# Learning Higher-Order Logic Programs Through Abstraction and Invention

**Andrew Cropper and Stephen H. Muggleton**
Imperial College London
United Kingdom
{a.cropper13,s.muggleton}@imperial.ac.uk

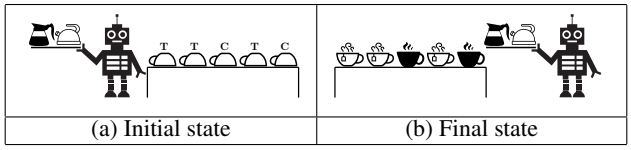
## Abstract

Many tasks in AI require the design of complex programs and representations, whether for programming robots, designing game-playing programs, or conducting textual or visual transformations. This paper explores a novel inductive logic programming approach to learn such programs from examples. To reduce the complexity of the learned programs, and thus the search for such a program, we introduce higher-order operations involving an alternation of Abstraction and Invention. Abstractions are described using logic program definitions containing higher-order predicate variables. Inventions involve the construction of definitions for the predicate variables used in the Abstractions. The use of Abstractions extends the Meta-Interpretive Learning framework and is supported by the use of a user-extendable set of higher-order operators, such as *map*, *until*, and *ifthenelse*. Using these operators reduces the textual complexity required to express target classes of programs. We provide sample complexity results which indicate that the approach leads to reductions in the numbers of examples required to reach high predictive accuracy, as well as significant reductions in overall learning time. Our experiments demonstrate increased accuracy and reduced learning times in all cases. We believe that this paper is the first in the literature to demonstrate the efficiency and accuracy advantages involved in the use of higher-order abstractions.

## 1 Introduction

Inductive Programming (IP) [Gulwani *et al.*, 2015] is a form of machine learning which aims to learn programs from examples given background knowledge (BK). To illustrate this form of machine learning, consider teaching a robot to pour tea and coffee for all place settings at a table. For each setting there is an indication of whether the associated guest prefers tea or coffee. Figure 1 shows an example in terms of an initial state (Figure 1a) and final state (Figure 1b).

Now consider learning a general strategy for the task from a set of such examples. Given that there may be an arbitrary number of place settings, existing approaches to IP, such as Meta-Interpretive Learning (MIL) [Muggleton *et al.*, 2015; Cropper and Muggleton, 2015a], would learn a recursive strategy, such as that shown in Figure 1c. In this paper, we extend the MIL framework to support learning theories with higher-order constructs, such as *map*, *until*, and *ifthenelse*. In this approach, an equivalent yet more compact strategy can be learned, as in Figure 1d. This is implemented in a system called Metagol$_{AI}$ which uses a form of interpreted BK to learn programs through a sequence of interleaved Abstraction and Invention steps (see Figure 1e). We show that the compactness of such definitions leads to substantially improved predictive accuracy and significantly reduced learning time.




(a) Initial state | (b) Final state

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

(c) First-order program

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

(d) Higher-order program

| → | **Abstract** | → | **Invent** | → | **Abstract** |
|---|---|---|---|---|---|
| f | until | | f3,f2,f1 | | ifthenelse |

(e) Alternation of Abstraction and Invention steps

Figure 1: 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. Figures (c) and (d) show higher-order and first-order target theories respectively.

The paper is organised as follows. Section 2 discusses related work. Section 3 describes the theoretical framework for the augmented form of MIL involving Abstraction and

Invention, together with a sample complexity result for the new representation. Section 4 describes Metagol$_{AI}$, including changes to the meta-interpretive learner required to support Abstraction and Invention. Section 5 details three experiments in which predictive accuracies and learning times for Metagol$_{AI}$ are compared with and without higher-order BK. In each case, a substantial increase in predictive accuracy is achieved when the higher-order BK is included, in accordance with the sample complexity result from Section 3. Finally, Section 6 summarises the outcomes and discusses further work.

## 2 Related work

Interest in IP has grown recently, partially due to successful applications in real-world problems, such as end-user programming [Gulwani, 2014a] and computer education [Gulwani, 2014b]. IP approaches can be classified as either task specific or general-purpose. Task specific approaches focus on learning programs for a specific domain and are often restricted to specific data types, such as numbers [Singh and Gulwani, 2012] and strings [Gulwani, 2011; Wu and Knoblock, 2015]. By contrast, the MIL framework is general-purpose, and has been used in a variety of problems including grammar induction [Muggleton *et al.*, 2014b], string transformations [Lin *et al.*, 2014], and extracting information from markup files [Cropper *et al.*, 2015].

