RESEARCH INSTITUTE FOR SYMBOLIC COMPUTATION

JOHANNES KEPLER UNIVERSITY LINZ

Learning Higher-Order Programs without Meta-Interpretive Learning

Stanislav Purgal and David Cerna and Cezary Kalisyk

December 2021

RISC Report Series No. 21-22

ISSN: 2791-4267 (online)

Available at `https://doi.org/10.35011/risc.21-22`

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Str. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Learning Higher-Order Programs without Meta-Interpretive Learning*

Stanisław J. Purgał[1† ‡], **David M. Cerna**[2,3§], **Cezary Kaliszyk**[1 ‡]
[1]University of Innsbruck, Innsbruck, Austria
[2]Czech Academy of Sciences Institute of Computer Science (CAS ICS), Prague, Czechia
[3]Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria
stanislaw.purgal@uibk.ac.at, dcerna@{cs.cas.cz, risc.jku.at}, cezary.kaliszyk@uibk.ac.at

Abstract

Learning complex programs through *inductive logic programming* (ILP) remains a formidable challenge. Existing higher-order enabled ILP systems show improved accuracy and learning performance, though remain hampered by the limitations of the underlying learning mechanism. Experimental results show that our extension of the versatile *Learning From Failures* paradigm by higher-order definitions significantly improves learning performance without the burdensome human guidance required by existing systems. Furthermore, we provide a theoretical framework capturing the class of higher-order definitions handled by our extension.

1 Introduction

Inductive Logic Programming (ILP) [Mug91; NdW97] is a form of symbolic machine learning which learns a logic program from background knowledge (BK) predicates and sets of positive and negative example runs of the goal program.

Naively, learning a logic program which takes a positive integer n and returns a list of list of the form $[[1], [1, 2], \cdots, [1, \cdots, n]]$ would not come across as a formidable learning task. A logic program is easily constructed using conventional higher-order (HO) definitions.

$$\boxed{\begin{array}{r}\texttt{allSeqN}(N, L) \vdash \texttt{iterate}(succ, 0, N, A),\ \texttt{map}(p, A, L).\\ \texttt{p}(A, B) \vdash \texttt{iterate}(succ, 0, A, B).\end{array}}$$

The first `iterate` produces the list $[1, \cdots, N]$ and `map` applies a functionally equivalent `iterate` to each member of $[1, \cdots, N]$, thus producing the desired outcome. However, this seemingly innocuous function requires 25 literals spread over five clauses when written as a function-free, first-order (FO) logic program, a formidable task for most if not all existing FO ILP approaches [CDEM21].

*Authors contributed equally to the work described in this paper.
†Contact Author
‡Supported by the ERC starting grant no. 714034 SMART.
§Supported by the Linz Institute of Technology Math$_{LP}$ project (LIT- 2019-7-YOU-213) funded by the state of upper Austria.

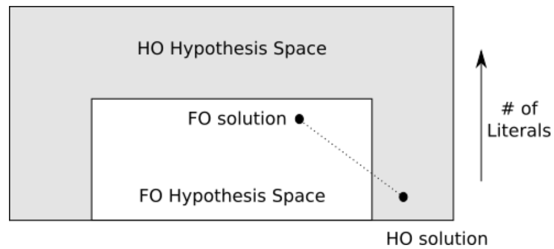

Figure 1: Inclusion of HO definitions increases the size of the search space, but can lead to the search space containing a shorter solutions.

Excessively large BK can, in many cases, lead to performance loss [Cro20; SKB03]. In contrast, adding HO definitions increases the overall size of the search space, but may result in the presence of significantly simpler solutions (see Figure 1). Enabling a learner, with a strong bias towards short solutions, with the ability to use HO definitions can result in improved performance. We developed an HO-enabled *Popper* [CM21a] (*Hopper*), a novel ILP system designed to learn optimally short solutions. Experiments show significantly better performance on hard tasks when compared with *Popper* and the best performing HO-enabled ILP system, *Metagol*$_{\text{HO}}$ [CMM20]. See Section 4.

