed metarule instantiations and the number of predicate symbols. We omit the details of the ASP constraints that restrict the number of metarule instantiations.

4.4 HEXMIL$_{ho}$

We now describe the extension of HEXMIL to HEXMIL$_{ho}$, which adds support for higher-order definitions, i.e. interpreted background knowledge (IBK). This extension allows HEXMIL to search for programs in *abstracted* forward-chained hypothesis spaces. To extend HEXMIL, we introduce a new predicate ibk to encode the higher-order atoms that occur in IBK. Note that ibk is a normal ASP predicate and not an external atom. This

[6] Details on the $\preceq$-relation can be found in the paper on HEXMIL [23] as well as in the files on our experimental work.

predicate allows us to encode higher-order clauses as a mix of $deduced$ atoms for first-order predicates and ibk atoms for those that involve predicates as arguments.

Let the following be a clause of an arbitrary (forward-chained) higher-order definition:

$$h(A,B,P_{0,1},\dots,P_{0,k_0}) \leftarrow h_1(A,C_1,P_{1,1},\dots,P_{1,k_1}),\dots,h_j(C_{j-1},B,P_{j,1},\dots,P_{j,k_j})$$

Every atom in this clause can have $0 \leq k_i$ higher-order terms. The higher-order clauses of the definition will have at least one atom with $k_i \neq 0$. For each clause in a higher-order definition we give a rule encoding the clause, where $C_0 = A$ and $C_j = B$:

$$\begin{array}{ll} ibk(h,A,B,P_{0,1},\dots,P_{0,k_0}) \leftarrow & \\ \quad state(A), & \\ \quad sig(P_{0,1}),\dots,sig(P_{0,k_0}), & \\ \quad ibk(h_i,C_{i-1},C_i,P_{i,1},\dots,P_{i,k_i}),sig(P_{i,1}),\dots,sig(P_{i,k_i}) & \text{if } k_i > 0 \\ \quad deduced(h_i,C_{i-1},C_i) & \text{if } k_i = 0 \end{array}$$

Figure 8 shows an example of this encoding for the `until/4` predicate. Figure 8 also contains a definition for `map/3` (which is slightly more involved). This approach to higher-order definitions also applies to metarules involving higher-order atoms. For instance, Figure 8 also shows the encoding of the *curry2* metarule.

Our extension is sufficient[7] to learn higher-order programs. Note that in this setting higher-order definitions are required to be forward-chained in their first-order arguments, meaning that left-to-right grounding of these arguments is still valid. The remaining (higher-order) arguments can be ground by the sig predicate, which contains all the predicate names. As predicate symbols were already arguments in the HEXMIL encoding, we can easily make a predicate argument occur as an atom's predicate symbol, e.g. see the variable `F` in `until/4` and `map/3` in Figure 8.

4.5 Complexity of the search

The experiments in the next section use both Metagol and HEXMIL, and their higher-order extensions. The purpose of the experiments is to test our claim that learning higher-order programs, rather than first-order programs, can improve learning performance. Although we do not directly compare them, the experimental results show a significant difference in the learning performances of Metagol and HEXMIL, and their higher-order variants. The experimental results also show that HEXMIL and HEXMIL$_{ho}$ do not scale well, both in terms of the amount of BK and the number of training examples. To help explain these results, we now contrast the theoretical complexity of Metagol and HEXMIL. For simplicity we focus on the $\mathscr{M}^2_2$ hypothesis space, although our results can easily be generalised. Our main observation is that the performance of HEXMIL is a function of the number of constant symbols, which is not the case for Metagol.

From Proposition 1 it follows that the $\mathscr{M}^2_2$ MIL hypothesis space contains at most $(mp^3)^n$ programs. For Metagol, this bound is an over-approximation on the number of programs that will be considered during the search. Given a training example, Metagol learns a program by trying different substitutions for the existentially quantified predicate symbols in metarules, where the search is driven by the example. Metagol only

[7] Kaminski, et al. [23] proposed an additional first-order *state-abstraction* encoding that improved the efficiency of the learning. It is currently unclear as how to integrate IBK into this encoding.

```
meta(curry2,P,Q,R,S) ∨ neg_meta(curry2,P,Q,R,S):-
    sig(P),sig(Q),sig(R),sig(S),
    ord(P,Q),ord(P,R),ord(P,S),
    ibk(Q,A,B,R,S).
deduced(P,A,B):-
    meta(curry2,P,Q,R,S),
    ibk(Q,A,B,R,S).
```

curry2 metarule

```
ibk(until,A,A,Cond,F):-
    state(A),
    sig(Cond),
    sig(F),
    deduced(Cond,A).
ibk(until,A,B,Cond,F):-
    state(A),
    sig(Cond),
    sig(F),
    not deduced(Cond,A),
    deduced(F,A,C),
    ibk(until,C,B,Cond,F).
```

`until/4`

```
ibk(map,"[]","[]",F):-
    sig(F).
ibk(map,L1,L2,F):-
    state(L1),
    sig(F),
    deduced(head,L1,H1),
    deduced(tail,L1,T1),
    deduced(F,H1,H2),
    ibk(map,T1,T2,F),
    &prepend[H2,T2](L2).
```

`map/3`

Fig. 8: HEXMIL$_{ho}$ code examples. The "[]" symbol in the `map/3` definition is special syntax we use to represent lists. Note that due to lists being encoded as strings, the `prepend` external atom is required to manipulate the lists in the `map/3` definition.

considers constants that it encounters when it evaluates whether a hypothesis covers an example, in which case it only considers the constant symbols pertaining to that particular example (in fact it delegates this step to Prolog). It follows that the search complexity of Metagol is independent of the number of constant symbols and is the same[8] as Proposition 1.

By contrast, HEXMIL searches for a program by instantiating metarules in a bottom-up manner where the body atoms of metarules need to be grounded. This approach means that the number of options that HEXMIL considers is not only a function of the number of metarules and predicate symbols (as is the case for Metagol), but it is also a function of the number of constant symbols because it needs to ground the first-order variables in a metarule. Even in the more efficient forward-chained MIL encoding, which incrementally imports new constants, body atoms are ground using many constant symbols unrelated to the examples. Any constant that can be marked as a state will be used to ground atoms. Therefore, the search complexity of HEXMIL is bounded by $(mp^3c^6)^n$,

[8] Metagol is sensitive to the size of the examples and to the computational complexity of a hypothesis because, as Schapire showed [40], if checking whether a hypothesis H covers an example e cannot be performed in time polynomial in the size of e and H then H cannot be learned in time polynomial in the size of e and H, i.e. Metagol needs to execute a learned program on the example.

where m is the number of metarules, p is the number of predicate symbols, n is a maximum program size, and c is the number of constant symbols.

