Learning Higher-Order Programs through Predicate Invention

Andrew Cropper
University of Oxford
andrew.cropper@cs.ox.ac.uk

Rolf Morel
University of Oxford
rolf.morel@cs.ox.ac.uk

Stephen H. Muggleton
Imperial College London
s.muggleton@imperial.ac.uk

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 ILP techniques to learn higher-order programs. We implement our idea in Metagol$_{ho}$, an ILP system which can learn higher-order programs with higher-order predicate invention. Our experiments show that, compared to first-order programs, learning higher-order programs can significantly improve predictive accuracies and reduce learning times.

1 Introduction

Suppose we want to learn a decryption program from encrypted/decrypted messages. Figure 1 shows such a scenario, where the encryption algorithm is a +1 Caesar cipher. Given these examples and suitable background knowledge (BK), most inductive logic programming (ILP) approaches would learn a recursive first-order program, such as the one shown in Figure 2. Although correct, this program is overly complex and most of the program manipulates the input and output list. In this paper[1], we introduce techniques to learn higher-order programs that abstract away this boilerplate code. Specifically, we extend meta-interpretive learning (MIL) (Muggleton, Lin, and Tamaddoni-Nezhad 2015) to support learning higher-order programs. Using this new approach, we can learn an equivalent[2] yet smaller program, such as the one shown in Figure 3, which uses *map/3* to abstract away the recursion and list manipulation.

Encrypted	Decrypted
joevdujwf	inductive
mphjd	logic
qsphsbnnjoh	programming

Figure 1: Example encrypted and decrypted messages.

```
f(A,B):-empty(A),empty(B).
f(A,B):-head(A,C),tail(A,G),head(B,F),tail(B,H),
  int_to_chr(E,F),chr_to_int(C,D),succ(E,D),f(G,H).
```

Figure 2: A first-order decryption program.

```
f(A,B):-map(A,B,f1).
f1(A,B):-chr_to_int(A,C),succ(D,C),int_to_chr(D,B).
```

Figure 3: A higher-order decryption program.

We implement our idea in Metagol$_{ho}$, which extends Metagol (Cropper and Muggleton 2016), a MIL implementation based on a Prolog meta-interpreter, to support learning higher-order programs. The key novelty of Metagol$_{ho}$ is the combination of abstraction (learning higher-order programs) and invention (predicate invention), allowing for higher-order arguments to be invented, such as the predicate *f1/2* in Figure 3. Our main contributions are:

- We show that learning higher-order programs can reduce the size of the hypothesis space and sample complexity in MIL.
- We introduce Metagol$_{ho}$, which extends Metagol to support learning higher-order programs.
- We experimentally show that, compared to learning first-order programs, learning higher-order programs can substantially improve predictive accuracies and reduce learning times.

2 Related Work

The goal of program induction is to learn a program from an incomplete specification, typically input/output examples. ILP is a form of program induction which learns logic programs. MIL is a powerful form of ILP that supports *predicate invention*, the ability to introduce new predicate symbols to improve learning performance.

Most ILP systems, including the popular system Progol (Muggleton 1995), learn first-order programs. Given appropriate mode declarations (Muggleton 1995) for higher-order predicates such as *map/3*, Progol could learn higher-order

[1]This is a shortened version of a MLJ paper presented at ILP19 (Cropper, Morel, and Muggleton 2019)

[2]Success set equivalent when restricted to the predicate *f/2*.

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

Figure 4: 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.

programs, such as *f(A,B)* ← *map(A,B,succ)*. However, because these systems do not support predicate invention they cannot invent any arguments for *map/3*, and would instead need the arguments to be predefined. Likewise, the main MIL implementation, Metagol, cannot learn the program in Figure 3 without being given the definition for the predicate *f1*. Metagol$_{ho}$ overcomes this limitation and can invent arguments for higher-order predicates.

3 Meta-Interpretive Learning

We now extend MIL to support learning higher-order programs and show the potential sample complexity benefits.

3.1 Preliminaries

We assume familiarity with logic programming notation, but we restate specific terminology. We denote the predicate and constant signatures as $\mathcal{P}$ and $\mathcal{C}$ 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 $\mathcal{V}_1$ and $\mathcal{V}_2$ respectively. A first-order term is a variable or a constant symbol. A higher-order term is a higher-order variable or a predicate symbol. 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 first-order if all of its terms are first-order. An atom is higher-order if it has at least one higher-order term. A literal is an atom or its negation. A Horn clause is a clause with at most one positive literal. A clause is higher-order if it contains at least one higher-order literal. 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

MIL uses metarules (Cropper and Tourret 2019) as a form of inductive bias to define the structure of learnable programs. A metarule is a higher-order formula of the form $\exists\pi\forall\mu \quad l_0 \leftarrow l_1, \ldots, l_m$, where each l_i is a literal, $\pi \subseteq \mathcal{V}_1 \cup \mathcal{V}_2$, $\mu \subseteq \mathcal{V}_1 \cup \mathcal{V}_2$, and π and μ are disjoint. When describing metarules, we omit the quantifiers and instead 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 4 shows example metarules.