Existing HO-enabled ILP systems are based *Meta-interpretive Learning* (MiL) [MLPT14]. The efficiency and performance of MiL-based systems is strongly dependent on significant human guidance in the form of *metarules* (a restricted form of HO horn clauses). Choosing these rules is an art in all but the simplest of cases. For example, `iterate`, being ternary, poses a challenge for some systems, and in the case of *HEXMIL*$_{\text{HO}}$ [CMM20], this definition cannot be considered as only dyadic definitions are allowed.

Limiting human participation when fine-tuning the search space is an essential step towards strong symbolic machine learning. The *Learning from Failures* (LFF) paradigm [CM21a], realized through *Popper*, prunes the search space as part of the learning process. Not only does this decrease human guidance, it also removes limitations on the structure of HO definitions allowing us to further exploit the above mentioned benefits.

Integrating HO concepts into MiL-based systems is quite seamless as HO definitions are essentially a special type of metarule. Thus, HO enabling MiL learners requires minimal change to the theoretical foundations. In the case of LFF learners, like *Popper*, the pruning mechanism influences which HO definitions may be soundly used.

We examine this issue in Section 3 and provide a construction encapsulating the accepted class of HO definitions. Succinctly, it is the class of definitions that are *monotone* with respect to *subsumption* and *entailment*; $p_1 \leq_\theta (\models)\ p_2 \Rightarrow H(p_1) \leq_\theta (\models)\ H(p_2)$ where p_1 and p_2 are logic programs, and $H(\cdot)$ is an HO definition incorporating parts of p_1 and p_2. Similar to classes considered in literature, our class excludes most cases of HO negation (see Section 3.4), though our framework opens the opportunity to invent HO predicates during learning (an important open problem), though this remains too inefficient in practice and is left to future work.

2 Related Work

The authors of [CMM20] (Section 2) provide a literature survey concerning the synthesis of Higher-Order (HO) programs and, in particular, how existing ILP systems deal with HO constructions. Below we provide a brief summary of this survey and focus on introducing the state-of-the-art systems, namely, HO extensions of *Metagol* [CM16] and *HEXMIL* [KEI18]. We also introduce Popper [CM21a], the system *Hopper* is based on. Additionally, a detailed survey of the current state of ILP research has recently been published [CDEM21].

2.1 Predicate Invention and HO Synthesis

Effective use of HO predicates is intimately connected to auxiliary Predicate Invention (PI). The following illustrates how `fold/4` can be used together with PI to provide a succinct program for reversing a list:

$$\boxed{\begin{array}{c}\texttt{reverse}(A,B) \vdash \texttt{empty}(C),\ \texttt{fold}(p,C,A,B).\\ \texttt{p}(A,B,C) \vdash \texttt{head}(C,B),\ \texttt{tail}(C,A).\end{array}}$$

Including p in the background knowledge is unintuitive. It is reasonable to expect the synthesizer to produce it. Many of the well known, non-MiL based ILP frameworks do not support predicate invention, *Foil* [Qui90], *Progol* [Mug95], *Tilde* [Blo99], and *Aleph* [Sri01] to name a few. While there has been much interest, throughout ILP's long history, concerning PI, it remained an open problem discussed in "ILP turns 20" [MRP$^+$12]. Since then, there have been a few successful approaches. Both *ILASP* [Law18] and *δILP* [EG18] can, in a restricted sense, introduce invented predicates, however neither handles infinite domains nor are suited for the task we are investigating, manipulation of trees and lists.

The best performing systems with respect to the aforementioned tasks are *Metagol* [CM16] and *HEXMIL* [KEI18]; both are based on *Meta-interpretive Learning* (MiL) [MLPT14], where PI is considered at every step of program construction. However, a strong language bias is needed for an efficient search procedure. This language bias comes in the form of *Metarules* [CM14], a restricted form of HO horn clauses.