For simplicity, the above complexity reasoning was for the first-order systems. We can easily apply the same reasoning to the abstracted MIL setting.

5 Experiments

Our main claim is that compared to learning first-order programs, learning higher-order programs can improve learning performance. Theorem 1 supports this claim and shows that, compared to unabstracted MIL, abstraction in MIL reduces sample complexity proportional to the reduction in the number of clauses required to represent hypotheses. We now experimentally[9] explore this result. We describe four experiments which compare the performance when learning first-order and higher-order programs. We test the null hypotheses:

Null hypothesis 1 Learning higher-order programs cannot improve predictive accuracies

Null hypothesis 2 Learning higher-order programs cannot reduce learning times

To test these hypotheses we compare Metagol with Metagol$_{ho}$ and HEXMIL with HEXMIL$_{ho}$, i.e. we compare unabstracted MIL with abstracted MIL.

5.1 Common materials

In the Prolog experiments we use the same metarules and IBK in each experiment, i.e. the only variable in the Prolog experiments is the system (Metagol or Metagol$_{ho}$). We use the metarules shown in Figure 7. We use the higher-order definitions `map/3`, `until/4`, and `ifthenelse/5` as IBK. We run the Prolog experiments using SWI-Prolog 7.6.4 [43].

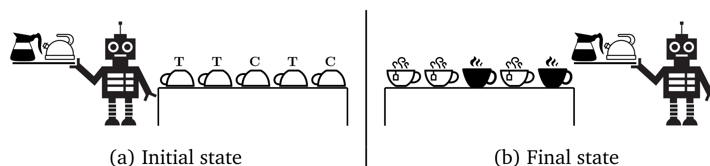
We tried to use the same experimental methodology in the ASP HEXMIL experiments as in the Prolog experiments but HEXMIL failed to learn any programs (first or higher-order) because of scalability issues. Therefore, in each ASP experiment we use the exact metarules and background relations necessary to represent the target hypotheses. We run the ASP experiments using Hexlite 1.0.0[10]. We run Hexlite with the *flpcheck* disabled. We also set Hexlite to enumerate a single model.

5.2 Robot waiter

Imagine teaching a robot to pour tea and coffee at a dinner table, where each setting has an indication of whether the guest prefers tea or coffee. Figure 9 shows an example in terms of initial and final states. This experiment focuses on learning a general robot waiter strategy [7] from a set of examples.

[9] All the experimental code and materials, including the code for Metagol, HEXMIL, and their higher-order extensions, is available at https://github.com/andrewcropper/mlj19-metaho

[10] https://github.com/hexhex/hexlite



(a) Initial state

(b) Final state

Fig. 9: Figures (a) and (b) show initial/final state waiter examples respectively. In the initial state, the cups are empty and each guest has a preference for tea **(T)** or coffee **(C)**. In the final state, the cups are facing up and are full with the guest's preferred drink.

5.2.1 Materials

Examples are `f/2` atoms where the first argument is the initial state and the second is the final state. A state is a list of ground Prolog atoms. In the initial state, the robot starts at position 0, there are d cups facing down at positions $0,\dots,d-1$; and for each cup there is a preference for tea or coffee. In the final state, the robot is at position d; all the cups are facing up; and each cup is filled with the preferred drink. We allow the robot to perform the fluents and actions (defined as compiled BK) shown in Figure 10.

at_end/1	wants_tea/1
wants_coffee/1	move_left/2
move_right/2	turn_cup_over/2
pour_tea/2	pour_coffee/2

Fig. 10: Compiled BK in the robot waiter experiment. We omit the definitions for brevity.

We generate positive examples as follows. For the Prolog experiments, for the initial state we select a random integer d from the interval $[1,20]$ as the number of cups. For the ASP experiments the interval is $[1,5]$. For each cup, we randomly select whether the preferred drink is tea or coffee and set it facing down. For the final state, we update the initial state so that each cup is facing up and is filled with the preferred drink. To generate negative examples, we repeat the aforementioned procedure but we modify the final state so that the drink choice is incorrect for a random subset of $k>0$ drinks.

5.2.2 Method

Our experimental method is as follows. For each learning system s and for each m in $\{1,2,\dots,10\}$:

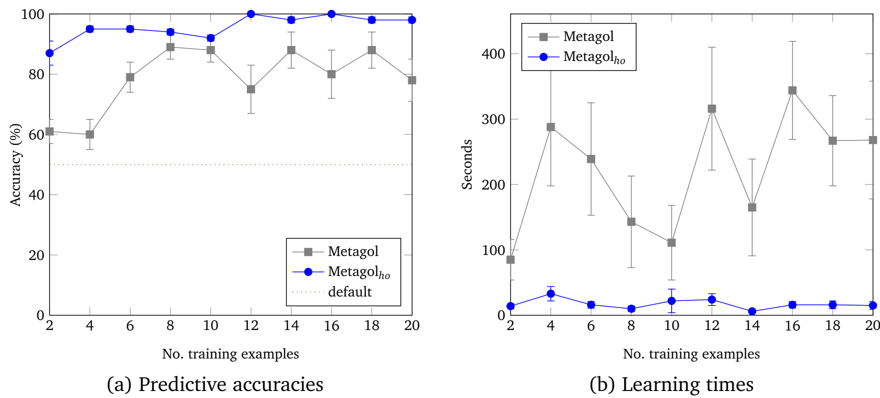
1. Generate m positive and m negative training examples
2. Generate 1000 positive and 1000 negative testing example
3. Use s to learn a program p using the training examples
4. Measure the predictive accuracy of p using the testing examples

If no program is found in 10 minutes then we deem that every testing example is false. We measure mean predictive accuracies, mean learning times, and standard errors of the mean over 10 repetitions.