MagicHaskeller [Katayama, 2008] is a general-purpose IP system which learns Haskell functions by selecting and instantiating higher-order functions from a pre-defined vocabulary. In contrast to MagicHaskeller, MIL supports predicate invention and learning explicitly recursive programs. Igor2 [Kitzelmann, 2007] 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. Esher [Albarghouthi *et al.*, 2013] learns recursive programs but needs to query an oracle each time a recursive call is encountered to ask for examples. The *L2* system [Feser *et al.*, 2015] synthesises recursive functional algorithms, but the hypotheses learned by *L2* are not directly executable. By contrast, Metagol$_{AI}$ learns Prolog programs.

Section 5 includes experiments in learning *robot strategies* [Cropper and Muggleton, 2015a]. Various machine learning approaches support the construction of strategies, including the SOAR architecture [Laird, 2008], reinforcement learning [Sutton and Barto, 1998], and action learning in inductive logic programming (ILP) [Moyle and Muggleton, 1997; Otero, 2005]. This work differs from most of these approaches in that the Metagol$_{AI}$ learns human-readable Prolog programs.

Early work in ILP [Flener and Yılmaz, 1999] considered using schema to specify the overall form of recursive programs to be learned. By contrast, the use of abstraction described in this paper involves higher-order definitions which treat predicate symbols as first-class citizens. This approach supports a form of abstraction which goes beyond typical first-order predicate invention [Saitta and Zucker, 2013] 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.

Lloyd [Lloyd, 2003] advocates using higher-order logic in the learning process, though the approach was more strongly allied to learning functional programs, and did not support predicate invention.

until(S1,S2,Cond,Do) ← Cond(S1)
until(S1,S2,Cond,Do) ← not(Cond(S1)), Do(S1,S2)
(a) **Higher-order definition**

f(A,B) ← until(A,B,at_end,f3)
(b) **Abstraction**

f3(A,B) ← f2(A,C),move_right(C,B)
(c) **Invention**

Figure 2: Higher-order definition with related Abstraction and Invention

## 3 Theoretical framework

The sets of constants, predicate symbols and first and second-order variables are denoted $\mathcal{C}, \mathcal{P}, \mathcal{V}_1$ and $\mathcal{V}_2$. Elements of $\mathcal{V}_1$ and $\mathcal{V}_2$ can bind to elements of $\mathcal{C}$ and $\mathcal{P}$ respectively.

### 3.1 Higher-order definitions

**Definition 1 (Higher-order definite clause)** *A higher-order definite clause is a well-formed formulae* $\forall\tau P(s_1,..,s_m) \leftarrow ..,Q_i(t_1,..,t_n),..$ *where* $\tau \subseteq \mathcal{V}_1 \cup \mathcal{V}_2$ *and* $P,Q_i,s_j,t_k \in \mathcal{P} \cup \mathcal{C} \cup \mathcal{V}_1 \cup \mathcal{V}_2$.

**Definition 2 (Higher-order definite definition)** *A higher-order definition is a set of higher-order clauses which all have the form* $\forall\tau\ p(s_1,..,s_m) \leftarrow ..$ *where* $\tau \subseteq \mathcal{V}_1 \cup \mathcal{V}_2$ *and* $p \in \mathcal{P}$.

The clauses in Figure 2a comprise a higher-order definition.

### 3.2 Abstractions and inventions

**Definition 3 (Abstraction)** *An abstraction is a higher-order definite clause having the form* $\forall\tau\ p(s_1,..,s_m) \leftarrow q(v_1,..,v_n,r_1,..,r_o),..$ *where* $\tau \subseteq \mathcal{V}_1 \cup \mathcal{V}_2$ *and* $p,q,r_1,..,r_o \in \mathcal{P}$ *and* $v_1,..,v_n \in \mathcal{V}_1$.

Within Computer Science *code abstraction* [Cardelli and Wegner, 1985] involves hiding complex code to provide a simplified interface for users to select key details. In this paper Abstractions contain one atom in the body which references a higher-order predicate, as shown in Figure 2b. The second-order arguments of *until* are grounded to predicate symbols.

**Definition 4 (Invention)** *In the case background knowledge* $B$ *is extended to* $B \cup H$*, where* $H$ *is a set of higher-order definite definitions, we call predicate* $p$ *an Invention iff* $p$ *is defined in* $H$ *but not in* $B$.

Within this paper Abstractions are used by a meta-interpreter to generate Inventions (Figure 2c).