Definition 1 ([CT20]) *A* metarule *is a second-order Horn clause of the form $A_0 \leftarrow A_1, \cdots, A_n$, where A_i is a literal $P(T_1, \cdots, T_m)$, s.t. P is either a predicate symbol or a HO variable and each T_i is either a constant or a FO variable.*

For further discussion see Section 2.2. *Popper* [CM21a], does not directly support PI, though, it is possible to enforce PI through the language bias (*Poppi* is an PI-enabled extension [CM21b]) . *Popper*'s language bias, while partially fixed, is essentially an arbitrary ASP program. The Authors of [CM21a] illustrate this by providing ASP code emulating the chain metarule[1] (see Appendix A of [CM21a]). We exploit this feature to extend *Popper*, allowing it to construct programs containing instances of HO definitions. *Hopper*, our extension, has drastically improved performance when compared with *Popper*. *Hopper* also outperforms the state-of-the-art Mil-based ILP systems extended by HO definitions. For further discussion of *Popper* see Section 2.3, and for *Hopper* see Section 3.

2.2 *Metagol* and *HEXMIL*

We briefly summarize existing HO-capable ILP systems introduced by A. Cropper *et al.* [CMM20].

Higher-order *Metagol*

In short, *Metagol* is an MiL-learner implemented using a Prolog meta-interpreter. As input, *Metagol* takes a set of predicate declarations PD of the form `body_pred(P/n)`, sets of positive E^+ and negative E^- examples , *compiled* background knowledge BK_c, and a set of metarules M. The examples provide the arity and name of the goal predicate. Initially, *Metagol* attempts to satisfy E^+ using BK_c. If this fails, then *Metagol* attempts to unify the current goal atom with a metarule from $m \in M$. At this point *Metagol* tries to prove the body metarule m. If successful, the derivation provides a Prolog program which can be tested on E^-. If the program entails some of E^-, *Metagol* backtracks and tries to find another program. Invented predicates are introduced while proving the body of a metarule when BK_c is not sufficient for construction of a program.

The difference between *Metagol* and *Metagol*$_{\mathrm{HO}}$ is the inclusion of *interpreted* background knowledge BK_{in}. For example, `map/3` as BK_{in} takes the form:

```
ibk([map,[],[],_],[]).
ibk([map,[A|As],[B|Bs],F],[[F,A,B],[map,As,Bs,F]]).
```

Metagol handles BK_{in} as it handles metarules. When used, *Metagol* attempts to prove the body of `map`, i.e. $F(A,B)$. Either F is substituted by a predicate contained in BK_c or replaced by an invented predicate that becomes the goal atom and is proven using metarules or BK_{in}.

A consequence of this approach is that substituting the goal atom by a predicate defined as BK_{in} cannot result in a derivation defining a Prolog program. Like with metarules, additional proof steps are necessary. The following program defining $\texttt{half}_{\texttt{lst}}(A,B)$, which computes the last half of a list[2], illustrates why this may be problematic:

[1] $\texttt{P}(A,B) \vdash \texttt{Q}(A,C), \texttt{R}(C,B)$.

[2] $\texttt{half}_{\texttt{lst}}([1,2],[2])$, $\texttt{half}_{\texttt{lst}}([1,2,3],[3])$, $\texttt{half}_{\texttt{fst}}([1,2],[1])$, and $\texttt{half}_{\texttt{fst}}([1,2,3],[1,2])$.

$$
\begin{aligned}
\mathtt{half}_{\mathtt{1st}}(A,B) &\vdash \mathtt{reverse}(A,C),\\
&\quad\ \mathtt{case}_{\mathtt{list}}(p_{[\,]},p_{[H|T]},C,B).\\
\mathtt{p}_{[\,]}(A) &\vdash \mathtt{empty}(A).\\
\mathtt{p}_{[H|T]}(A,B,C) &\vdash \mathtt{empty}(B),\mathtt{empty}(C).\\
\mathtt{p}_{[H|T]}(A,B,C) &\vdash \mathtt{front}(B,D)^3,\\
&\quad\ \underline{\mathtt{case}_{\mathtt{list}}(p_{[\,]},p_{[H|T]},D,E)},\\
&\quad\ \mathit{append}(E,A,C).
\end{aligned}
$$

The HO predicate $\mathtt{case}_{\mathtt{list}}(p_{[\,]},p_{[H|T]},A,B)$ calls $p_{[\,]}$ if A is empty and $p_{[H|T]}$ otherwise. Our definition of $\mathit{half}_{\mathtt{1st}}(A,B)$ cannot be found using the standard search procedure as every occurrence of $\mathtt{case}_{\mathtt{list}}$ results in a call to the meta-interpreter's proof procedure. The underlined call to $\mathtt{case}_{\mathtt{list}}$ results in PI for $\mathtt{p}_{[H|T]}$ *ad infinitum*. Similarly, the initial goal cannot be substituted unless it's explicitly specified.