5.2.3 Results

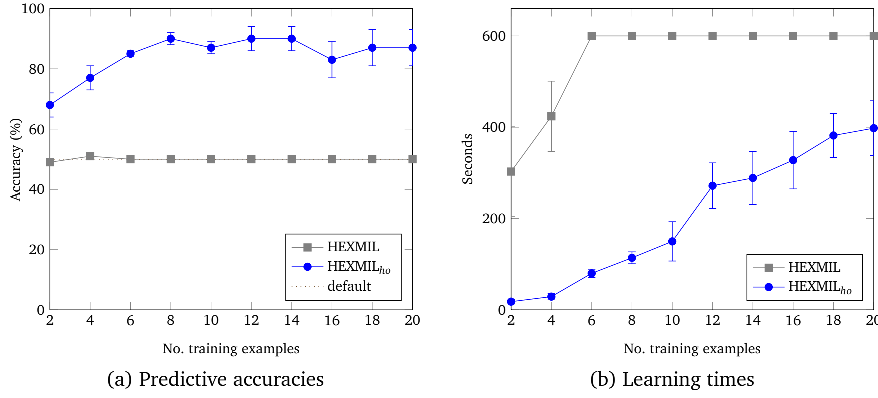
Figure 11 shows that in all cases Metagol$_{ho}$ learns programs with higher predictive accuracies and lower learning times than Metagol. Figure 12 shows similar results when comparing HEXMIL with HEXMIL$_{ho}$. We can explain these results by looking at example programs learned by Metagol and Metagol$_{ho}$ shown in Figures 13 and 14 respectively. Although both programs are general and handle any number of guests and any assignment of drink preferences, the program learned by Metagol$_{ho}$ is smaller than the one learned by Metagol. Whereas Metagol learns a recursive program, Metagol$_{ho}$ avoids recursion and uses the higher-order abstraction `until/4`. The abstraction `until/4` essentially removes the need to learn a recursive two clause definition to move along the dinner table. Likewise, Metagol$_{ho}$ uses the abstraction `ifthenelse/5` to remove the need to learn two clauses to decide which drink to pour. The compactness of the higher-order program affects predictive accuracies because, whereas Metagol$_{ho}$ almost always finds the target hypothesis in the allocated time, Metagol often struggles because the programs are too large, as explained by our theoretical results in Section 3.3. The results from this experiment suggest that we can reject null hypotheses 1 and 2.

Although we are not directly comparing the Prolog and ASP implementations of MIL, it is interesting to note that despite having more irrelevant BK, more irrelevant metarules, and having larger training instances, Metagol$_{ho}$ outperforms HEXMIL$_{ho}$ in all cases, both in terms of predictive accuracies and learning times. Figure 12 also shows that both HEXMIL and HEXMIL$_{ho}$ do not scale well in the number of training examples, especially the learning times. Our results in Section 4.5 help explain the poor scalability of HEXMIL and HEXMIL$_{ho}$ because more training examples typically means more constant symbols which in turn means a larger search complexity for both HEXMIL and HEXMIL$_{ho}$, although this issue can be mitigated using state abstraction [23].



(a) Predictive accuracies (b) Learning times

Fig. 11: Prolog robot waiter experiment results which show learning performance when varying the number of training examples.



(a) Predictive accuracies (b) Learning times

Fig. 12: ASP robot waiter experiment results which show learning performance when varying the number of training examples.

```
f(A,B):-turn_cup_over(A,C),f1(C,B).
f1(A,B):-move_right(A,B),at_end(B).
f1(A,B):-f2(A,C),f1(C,B).
f2(A,B):-wants_coffee(A),pour_coffee(A,B).
f2(A,B):-move_right(A,C),turn_cup_over(C,B).
f2(A,B):-wants_tea(A),pour_tea(A,B).
```

Fig. 13: An example first-order waiter program learned by Metagol.

```
f(A,B):-until(A,B,at_end,f1).
f1(A,B):-turn_cup_over(A,C),f2(C,B).
f2(A,B):-f3(A,C),move_right(C,B).
f3(A,B):-ifthenelse(A,B,wants_coffee,pour_coffee,pour_tea).
```

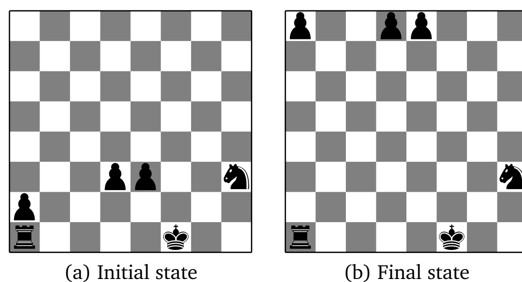
Fig. 14: An example higher-order waiter program learned by Metagol$_{ho}$.

5.3 Chess strategy

Programming chess strategies is a difficult task for humans [3]. For example, consider maintaining a wall of pawns to support promotion [21]. In this case, we might start by trying to inductively program the simple situation in which a black pawn wall advances without interference from white. Figure 15 shows such an example, where in the initial state the pawns are at different ranks and in the final state all the pawns have advanced to rank 8 but the other pieces have remained in the initial positions. In this experiment, we try to learn such strategies.

5.3.1 Materials

Examples are `f/2` atoms where the first argument is the initial state and the second is the final state. A state is a list of pieces, where a piece is denoted as a tuple of the form `(Type,Id,X,Y)`, where `Type` is the type (king=k, pawn=p, etc.), `Id` is a unique identifier,



(a) Initial state (b) Final state

Fig. 15: Chess initial/final state example.

X is the file, and Y is the rank. We generate a positive example as follows. For the initial state for the Prolog experiments, we select a random subset of n pieces from the interval $[1,16]$ and randomly place them on the board. For the ASP experiments the interval is $[1,5]$. For the final state, we update the initial state so that each pawn finishes at rank 8. To generate negative examples, we repeat the aforementioned procedure but we randomise the final state positions whilst ensuring that the input/output pair is not a positive example. We use the compiled BK shown in Figure 16.

```
rank8((_,_,_,8)).
not_rank8(A):-
    \+rank8(A).
pawn((p,_,_,_)).
not_pawn(A):-
    \+pawn(A).
head([A|_],A).
tail([_|T],T).
empty([]).
hold(A,A).
forward((Type,Id,X,Y1),(Type,Id,X,Y2)):-
    Y1 < 8,
    Y2 is Y1+1.
```

Fig. 16: Compiled BK used in the chess experiment.