To extend MIL to support learning higher-order programs we introduce higher-order definitions. A higher-order definition is a set of higher-order Horn clauses where the head atoms have the same predicate symbol. Figure 5 shows an example higher-order definition. We extend the standard MIL input to support higher-order definitions:

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

Figure 5: Higher-order map definition. The variables are all universally quantified first-order variables except the symbol F, which is a higher-order variable.

Definition 1 (**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.

We define the abstracted MIL problem:

Definition 2 (**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 (1) $\forall h \in H, \exists m \in M$ such that $h = m\theta$, where θ is a substitution that grounds all the existentially quantified variables in m, (2) $H \cup B \models E^+$, and (3) $H \cup B \not\models E^-$. We call H a solution to the MIL problem.

3.3 Sample Complexity

The size of the MIL 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. 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 4 is in the fragment $\mathcal{M}^2_2$. By restricting the form of metarules we can calculate the size of a MIL hypothesis space:

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 $(mp^{j+1})^n$.

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 $(mp^{j+1+k})^n$.

We use this result to develop sample complexity 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 $s_u \geq \frac{1}{\epsilon}(n_u \ \ln(m) + (j+1)n_u \ \ln(p) + \ln(\frac{1}{\delta}))$.

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

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 the number of additional higher-order variables used by abstracted MIL. Then $s_u > s_a$ when $n_u - n_a > \frac{k}{j+1} n_a$.

The results from this section motivate the use of abstracted MIL and help explain the experimental results (Section 5).

4 Metagol$_{ho}$

4.1 Metagol

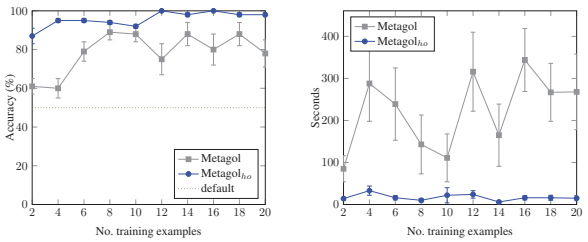
Metagol$_{ho}$ is based on Metagol, a MIL learner based on a Prolog meta-interpreter. Given a set of atoms which represent positive examples, Metagol tries to prove each atom in turn. Metagol first tries to deductively prove an atom using standard BK by delegating the proof to Prolog. If this deductive step fails, Metagol tries to unify the atom with the head of a metarule and tries to bind the existentially quantified variables in a metarule to symbols in the predicate signature. Metagol saves the resulting substitutions and tries to prove the body of the metarule through meta-interpretation. After proving all atoms, Metagol induces a Prolog program 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 programs.

4.2 Metagol$_{ho}$

Metagol$_{ho}$ extends Metagol with an additional clause in the meta-interpreter. This additional clause allows Metagol$_{ho}$ to prove an atom using a clause in a higher-order definition (such as *map/3*), and, as with metarules, proves the body of the clause through meta-interpretation. This meta-interpretive approach allows for predicate invention to be driven by the proof of conditions (as in *filter/3*) and functions (as in *map/3*). It is important to note that proving a clause in a higher-order definition is different to using a metarule because the variables in a higher-order definition are all universally quantified. By contrast, metarules contain existentially quantified variables whose substitutions form the hypothesised program, and thus requires a search for appropriate substitutions.

5 Experiments

We now experimentally test our claim that, compared to learning first-order programs, learning higher-order programs can improve learning performance. In the experiments we use



(a) Predictive accuracies (b) Learning times

Figure 6: Robot waiter experiment results.

10 general-purpose metarules and the higher-order definitions *map/3*, *until/4*, and *ite/5* (if then else).

5.1 Robot Waiter

This experiment focuses on learning a general robot waiter strategy (Cropper and Muggleton 2015) from a set of examples, where the goal is to serve drinks at a table.

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 $0, \ldots, d-1$; 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 (monadic predicates) *at_end*, *wants_tea*, and *wants_coffee*, and to perform the actions (dyadic predicates) *move_left*, *move_right*, *turn_cup_over*, *pour_tea*, and *pour_coffee*.