### 3.3 Meta-Interpretive Learning

Given background knowledge $B$ and examples $E$ the aim of a MIL system is to learn a hypothesis $H$ such that $B,H \models E$, where $B = B_p \cup M$, $B_p$ is a set of compiled Prolog definitions and $M$ is a set of *metarules* (see Figure 3). MIL [Muggleton *et al.*, 2014b; 2015; Cropper and Muggleton, 2015b; Muggleton

*et al.*, 2014a] is a form of ILP based on an adapted Prolog meta-interpreter. A standard Prolog meta-interpreter proves goals by repeatedly fetching first-order clauses whose heads unify with the goal. By contrast, a MIL learner proves goals by fetching higher-order metarules (Figure 3) whose heads unify with the goal. The resulting meta-substitutions are saved, allowing them to be used as background knowledge by substituting them into corresponding metarules.

| Name | Metarule |
|---|---|
| Curry3 | $P(x,y) \leftarrow Q(x,y,R)$ |
| Curry4 | $P(x,y) \leftarrow Q(x,y,R,S)$ |
| Precon | $P(x,y) \leftarrow Q(x), R(x,y)$ |
| Postcon | $P(x,y) \leftarrow Q(x,y), R(y)$ |
| Chain | $P(x,y) \leftarrow Q(x,z), R(z,y)$ |
| Tailrec | $P(x,y) \leftarrow Q(x,z), R(z,y)$ |

Figure 3: Example metarules. Uppercase letters $P, Q, R, S$ denote existentially quantified variables. Lowercase letters $x$, $y$, and $z$ denote universally quantified variables.

### 3.4 Abstracted Meta-Interpretive Learning

We extend the MIL framework by assuming the background knowledge $B = B_p \cup B_i \cup M$, where $B_p$ consists of compiled Prolog code (compiled BK), $B_i$ consists of higher-order definitions (interpreted BK), and $M$ is a set of metarules. The existence of $B_p$ supports efficient execution of background knowledge, but makes the substitution of meta-variables inaccessible to the meta-interpreter for inventing new predicates. By contrast, the existence of $B_i$ allows the meta-interpreter to efficiently interleave Abstraction and Invention.

### 3.5 Language classes, expressivity and complexity

Metarules limit the language class for the hypothesis space. For instance, the Chain rule in Figure 3 restricts clauses to be definite with two body atoms and a predicate arity of two. This corresponds to the language class $H^2_2$. In [Lin *et al.*, 2014] it was shown the number of $H^2_2$ programs expressible with $n$ clauses is $O(|M|^n p^{3n})$. The result below updates this bound for the abstracted MIL framework.

**Lemma 1 (Number of abstracted $H^2_2$ programs of size $n$.)** *Given $p$ predicate symbols, $|M|$ metarules, and abstractions each with at most $k \geq 1$ second-order variables the number of $H^2_2$ programs expressible with $n$ clauses is $O(|M|^n p^{(2+k)n})$.*

**Proof.** *Since each abstraction has at most $k \geq 1$ second-order variables the number of clauses $S_p$ which can be constructed from an $H^2_2$ metarule given $p$ predicate symbols is at most $max(p^3, p^{2+k}) = p^{2+k}$. The set of such clauses $S_{m,p}$ has cardinality at most $|M|p^{2+k}$. It follows the number of logic programs constructed from a selection of $n$ rules chosen from $S_{|M|,p}$ is at most $\binom{|M|p^{2+k}}{n} \leq (|M|p^{2+k})^n = O(|M|^n p^{(2+k)n})$.*

We use this result to develop sample complexity results for unabstracted versus abstracted MIL.

**Theorem 1 (Sample complexity of unabstracted MIL)** *Unabstracted MIL has a polynomial sample complexity of $m \geq \frac{\boldsymbol{n}\ ln|M| + \boldsymbol{p}\ ln(3n) + ln\frac{1}{\delta}}{\epsilon}$.*

*Proof. According to the Blumer bound [Blumer* et al.*, 1989] the error of consistent hypotheses is bounded by $\epsilon$ with probability at least $(1-\delta)$ once $m \geq \frac{ln|H| + ln\frac{1}{\delta} + ln(c)}{\epsilon}$, where $|H|$ is the size of the hypothesis space. From [Lin* et al.*, 2014] $|H| = c(|M|^n p^{3n} + d)$ where $c, d$ are constants. Applying logs and substiting gives $m \geq \frac{\boldsymbol{n}\ ln|M| + \boldsymbol{p}\ ln(3n) + ln\frac{1}{\delta}}{\epsilon}$.*

**Theorem 2 (Sample complexity of abstracted MIL)** *Abstracted MIL has a polynomial sample complexity of $m \geq \frac{\boldsymbol{n}\ ln|M| + \boldsymbol{p}\ ln((2+k)n) + ln\frac{1}{\delta}}{\epsilon}$.*

*Proof. Analogous to Theorem 1.*

We now consider the ratio of these bounds in the case $n \gg p$.