5.3.2 Method

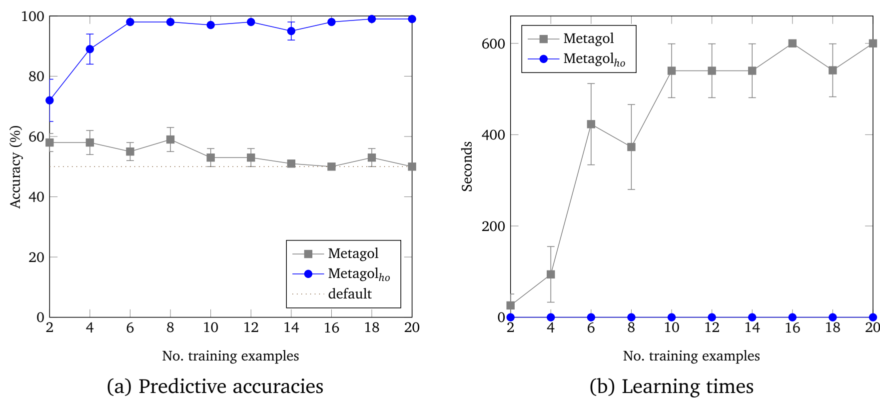
The experimental method is the same as in Experiment 1.

5.3.3 Results

Figure 17 shows that in all cases Metagol$_{ho}$ learns programs with higher predictive accuracies and lower learning times than Metagol. Figure 17 shows that Metagol$_{ho}$ learns programs approaching 100% accuracy after around six examples. By contrast, Metagol

learns programs with around default accuracy. Figure 18 shows similar results when comparing HEXMIL with HEXMIL$_{ho}$. The poor performance of Metagol and HEXMIL is because they both rarely find solutions in the allocated time. By contrast, Metagol$_{ho}$ and HEXMIL$_{ho}$ typically learn programs within two seconds.

We can again explain the performance discrepancies by looking at example learned programs in Figure 19. Figure 19b shows the compact higher-order program typically learned by Metagol$_{ho}$. This program is compact because it uses the abstractions `map/3` and `until/4`, where `map/3` decomposes the problem into smaller sub-goals of moving a single piece to rank eight and `until/4` solves the sub-problem of moving a pawn to rank eight. These sub-goals are solved by the invented `f1/2` predicate. By contrast, Figure 19a shows the large target first-order program that Metagol struggled to learn. As shown in Proposition 1, the MIL hypothesis space grows exponentially in the size of the target hypothesis, which is why the larger first-order program is more difficult to learn. The results from this experiment suggest that we can reject null hypotheses 1 and 2.



(a) Predictive accuracies (b) Learning times

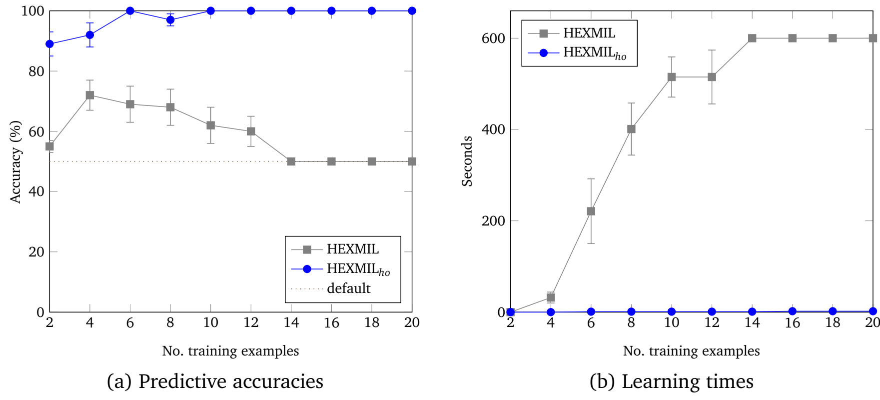
Fig. 17: Prolog chess experimental results which show predictive accuracy when varying the number of training examples. Note that Metagol$_{ho}$ typically learns a program in under two seconds.

5.4 Droplast

In this experiment, the goal is to learn a program that drops the last element from each sublist of a given list-of-lists – a problem frequently used to evaluate program induction systems [25]. In this experiment, we try to learn a program that drops the last character from each string in a list of strings. Figure 20 shows input/output examples for this problem described using the `f/2` predicate.

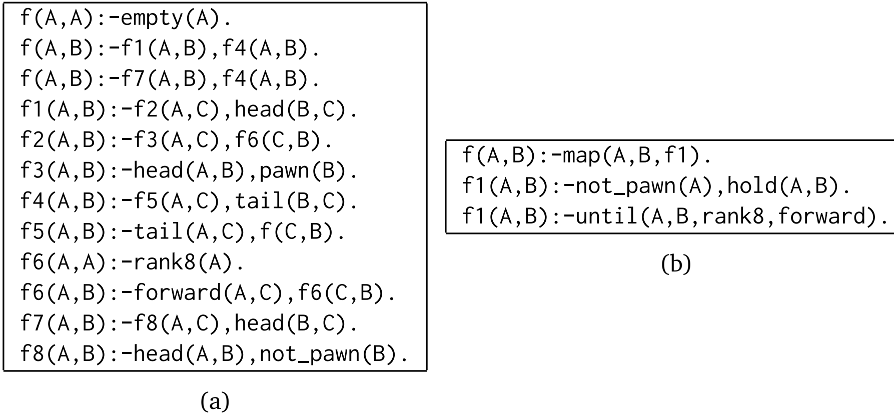
5.4.1 Materials

Examples are `f/2` atoms where the first argument is the initial list and the second is the final list. We generate positive examples as follows. For the Prolog experiments, to form the input, we select a random integer i from the interval $[1, 10]$ as the number of sublists.



(a) Predictive accuracies (b) Learning times

Fig. 18: ASP chess experimental results which show predictive accuracy when varying the number of training examples. Note that HEXMIL$_{ho}$ typically learns a program in under two seconds.



```
f(A,A):-empty(A).
f(A,B):-f1(A,B),f4(A,B).
f(A,B):-f7(A,B),f4(A,B).
f1(A,B):-f2(A,C),head(B,C).
f2(A,B):-f3(A,C),f6(C,B).
f3(A,B):-head(A,B),pawn(B).
f4(A,B):-f5(A,C),tail(B,C).
f5(A,B):-tail(A,C),f(C,B).
f6(A,A):-rank8(A).
f6(A,B):-forward(A,C),f6(C,B).
f7(A,B):-f8(A,C),head(B,C).
f8(A,B):-head(A,B),not_pawn(B).
```

(a)

```
f(A,B):-map(A,B,f1).
f1(A,B):-not_pawn(A),hold(A,B).
f1(A,B):-until(A,B,rank8,forward).
```

(b)

Fig. 19: Figure (a) shows the target first-order chess program, which Metagol could not learn within 10 minutes. Figure (b) shows the higher-order program often learned by Metagol$_{ho}$. The higher-order program is clearly smaller than the first-order program, which is why Metagol$_{ho}$ could typically learn it within a couple of seconds.