Method For each learning system s in {Metagol, Metagol$_{ho}$} and for each m in $\{1, 2, \ldots, 10\}$, we (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, and (4) measure the predictive accuracy of p using the testing examples. We measure mean predictive accuracies, mean learning times, and standard errors of the mean over 10 repetitions.

Results Figure 6 shows that in all cases Metagol$_{ho}$ learns programs with higher predictive accuracies and lower learning times than Metagol. Figures 7 and 8 show example programs learned by Metagol and Metagol$_{ho}$ respectively. Although both programs are general, the program learned by Metagol$_{ho}$ is smaller. Whereas Metagol learns a first-order recursive program, Metagol$_{ho}$ avoids recursion and instead uses the higher-order abstraction *until/4*, which removes the need to learn a recursive two clause definition. Likewise, Metagol$_{ho}$ uses the abstraction *ite/5* (if then else) to remove the need to learn two clauses to decide which drink to pour.

5.2 Droplast

In this experiment, we try to learn a program that drops the last character from each string in a list of strings, for instance *[alice,bob,carol]* maps to *[alic,bo,caro]*.

```
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).
```

Figure 7: A first-order waiter program learned by Metagol.

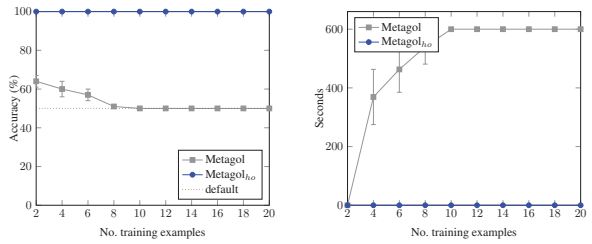
```
f(A,B):-until(A,B,at_end,f1).
f1(A,B):-turn_cup_over(A,C),f2(C,B).
f2(A,B):-f3(A,C),move_right(C,B).
f3(A,B):-ite(A,B,wants_coffee,pour_coffee,pour_tea).
```

Figure 8: A higher-order waiter program learned by Metagol$_{ho}$.

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. To form the input, we select a random integer from the interval $[1, 10]$ as the number of sublists. 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. 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 BK predicates *empty/1*, *head/2*, *tail/2*, and *reverse/2*.

Method The experimental method is the same as in Experiment 1.

Results Figure 9 shows that Metagol$_{ho}$ achieved 100% accuracy after two examples at which point it learned the program shown in Figure 10. 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.



(a) Predictive accuracies (b) Learning times

Figure 9: Droplast experimental results.

```
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).
```

Figure 10: A higher-order droplast program learned by Metagol$_{ho}$.

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 character from each string, it also drops the last string, e.g. *[alice,bob,carol]* maps to *[alic,bo]*. Given two examples of this problem, Metagol$_{ho}$ learns the program shown in Figure 11. This program is similar to the program shown in Figure 10 but it makes an additional 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(A,B):-map(A,C,f1),f1(C,B).
f1(A,B):-f2(A,C),reverse(C,B).
f2(A,B):-reverse(A,C),tail(C,B).
```

Figure 11: A *double droplast* program learned by Metagol$_{ho}$.

6 Conclusions and Limitations

We have shown that learning higher-order programs can reduce the size of the hypothesis space and sample complexity in MIL. We introduced Metagol$_{ho}$, a MIL learner which can learn higher-order programs with predicate invention. Our experiments showed that, compared to learning first-order programs, learning higher-order programs can significantly improve predictive accuracies and reduce learning times.

One limitation of this work is that we need pre-defined higher-order definitions, such as *map/3*. In future work we want to invent such definitions. For instance, when learning the decryption program in the introduction, it may be beneficial to learn and invent a sub-definition that corresponds to *map/3*.

References

Cropper, A., and Muggleton, S. H. 2015. Learning efficient logical robot strategies involving composable objects. In *IJCAI 2015*, 3423–3429. AAAI Press.

Cropper, A., and Muggleton, S. H. 2016. Metagol system. https://github.com/metagol/metagol.

Cropper, A., and Tourret, S. 2019. Logical reduction of metarules. *Machine Learning*.

Cropper, A.; Morel, R.; and Muggleton, S. 2019. Learning higher-order logic programs. *Machine Learning*.

Muggleton, S. H.; Lin, D.; and Tamaddoni-Nezhad, A. 2015. Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. *Machine Learning* 100(1):49–73.

Muggleton, S. 1995. Inverse entailment and progol. *New Generation Comput.* 13(3&4):245–286.