**Proposition 1 (Ratio of unabstracted and abstracted bounds)** *Given $m, m_A$ are the bounds on the number of training examples required to achieve error less than $\epsilon$ with probability at least $1-\delta$ and $n, n_A$ are the numbers of clauses in the minimum expression of the target theories in these cases then the ratio $m : m_A$ approaches $n : n_A$ in the case $n \gg p$.*

*Proof. Since $n \gg p$ it follows $m : m_A \approx (n\, ln|M| : n_A ln|M|) = n : n_A$.*

Proposition 1 indicates abstraction in MIL reduces sample complexity proportional to the number of clauses required to express abstracted hypotheses. For instance, in Figure 1 the use of *until* and *ifthenelse* reduces the hypothesis size by one clause each. Thus the minimal hypothesis reduces from six clauses to four leading to a sample complexity reduction of $3 : 2$. Figure 4 tabulates higher-order predicates with corresponding clause reductions.

| HO predicate | Reduction |
|---|---|
| until | 1 |
| ifthenelse | 1 |
| map | 1 |
| filter | 2 |

Figure 4: Reductions in the number of clauses when using higher-order predicates

## 4 Metagol$_{AI}$

Metagol$_{AI}$ extends Metagol[1], an existing MIL implementation, to support Abstractions and Invention by learning with interpreted BK. Figure 5 shows the implementation of Metagol$_{AI}$ as a generalised meta-interpreter [Muggleton *et al.*, 2015], similar in form to a standard Prolog meta-interpreter.

**Background knowledge** The key difference between Metagol$_{AI}$ and Metagol is the introduction of the second *prove_aux* clause in the meta-interpreter, denoted in boldface. This clause allows Metagol$_{AI}$ to prove a goal by fetching a clause from the interpreted BK (such as *map*) whose head unifies with a given goal. The distinction between compiled BK and interpreted BK is that whereas a clause from the compiled BK is proved deductively by calling Prolog, a clause from the interpreted BK is proved through meta-interpretation. This

[1] https://github.com/metagol/metagol

```
prove([],H,H).
prove([Atom|Atoms],H1,H2):-
   prove_aux(Atom,H1,H3),
   prove(Atoms,H3,H2).
prove_aux(Atom,H,H):-
   call(Atom).
prove_aux(Atom,H1,H2):-
   background((Atom:-Body)),
   prove(Body,H1,H2).
prove_aux(Atom,H1,H2):-
   member(sub(Name,Subs),H1),
   metarule(Name,Subs,(Atom :- Body)),
   prove(Body,H1,H2),
prove_aux(Atom,H1,H2):-
   metarule(Name,Subs,(Atom :- Body)),
   new_metasub(H1,sub(Name,Subs)),
   abduce(H1,H3,sub(Name,Subs)),
   prove(Body,H3,H2).
```

Figure 5: Prolog code for the Metagol$_{AI}$ meta-interpreter. The clause denoted in boldface is used to fetch higher-order clauses from the interpreted BK.

approach allows for predicate invention to be driven by the proof of conditions (as in *filter*) and functions (as in *map*). Interpreted BK is different to metarules because the clauses are all universally quantified. By contrast, metarules contain existentially quantified variables whose meta-substitutions form the hypothesised program. Figure 6 shows examples of the three forms of BK used by Metagol$_{AI}$.

**Compiled BK**

```
head([H|_],H).
tail([_|T],T).
move_forward(X/Y1),X/Y2):-Y2 is Y1+1.
```

**Interpreted BK**

```
background(([map,[],[],F]:- [])).
background(([map,[A|As],[B|Bs],F]:-
   [[F,A,B],[map,As,Bs,F]])).
```

**Metarules**

```
metarule([P,Q,R],([P,A,B]:-[[Q,A],[R,A,B]])).
metarule([P,Q,R],([P,A,B]:-[[Q,A,B,R]])).
metarule([P,Q,R],([P,A,B]:-[[Q,A,C],[R,C,B]])).
```

Figure 6: Examples of the three forms of BK used by Metagol$_{AI}$

**Algorithm** Metagol$_{AI}$ first tries to prove a goal deductively using compiled BK by delegating the proof to Prolog (*call(Atom)*). Failing this, Metagol$_{AI}$ tries to unify the goal with the head of a clause in the interpreted BK (*background((Atom:-Body))*) and tries to prove the body goals of the clause. Failing this, Metagol$_{AI}$ tries to unify the goal with the head of a metarule (*metarule(Name,Subs,(Atom :- Body))*) and to bind the existentially quantified variables in a metarule to symbols in the signature. Metagol$_{AI}$ saves the resulting *meta-substitutions* (*Subs*) and tries to prove the body goals of the metarule. After proving all goals, a Prolog program is formed by projecting the meta-substitutions onto their corresponding metarules. Negation as failure [Clark, 1987] is used to negate predicates in the compiled BK. Negation of invented predicates is unsupported and is left for future work.