For each sublist i, we select a random integer k from the interval $[1, 10]$ and then sample with replacement a sequence of k letters from the alphabet a-z to form the sublist i. To form the output, we wrote a Prolog program to drop the last element from each sublist. For the ASP experiments the interval for i and k is $[1, 5]$.We generate negative examples using a similar procedure, but instead of dropping the last element from each sublist, we drop j random elements (but not the last one) from each sublist, where $1 < j < k$. We use the compiled BK shown in Figure 21.

5.4.2 Method

The experimental method is the same as in Experiment 1.

```
f([alice,bob,carol],[alic,bo,caro]).
f([inductive,logic,programming],[inductiv,logi,programmin]]).
f([ferrara,orleans,london,kyoto],[ferrar,orlean,londo,kyot]).
```

Fig. 20: Examples of the droplast problem. Note that in the experimental code we treat strings as lists of individual symbols, e.g. `alice` is represented as `[a,l,i,c,e]`.

```
head([H|_],H).
tail([_|T],T).
empty([]).
reverse(A,B):- ...
```

Fig. 21: Compiled BK used in the droplast experiment.

5.4.3 Results

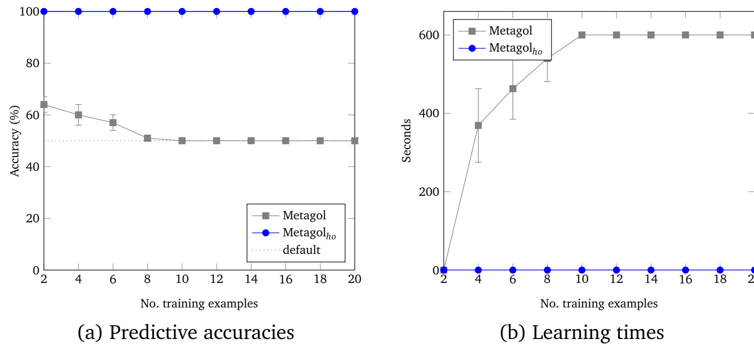
Figure 22 shows that Metagol$_{ho}$ achieved 100% accuracy after two examples at which point it learned the program shown in Figure 24a. This program again uses abstractions to decompose the problem. The predicate `f/2` maps over the input list and applies `f1/2` to each sublist to form the output list, thus abstracting away the reasoning for iterating over a list. The invented predicate `f1/2` drops the last element from a single list by reversing the list, calling `tail/2` to drop the head element, and then reversing the shortened list back to the original order. By contrast, Metagol was unable to learn any solutions because the corresponding first-order program is too long and the search is impractical, similar to the issues in the chess experiment.

Figure 23 shows slightly unexpected results for the ASP experiment. The figure shows that HEXMIL$_{ho}$ learns programs with higher predictive accuracies than HEXMIL when given up to 14 training examples. However, the predictive accuracies of HEXMIL$_{ho}$ progressively decreases given more examples. This performance degradation is because, as we have already explained, HEXMIL and HEXMIL$_{ho}$ do not scale well given more examples. This inability to scale given more examples is clearly shown in Figure 23, which shows that the learning times of HEXMIL$_{ho}$ increase significantly given more training examples.

We repeated the *droplast* experiment but replaced `reverse/2` in the BK with the higher-order definition `reduceback/3` and the compiled clause `concat/3`. In this scenario, Metagol$_{ho}$ learned the higher-order program shown in Figure 24b. This program now includes the invented predicate `f3/2` which reverses a given list and is used twice in the program. This more complex program highlights invention through the repeated calls to `f3/2` and abstraction through the use of higher-order functions.

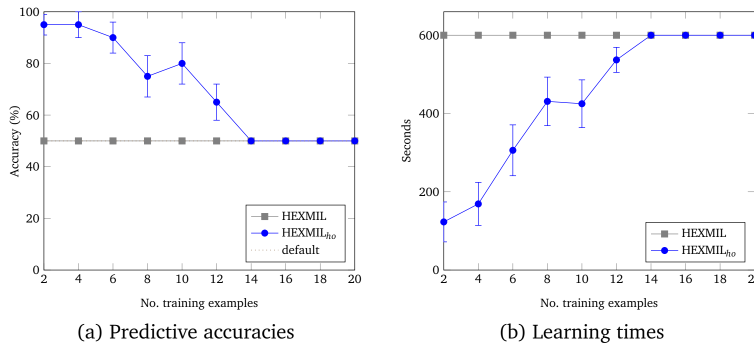
5.4.4 Further discussion

To further demonstrate invention and abstraction, consider learning a *double droplast* program which extends the droplast problem so that, in addition to dropping the last element from each sublist, it also drops the last sublist. Figure 25 shows examples of this problem, again represented as the target predicate `f/2`. Given two examples of this problem, Metagol$_{ho}$ learns the program shown in Figure 26a. For readability Figure 26b shows the folded program where non-reused invented predicates are removed. This pro-



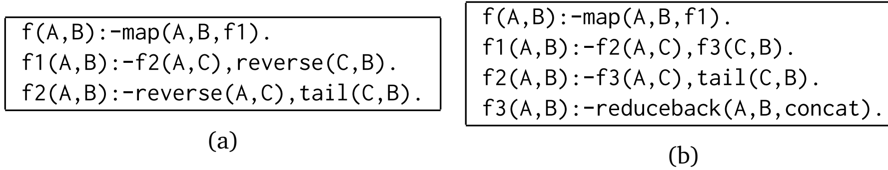
(a) Predictive accuracies (b) Learning times

Fig. 22: Prolog droplast experimental results which show predictive accuracy when varying the number of training examples. Note that Metagol$_{ho}$ typically learns a program in under two seconds.



(a) Predictive accuracies (b) Learning times

Fig. 23: ASP droplast experimental results which show predictive accuracy when varying the number of training examples.



```
f(A,B):-map(A,B,f1).
f1(A,B):-f2(A,C),reverse(C,B).
f2(A,B):-reverse(A,C),tail(C,B).
```

(a)

```
f(A,B):-map(A,B,f1).
f1(A,B):-f2(A,C),f3(C,B).
f2(A,B):-f3(A,C),tail(C,B).
f3(A,B):-reduceback(A,B,concat).
```

(b)

Fig. 24: Figure (a) shows the higher-order program often learned by Metagol$_{ho}$. Figure (b) shows a more complex program learned by Metagol$_{ho}$ when we repeated the experiment but disallowed Metagol$_{ho}$ to use `reverse/2` and instead gave it `reduceback/3` and `concat/3`.

gram is similar to the program shown in Figure 24b but it makes an additional final call to the invented predicate `f1/2` which is used twice in the program, once as a higher-order argument in `map/3` and again as a first-order predicate. This form of higher-order abstraction and invention goes beyond anything in the existing literature.

```
f([alice,bob,carol],[alic,bo]).
f([inductive,logic,programming],[inductiv,logi]]).
f([ferrara,orleans,london,kyoto],[ferrar,orlean,londo]).
```

Fig. 25: Examples of the more complex double droplast problem.

```
f(A,B):-f1(A,C),f2(C,B).
f1(A,B):-map(A,B,f2).
f2(A,B):-f3(A,C),f4(C,B).
f3(A,B):-f4(A,C),tail(C,B).
f4(A,B):-reduceback(A,B,concat).
```

(a)

```
f(A,B):-map(A,C,f1),f1(C,B).
f1(A,B):-f2(A,C),tail(C,D),f2(D,B).
f2(A,B):-reduceback(A,B,concat).
```

(b)

Fig. 26: Figure (a) shows a the higher-order *double droplast* program learned by Metagol$_{ho}$. For readability Figure (b) shows the folded program in which non-reused invented predicates are removed. Note how in Figure (b) the predicate symbol `f1/2` is used both as an argument to `map/3` and as a standard literal in the clause defined by the head `f(A,B)`.

5.5 Encryption

In this final experiment, we revisit the encryption example from the introduction.

5.5.1 Materials

Examples are `f/2` atoms where the first argument is the encrypted string and the second is the unencrypted string. For simplicity we only allow the letters a-z. We generate a positive example as follows. For the Prolog experiments we select a random integer k from the interval $[1,20]$ to denote the unencrypted string length. For the ASP experiments we select k from the interval $[1,5]$. We sample with replacement a sequence y of length k from the set $\{a,b,\ldots,z\}$. The sequence y denotes the unencrypted string. We form the encrypted string x by shifting each character in y two places to the right, e.g. $a \mapsto c, b \mapsto d, \ldots, z \mapsto b$. The atom $f(x,y)$ thus represents a positive example. To generate negative examples we repeat the aforementioned procedure but we shift each character by n places where $0 \leq n < 25$ and $n \neq 2$. For the BK we use the relations `char_to_int/2`, `int_to_char/2`, `succ/2`, and `prec/2`, where, for simplicity, `succ(25,0)` and `prec(0,25)` hold.

5.5.2 Method

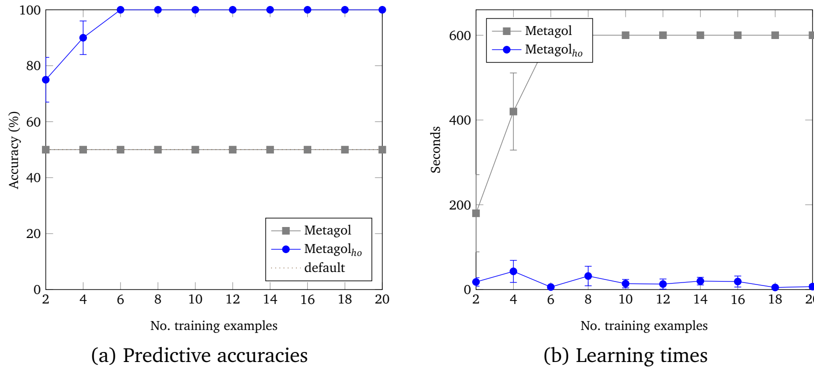
The experimental method is the same as in Experiment 1.

5.5.3 Results

Figure 27 shows that, as with the other experiments, Metagol$_{ho}$ learns programs with higher predictive accuracies and lower learning times than Metagol. These results are as expected because, as shown in Figure 4a, to represent the target encryption hypothesis as

a first-order program $\mathscr{M}^2_2$ requires seven clauses. By contrast, as shown in Figure 4b, to represent the target hypothesis as a higher-order program in $\mathscr{M}^2_2$ requires three clauses with one additional higher-order variable in the `map/3` abstraction.

We attempted to run the experiment using HEXMIL and HEXMIL$_{ho}$. However, both systems failed to find any programs within the timelimit. In fact, even in an extremely simple version of the experiment (where the alphabet contained only 10 letters, each string had at most 3 letters, and the character shift was +1) both systems failed to learn anything in the allocated time. Our theoretical results in Section 4.5 explain these empirical results. In this scenario, the number of ways that the BK predicates can be chained together and instantiated is no longer tractable for HEXMIL. The experiment suggests that HEXMIL needs to be better at determining which groundings are relevant to consistent hypotheses.



(a) Predictive accuracies (b) Learning times

Fig. 27: Prolog encryption experiment results which show learning performance when varying the number of training examples.

5.6 Discussion

Our main claim is that compared to learning first-order programs, learning higher-order programs can improve learning performance. Our experiments support this claim and show that learning higher-order programs can significantly improve predictive accuracies and reduce learning times.

Although it was not our purpose, our experiments also implicitly (implicitly because we do not directly compare the systems) show that Metagol outperforms HEXMIL, and similarly Metagol$_{ho}$ outperforms HEXMIL$_{ho}$. Our empirical results contradict those by Kaminski et al. [23], but support those by Morel et al. [30]. There are multiple explanations for this discrepancy. We think that the main problem is the ASP grounding problem faced by HEXMIL: in most of our experiments, HEXMIL timed out during the grounding (and not solving) stage. To alleviate this issue, future work could consider using state abstraction [23] to mitigate the grounding issues.