## 5 Experiments

This section describes three experiments[2] which compare abstracted MIL with unabstracted MIL, i.e. learning with and without interpreted higher-order BK. To do this, we compare Metagol$_{AI}$ (which supports interpreted BK) with Metagol (which does not support interpreted BK). Accordingly, we investigate the following null hypotheses:

**Null hypothesis 1** Metagol$_{AI}$ cannot learn more accurate programs than Metagol

**Null hypothesis 2** Metagol$_{AI}$ cannot learn programs quicker than Metagol

**Common materials** We provide Metagol$_{AI}$ and Metagol with the same BK. The compiled BK varies in each experiment. The interpreted BK contains the following definitions: *map/3*,*reduce/3*,*reduceback/3*, *until/4*, and *ifthenelse/5*. The metarules used are shown in Figure 3. Therefore, the only variable in the experiments is the learning system. The only difference between the two systems is the additional clause used by Metagol$_{AI}$, described in Section 4.

**Common methods** We train using $m$ randomly chosen positive examples for each $m$ in the set $\{1,2,3,4,5\}$. We test using 40 examples, half positive and half negative, so the default accuracy is 50%. We average predictive accuracies and learning times over 20 trials. For each learning task, we enforce a 10-minute timeout.

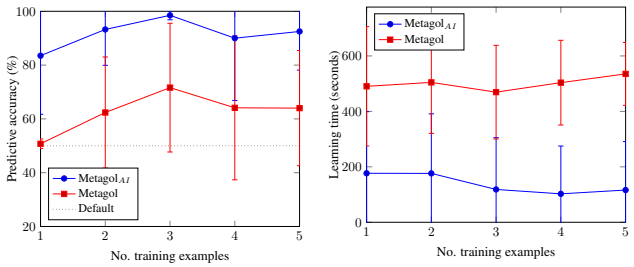
### 5.1 Robot waiter

This experiment revisits the waiter example in Figure 1, in which a robot waiter is learning to serve drinks.

**Materials** The state is a list of facts. In the initial state, the robot starts at position 0; there are $d$ cups facing down at positions $1, \ldots, d$; and for each cup there is a preference for tea or coffee. In the final state, the robot is at position $d+1$; all the cups are facing up; and each cup is filled with the preferred drink. We generate positive examples as follows. For the initial state, we select a random integer $d$ from the interval $[1, 20]$ as the number of cups. 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 $k$ drinks. The robot can perform the following fluents and actions (details omitted for brevity) defined as compiled BK: *at_end/1*, *wants_tea/1*, *wants_coffee/1*, *move_left/2*, *move_right/2*, *turn_cup_over/2*, *pour_tea/2*, and *pour_coffee/2*.

[2]Experimental data are available at http://ilp.doc.ic.ac.uk/ijcai16-metagolai

**Results** Figure 7a shows that Metagol$_{AI}$ learns more accurate programs than Metagol, refuting null hypothesis 1. Figure 7b shows that Metagol$_{AI}$ learns programs quicker than Metagol, refuting null hypothesis 2. Figure 1 shows example programs learned by Metagol (c) and Metagol$_{AI}$ (d)). Although both programs are general and can handle any number of guests and any assignment of drink preferences, program (b) is smaller because it uses the higher-order abstractions *until* and *ifthenelse*. This compactness affects predicate accuracies because whereas Metagol$_{AI}$ can find solutions in the allocated time, Metagol stuggles because the solutions are too long.

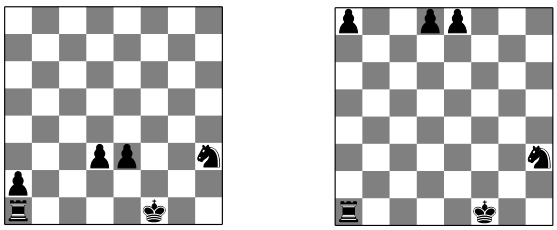



(a) Predictive accuracies (b) Learning times

Figure 7: Robot waiter experiment results

## 5.2 Chess strategy

Programming robust chess playing strategies is an exceptionally difficult task for human programmers [Bratko and Michie, 1980]. Consider the concept of maintaining a wall of pawns to support promotion [Harris, 1988]. 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. Having constructed such a program one might consider using negative examples involving interposition of white pieces to deal with exceptional behaviour. Figure 8 shows such an example, where in the initial state 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.