Also, by adjusting the experimental methodology, some of the results may change. For instance, Kaminski et al. showed that HEXMIL can sometimes learn solutions quicker than Metagol because of conflict propagation in ASP. They claim that this performance

improvement is because Metagol only considers negative examples after inducing a program from the positive examples (as described in Section 4.1). Therefore, HEXMIL should benefit from more negative examples, but may suffer from fewer.

To summarise, although our empirical results suggest that Metagol outperforms HEXMIL, future work should more rigorously compare the two approaches on multiple domains along multiple dimensions (e.g. varying the numbers of examples, size of BK, etc.).

6 Conclusions and further work

We have extended MIL to support learning higher-order programs by allowing for higher-order definitions to be included as background knowledge. We showed that learning higher-order programs can reduce the textual complexity required to express target classes of programs which in turn reduces the hypothesis space. Our sample complexity results show that learning higher-order programs can reduce the number of examples required to reach high predictive accuracies. To learn higher-order programs, we introduced Metagol$_{ho}$, a MIL learner which also supports higher-order predicate invention, such as inventing predicates for the higher-order abstractions `map/3` and `until/4`. We also introduced HEXMIL$_{ho}$, an ASP implementation of MIL that also supports learning higher-order programs. Our experiments showed that, compared to learning first-order programs, learning higher-order programs can significantly improve predictive accuracies and reduce learning times.

6.1 Limitations and future work

6.1.1 Metarules

There are at least two limitations with our work regarding the choice of metarules when learning higher-order programs.

One issue is deciding which metarules to use. Figure 7 shows the 11 metarules used in our experiments. Of these metarules, eight of them (the ones with only monadic or dyadic literals) are a subset of a *derivationally* irreducible set of monadic and dyadic metarules [12]. We can therefore justify their selection because they are sufficient to learn any program in a slightly restricted subset of Datalog. However, we have additionally used three *curry* metarules with arities three, four, and five, which were not considered in the work on identifying derivationally irreducible metarules. In addition, the curry metarules also include existentially quantified predicate arguments (e.g. R in $P(A,B) \leftarrow Q(A,B,R)$). Although these metarules seem intuitive and sensible to use, we have no theoretical justification for using them. Future work should address this issue, such as by extending the existing work [12] to include such metarules.

A second issue regarding the curry metarules is that when used with abstractions they each require an extra clause in the learned program. Our motivation for learning higher-order programs was to reduce the number of clauses necessary to express a target theory. Although our theoretical and experimental results support this claim, further improvements can be made. For instance, suppose you are given examples of the concept $f(x,y)$ where x is a list of integers and y is x but reversed, where each element has had one added to it, and then doubled, such as $f([1,2,3],[8,6,4])$. Then Metagol$_{ho}$ could learn the following program given the metarules used in Figure 7:

```
f(A,B):-f1(A,C),f5(C,B).
f1(A,B):-f3(A,C),f4(C,B).
f3(A,B):-reduceback(A,B,concat).
f4(A,B):-map(A,B,succ).
f5(A,B):-map(A,B,double).
```

This program requires five clauses. By contrast, a more compact representation would be:

```
f(A,B):-reduceback(A,C,concat),
    map(C,D,succ),
    map(D,B,double).
```

This more compact program is formed of a single clause and four literals, so should therefore be easier to learn. Future work should try to address this limitation of the current approach[11].

6.1.2 Higher-order definitions

Our experiments rely on a few higher-order definitions, mostly based on higher-order programming concepts, such as `map/3` and `until/4`. Future work should consider other higher-order concepts. For instance, consider learning regular grammars, such as $a^*b^*c^*$. To improve learning efficiency it would be desirable to encode the concept of *Kleene star operator*[12] as a higher-order definition, such as:

```
kstar(P,A,A).
kstar(P,A,B):-
    call(P,A,C),
    kstar(P,C,B).
```

Similarly, we have used abstracted MIL to invent functional constructs. Future work could consider inventing relational constructs. For instance, consider this higher-order definition of a closure:

closure(P,A,B) ← P(A,B)
closure(P,A,B) ← P(A,C), closure(P,C,B)

We could use this definition to learn compact abstractions of relations, such as:

ancestor(A,B) ← closure(parent,A,B)
lessthan(A,B) ← closure(increment,A,B)
subterm(A,B) ← closure(headortail,A,B)

[11] This limitation is not specific to when learning higher-order programs. Curry metarules can also be used when learning first-order programs, where existentially quantified argument variables could be bound to constant symbols, rather than predicate symbols. In other words, this issue is a limitation of MIL but manifests itself clearly when learning higher-order programs.

[12] The Kleene star operator represents zero-or-more repetitions (here applications) of its argument.

6.1.3 Learning higher-order abstractions

One clear limitation of the current approach is that we require user-provided higher-order definitions, such as `map/3`. In future work we want to learn or invent such definitions. For instance, when learning a solution to the decryption program in the introduction it may be beneficial to learn and invent a sub-definition that corresponds to `map/3`. The program below shows such a scenario, where the definition `decrypt1/3` corresponds to `map/3`.

```
decrypt(A,B):-
    decrypt1(A,B,decrypt2).
decrypt1(A,B,C):-
    empty(A),
    empty(B).
decrypt1(A,B,C):-
    head(A,D),
    call(C,D,E),
    head(B,E),
    tail(A,F),
    tail(B,G),
    decrypt1(F,G,C).
decrypt2(A,B):-
    char_to_int(A,C),
    prec(C,D),
    int_to_char(D,B).
```

Our preliminary work suggests that learning such definitions is possible.

6.2 Summary

In summary, our primary contribution is a demonstration of the value of higher-order abstractions and inventions in MIL. We have shown that the techniques allow us to learn substantially more complex programs using fewer examples with less search.

Acknowledgements We thank Stassa Patsantzis and Tobias Kaminski for helpful feedback on the paper.

References

1. Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. *Artif. Intell.*, 101(1-2):285–297, 1998.
2. Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Occam's razor. *Inf. Process. Lett.*, 24(6):377–380, 1987.
3. Ivan Bratko and Donald Michie. A representation for pattern-knowledge in chess endgames. *Advances in Computer Chess*, 2:31–56, 1980.
4. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, 1985.
5. K.L. Clark. Negation as failure. In M. L. Ginsberg, editor, *Readings in Nonmonotonic Reasoning*, pages 311–325. Kaufmann, Los Altos, CA, 1987.

6. Andrew Cropper. *Efficiently learning efficient programs*. PhD thesis, Imperial College London, UK, 2017.
7. Andrew Cropper and Stephen H. Muggleton. Learning efficient logical robot strategies involving composable objects. In Qiang Yang and Michael Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 3423–3429. AAAI Press, 2015.
8. Andrew Cropper and Stephen H. Muggleton. Learning higher-order logic programs through abstraction and invention. In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 1418–1424. IJCAI/AAAI Press, 2016.
9. Andrew Cropper and Stephen H. Muggleton. Metagol system. https://github.com/metagol/metagol, 2016.
10. Andrew Cropper and Stephen H. Muggleton. Learning efficient logic programs. *Machine Learning*, Apr 2018.
11. Andrew Cropper, Alireza Tamaddoni-Nezhad, and Stephen H. Muggleton. Meta-interpretive learning of data transformation programs. In Katsumi Inoue, Hayato Ohwada, and Akihiro Yamamoto, editors, *Inductive Logic Programming - 25th International Conference, ILP 2015, Kyoto, Japan, August 20-22, 2015, Revised Selected Papers*, volume 9575 of *Lecture Notes in Computer Science*, pages 46–59. Springer, 2015.
12. Andrew Cropper and Sophie Tourret. Derivation reduction of metarules in meta-interpretive learning. In Fabrizio Riguzzi, Elena Bellodi, and Riccardo Zese, editors, *Inductive Logic Programming - 28th International Conference, ILP 2018, Ferrara, Italy, September 2-4, 2018, Proceedings*, volume 11105 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2018.
13. Thomas Eiter, Michael Fink, Giovambattista Ianni, Thomas Krennwallner, Christoph Redl, and Peter Schüller. A model building framework for answer set programming with external computations. *TPLP*, 16(4):418–464, 2016.
14. Werner Emde, Christopher Habel, and Claus-Rainer Rollinger. The discovery of the equator or concept driven learning. In Alan Bundy, editor, *Proceedings of the 8th International Joint Conference on Artificial Intelligence. Karlsruhe, FRG, August 1983*, pages 455–458. William Kaufmann, 1983.
15. Cao Feng and Stephen Muggleton. Towards inductive generalization in higher order logic. In Derek H. Sleeman and Peter Edwards, editors, *Proceedings of the Ninth International Workshop on Machine Learning (ML 1992), Aberdeen, Scotland, UK, July 1-3, 1992*, pages 154–162. Morgan Kaufmann, 1992.
16. John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 229–239, 2015.
17. Pierre Flener and Serap Yilmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. *J. Log. Program.*, 41(2-3):141–195, 1999.
18. Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 802–815. ACM, 2016.
19. Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386, 1991.
20. Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 317–330, 2011.
21. Larry Harris. The heuristic search and the game of chess. a study of quiescence, sacrifices, and plan oriented play. In *Computer Chess Compendium*, pages 136–142. Springer, 1988.
22. Katsumi Inoue, Andrei Doncescu, and Hidetomo Nabeshima. Completing causal networks by meta-level abduction. *Machine Learning*, 91(2):239–277, 2013.
23. Tobias Kaminski, Thomas Eiter, and Katsumi Inoue. Exploiting answer set programming with external sources for meta-interpretive learning. *TPLP*, 18(3-4):571–588, 2018.
24. Susumu Katayama. Efficient exhaustive generation of functional programs using monte-carlo search with iterative deepening. In *PRICAI 2008: Trends in Artificial Intelligence, 10th Pacific Rim International Conference on Artificial Intelligence, Hanoi, Vietnam, December 15-19, 2008. Proceedings*, pages 199–210, 2008.
25. Emanuel Kitzelmann. Data-driven induction of functional programs. In *ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings*, pages 781–782, 2008.
26. J.W. Lloyd. *Logic for Learning*. Springer, Berlin, 2003.

27. Zohar Manna and Richard J. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.
28. John McCarthy. Making robots conscious of their mental states. In *Machine Intelligence 15, Intelligent Agents [St. Catherine's College, Oxford, July 1995]*, pages 3–17, 1995.
29. Tom M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997.
30. Rolf Morel, Andrew Cropper, and C.-H. Luke Ong. Typed meta-interpretive learning of logic programs. In Francesco Calimeri, Nicola Leone, and Marco Manna, editors, *Logics in Artificial Intelligence - 16th European Conference, JELIA 2019, Rende, Italy, May 7-11, 2019, Proceedings*, volume 11468 of *Lecture Notes in Computer Science*, pages 198–213. Springer, 2019.
31. Stephen Muggleton. Inverse entailment and progol. *New Generation Comput.*, 13(3&4):245–286, 1995.
32. Stephen Muggleton and Wray L. Buntine. Machine invention of first order predicates by inverting resolution. In *Machine Learning, Proceedings of the Fifth International Conference on Machine Learning, Ann Arbor, Michigan, USA, June 12-14, 1988*, pages 339–352, 1988.
33. Stephen Muggleton, Luc De Raedt, David Poole, Ivan Bratko, Peter A. Flach, Katsumi Inoue, and Ashwin Srinivasan. ILP turns 20 - biography and future challenges. *Machine Learning*, 86(1):3–23, 2012.
34. Stephen H. Muggleton, Dianhuan Lin, Niels Pahlavi, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning: application to grammatical inference. *Machine Learning*, 94(1):25–49, 2014.
35. Stephen H. Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. *Machine Learning*, 100(1):49–73, 2015.
36. Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 619–630. ACM, 2015.
37. J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.
38. Luc De Raedt and Maurice Bruynooghe. Interactive concept-learning and constructive induction by analogy. *Machine Learning*, 8:107–150, 1992.
39. Lorenza Saitta and Jean-Daniel Zucker. *Abstraction in artificial intelligence and complex systems*. Springer, 2013.
40. Robert E. Schapire. The strength of weak learnability. *Machine Learning*, 5:197–227, 1990.
41. A. Srinivasan. The ALEPH manual. *Machine Learning at the Computing Laboratory, Oxford University*, 2001.
42. Irene Stahl. The appropriateness of predicate invention as bias shift operation in ILP. *Machine Learning*, 20(1-2):95–117, 1995.
43. Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.