(a) Initial state (b) Final state

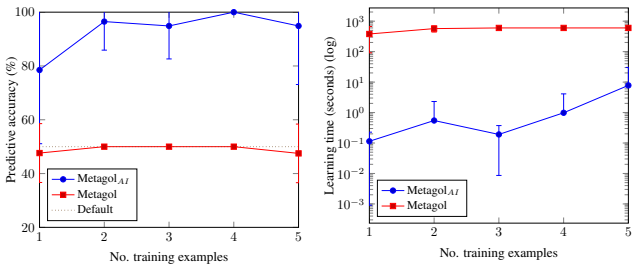
Figure 8: Chess initial/final state example

**Materials** The state is a list of pieces, where a piece is denoted as a triple of the form *(Type,Id,X/Y)*, where *Type* is the type (king=k, pawn=p, etc.), *Id* is a unique identifier, and *X/Y* is the position. We generate positive examples as follows. For the initial state, we select a random subset of $n$ pieces from the interval $[2, 16]$ and randomly place them on the board. 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 9.

```
at_rank8((_,_,_/8)).
is_pawn((p,_,_)).
not_pawn(X):-not(is_pawn(X)).
empty([]).
move_forward((Type,Id,X/Y1),(Type,Id,X/Y2)):-
  Y1 < 8,Y2 is Y1+1.
move_forward(A,B,Id):-
  append(Prefix,[(Type,Id,X/Y1)|Suffix],A),
  Y1 < 8,Y2 is Y1+1,
  append(Prefix,[(Type,Id,X/Y2)|Suffix],B).
```

Figure 9: Compiled BK used in the chess experiment

**Results** Figure 10a shows that Metagol$_{AI}$ learns programs approaching 100% accuracy after two examples. By contrast, Metagol learns programs with around default accuracy. This result refutes null hypothesis 1. The semi-log plot in Figure 10b shows that Metagol$_{AI}$ learns programs quicker than Metagol, refuting null hypothesis 2. We can explain these results by looking at the sample programs learned in Figure 11. Metagol$_{AI}$ (b) learns a small higher-order program using the abstractions *map* and *until*. In this program, the *map* operation decomposes the problem into smaller sub-problems of finding how to move a single piece to rank 8. These sub-goals are solved by the *chess1* predicate. By contrast, Metagol (a) learns a larger recursive and more specific first-order program.




(a) Predictive accuracies (b) Learning times

Figure 10: Chess experiment results

## 5.3 Drop lasts

In this experiment, the goal is to learn a program *droplasts* which drops the last element from each sublist of a given list, a problem frequently used to evaluate IP systems [Feser *et al.*, 2015; Kitzelmann, 2007]. Figure 12 shows input/output examples for this problem.

**Materials** We generate training examples as follows. To form the input, we select a random integer $i$ from the interval

```
chess(A,B):-chess2(A,C),chess2(C,B).
chess2(A,B):-chess1(A,C),chess1(C,B).
chess1(A,B):-move_forward(A,B,p3).
chess2(A,B):-move_forward(A,B,p5).
```

(a) First-order program

```
chess(A,B):-map(A,B,chess1).
chess1(A,A):-not_pawn(A).
chess1(A,B):-until(A,B,at_rank8,move_forward).
```

(b) Higher-order program

Figure 11: Examples of programs learned by Metagol (a) and Metagol$_{AI}$ (b) for the chess experiment

| Input | Output |
|---|---|
| [[i,j,c,a,i],[2,0,1,6]] | [[i,j,c,a],[2,0,1]] |
| [[1,2,3,4,5],[1,2,3,4,5]] | [[1,2,3,4],[1,2,3,4]] |
| [[1,2],[1,2,3],[1,2,3,4],[1,2,3,4,5]] | [[1],[1,2],[1,2,3],[1,2,3,4]] |

Figure 12: Input/output examples for the *droplasts* experiment

$[2, 20]$ as the number of sublists. For each sublist $i$, we select a random integer $k$ from the interval $[1, 100]$ and populate sublist $i$ with $k$ random integers. To form the output, we wrote a Prolog program to drop the last element from each sublist. We use the compiled BK shown in Figure 13.

```
head([H|_],H).
tail([_|T],T).
concat([H|T],B,C):-append([H|T],[B],C).
concat(A,B,C):-append([A],[B],C).
```

Figure 13: Compiled BK used in the *droplasts* experiment

**Results** Metagol$_{AI}$ achieved 100% accuracy after two examples (plot omitted for brevity). Figure 14 shows the program learned by Metagol$_{AI}$. This program contains a number of noteworthy sub-programs. The invented predicate *droplasts1* reverses a given list. The invented predicate *droplasts3* drops the last element from a single list by (1) reversing the list by calling *droplasts1*, (2) dropping the head from the reversed list, and (3) reversing the shortened list back to the original order by again calling *droplasts1*. Finally, *droplasts* maps over the input list and applies *droplasts3* to each sublist to form the output list. This program highlights invention through the repeated calls to *droplasts1* and abstraction through the higher-order functions. By contrast, Metagol was unable to learn any solution for this problem because the corresponding first-order program is too long and thus the search is intractable.

**Further discussion** To further demonstrate invention and abstraction, consider learning a program *d_droplasts* which extends the *droplasts* problem so that, in addition to dropping the last element from each sublist, the whole last sublist is also dropped. For this problem, given two examples under the same conditions as in Section 5.3, Metagol$_{AI}$ learns the program in Figure 15. The learned program is similar to the *droplasts* program, but it makes an additional final call to the

```
droplasts(A,B):-map(A,B,droplasts3).
droplasts3(A,B):-droplasts2(A,C),droplasts1(C,B).
droplasts2(A,B):-droplasts1(A,C),tail(C,B).
droplasts1(A,B):-reduceback(A,B,concat).
```

Figure 14: Program learned by Metagol$_{AI}$ for the *droplasts* experiment

invented predicate *d_droplasts3*, which is used twice in the program as both a higher-order argument in *d_droplasts4* and as a first-order predicate in *d_droplasts*.

```
d_droplasts(A,B):-d_droplasts4(A,C),d_droplasts3(C,B).
d_droplasts4(A,B):-map(A,B,d_droplasts3).
d_droplasts3(A,B):-d_droplasts2(A,C),d_droplasts1(C,B).
d_droplasts2(A,B):-d_droplasts1(A,C),tail(C,B).
d_droplasts1(A,B):-reduceback(A,B,concat).
```

Figure 15: Program learned by Metagol$_{AI}$ for the *d_droplasts* problem

## 6 Conclusions and further work

We have introduced Metagol$_{AI}$ which extends the MIL framework to support learning compact programs by using higher-order Abstractions. Our sample complexity results indicate that the consequent reduction in the number of clauses in the minimal representation of the target leads to a reduced hypothesis space which in turn leads to reductions in the size of the sample complexity required to learn the target theory. These complexity results are consistent with our experiments which indicate increased predictive accuracy and decreased learning time for abstracted MIL compared with unabstracted MIL.

**Future work** The experiments in this paper were largely related to the use of functional constructs, such as *map* and *reduceback*, within logic programs. However, we would like to investigate the use of relational constructs. For instance, consider the following higher-order definition of a closure.

closure(P,X,Y) ← P(X,Y).
closure(P,X,Y) ← P(X,Z), closure(P,Z,Y).

This definition could be used to learn compact abstractions of relations such as the following.

ancestor(X,Y) ← closure(parent,X,Y).
lessthan(X,Y) ← closure(increment,X,Y).
subterm(X,Y) ← closure(headortail,X,Y).

Moreover, the issue of how metarules might themselves be learned could be treated in a similar fashion using higher-order programs such as the following.

chain(P,Q,R,X,Y) ← Q(X,Z), R(Z,Y).
inverse(P,Q,X,Y) ← Q(Y,X).

In summary we believe that the use of abstractions in machine learning provides an important new approach to the use of powerful programming constructs within IP. We believe that such approaches could have wide application in AI domains such as planning, vision, and natural language processing.

## Acknowledgements

The first author thanks the BBSRC and Syngenta for his PhD case studentship. The second author acknowledges support from his Royal Academy of Engineering/Syngenta Research Chair at the Department of Computing at Imperial College London.

## References

[Albarghouthi *et al.*, 2013] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *Computer Aided Verification*, pages 934–950. Springer, 2013.

[Blumer *et al.*, 1989] A. Blumer, A. Ehrenfeucht, D. Haussler, and M.K. Warmuth. Learnability and the Vapnik-Chervonenkis dimension. *Journal of the ACM*, 36(4):929–965, 1989.

[Bratko and Michie, 1980] Ivan Bratko and Donald Michie. a representation for pattern-knowledge in chess endgames. *Advances in Computer Chess*, 2:31–56, 1980.

[Cardelli and Wegner, 1985] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–523, 1985.

[Clark, 1987] K.L. Clark. Negation as failure. In M. L. Ginsberg, editor, *Readings in Nonmonotonic Reasoning*, pages 311–325. Kaufmann, Los Altos, CA, 1987.

[Cropper and Muggleton, 2015a] A. Cropper and S.H. Muggleton. Learning efficient logical robot strategies involving composable objects. In *Proceedings of the 24th International Joint Conference Artificial Intelligence (IJCAI 2015)*, pages 3423–3429. IJCAI, 2015.

[Cropper and Muggleton, 2015b] A. Cropper and S.H. Muggleton. Logical minimisation of meta-rules within meta-interpretive learning. In *Proceedings of the 24th International Conference on Inductive Logic Programming*, pages 65–78. Springer-Verlag, 2015. LNAI 9046.

[Cropper *et al.*, 2015] A. Cropper, A. Tamaddoni-Nezhad, and S.H. Muggleton. Meta-interpretive learning of data transformation programs. In *Proceedings of the 24th International Conference on Inductive Logic Programming*, 2015. To appear.

[Feser *et al.*, 2015] 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.

[Flener and Yıilmaz, 1999] Pierre Flener and Serap Yıilmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. *The Journal of Logic Programming*, 41(2):141–195, 1999.

[Gulwani *et al.*, 2015] S. Gulwani, J. Hernandez-Orallo, E. Kitzelmann, S.H. Muggleton, U. Schmid, and B. Zorn. Inductive programming meets the real world. *Communications of the ACM*, 58(11):90–99, 2015.

[Gulwani, 2011] 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.

[Gulwani, 2014a] Sumit Gulwani. Applications of program synthesis to end-user programming and intelligent tutoring systems. In *Genetic and Evolutionary Computation Conference, GECCO '14, Vancouver, BC, Canada, July 12-16, 2014, Companion Material Proceedings*, pages 5–6, 2014.

[Gulwani, 2014b] Sumit Gulwani. Example-based learning in computer-aided STEM education. *Commun. ACM*, 57(8):70–80, 2014.

[Harris, 1988] 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.

[Katayama, 2008] 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, 2008. Proceedings*, pages 199–210, 2008.

[Kitzelmann, 2007] Emanuel Kitzelmann. Data-driven induction of recursive functions from input/output-examples. In *WORKSHOP ON APPROACHES AND APPLICATIONS OF INDUCTIVE PROGRAMMING*, page 15, 2007.

[Laird, 2008] J. E. Laird. Extending the soar cognitive architecture. *Frontiers in Artificial Intelligence and Applications*, pages 224–235, 2008.

[Lin *et al.*, 2014] D. Lin, E. Dechter, K. Ellis, J.B. Tenenbaum, and S.H. Muggleton. Bias reformulation for one-shot function induction. In *Proceedings of the 23rd European Conference on Artificial Intelligence (ECAI 2014)*, pages 525–530. IOS Press, 2014.

[Lloyd, 2003] J.W. Lloyd. *Logic for Learning*. Springer, Berlin, 2003.

[Moyle and Muggleton, 1997] S. Moyle and S.H. Muggleton. Learning programs in the event calculus. In *Proceedings of the Seventh Inductive Logic Programming Workshop (ILP97)*, LNAI 1297, pages 205–212, Berlin, 1997. Springer-Verlag.

[Muggleton *et al.*, 2014a] S.H. Muggleton, D. Lin, J. Chen, and A. Tamaddoni-Nezhad. Metabayes: Bayesian meta-interpretative learning using higher-order stochastic refinement. In *Proceedings of the 23rd International Conference on Inductive Logic Programming (ILP 2013)*, pages 1–17. Springer-Verlag, 2014. LNAI 8812.

[Muggleton *et al.*, 2014b] S.H. Muggleton, D. Lin, N. Pahlavi, and A. Tamaddoni-Nezhad. Meta-interpretive learning: application to grammatical inference. *Machine Learning*, 94:25–49, 2014.

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

[Otero, 2005] R. Otero. Induction of the indirect effects of actions by monotonic methods. In *Proceedings of the Fifteenth International Conference on Inductive Logic Programming (ILP05)*, volume 3625, pages 279–294. Springer, 2005.

[Saitta and Zucker, 2013] Lorenza Saitta and Jean-Daniel Zucker. *Abstraction in artificial intelligence and complex systems*. Springer, 2013.

[Singh and Gulwani, 2012] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 634–651, 2012.

[Sutton and Barto, 1998] R.S. Sutton and A.G. Barto. *Reinforcement learning: An introduction*, volume 1. Cambridge Univ Press, 1998.

[Wu and Knoblock, 2015] Bo Wu and Craig A. Knoblock. An iterative approach to synthesize data transformation programs. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1726–1732, 2015.