As with $\mathit{half}_{\mathtt{1st}}(A,B)$, The following program defining $\mathtt{issubtree}(A,B)$, which computes whether B is a subtree of A, requires recursively calling issubtree through any.

$$
\begin{aligned}
\mathtt{issubtree}(A,B) &\vdash A = B.\\
\mathtt{issubtree}(A,B) &\vdash \mathtt{children}(A,C),\mathtt{any}(\mathtt{cond},C,B).\\
\mathtt{cond}(A,B) &\vdash \underline{\mathtt{issubtree}(A,B)}.
\end{aligned}
$$

This can be resolved using *metatypes* (see Section 4), but this is non-standard, results in a strong language bias, and does not always work. *Hopper* successfully learns these predicates without any significant drawbacks.

Negation of invented predicates (HO arguments of BK_{in} definitions), to the best of our knowledge, is not fully supported by *Metagol*$_{\mathrm{HO}}$ (See Section 4.2 of [CMM20]). *Hopper* has similar issues which are discussed in Section 3.4.

Higher-order *HEXMIL*

HEXMIL is an ASP encoding of Meta-interpretive Learning [KEI18]. Given that ASP can be quite restrictive, *HEXMIL* exploits the HEX formalism for encoding MiL. HEX allows the ASP solver to interface with external resources [ERS16]. Unlike *Metagol*, *HEXMIL* is restricted to *forward-chained metarules*:

Definition 2 *A* forward-chained *metarule is of the form:* $P(A,B)\vdash Q_1(A,C_1),Q_2(C_1,C_2),\cdots,Q_n(C_{n-1},B),R_1(D_1),\cdots,R_m(D_m)$. *where* $D_i \in \{A,C_1,\cdots,C_{n-1},B\}$.

This restricts us to Dyadic learning task. Furthermore, many useful metarules are not of this form, i.e. $P(A,B) \vdash Q(A,B),R(A,B)$. *HEXMIL*$_{\mathrm{HO}}$, incorporates HO definitions into the forward-chained structure of Definition 2.

Definition 3 *A* forward-chained *HO definition consists of clauses of the form:* $P(A,B) \vdash h_1(A,C_1,P_{1,1},\cdots,P_{1,k_1}),\cdots,h_n(C_{n-1},B,P_{n,1},\cdots,P_{n,k_n})$, *where every atom in this clause can have* $0 \le k_i$ *higher-order terms, and at least one has* $k_i \neq 0$.

For details concerning the encoding see Section 4.4 of [CMM20]. The authors of [CMM20] illustrated *HEXMIL*$_{\mathrm{HO}}$'s poor performance on list manipulation tasks, and its limitations make application to tasks of interest difficult. Thus, we focus on *Metagol*$_{\mathrm{HO}}$ in Section 4.

2.3 Popper: Learning From Failures (LFF)

The LFF paradigm together with Popper provide a novel approach to inductive logic programming, based on counterexample guided inductive synthesis (CEGIS) [SL08]. Both LFF and the system implementing it were introduced by A. Cropper and R. Morel [CM21a]. As input, *Popper* takes a set of predicate declarations PD, sets of positive E^+ and negative E^- examples, and background knowledge BK, the typical setting for *learning from entailment* ILP [Rae08].

During the **generate phase**, candidate programs are chosen from the viable hypothesis space, i.e. the space of programs which have yet to be ruled out by generated constraints. The chosen program is then tested (**test phase**) against E^+ and E^-. If only some of E^+ and/or some of E^- is entailed by the candidate hypothesis, Popper builds constraints (**constrain phase**) which further restrict the viable hypothesis space searched during the **generate phase**. When a candidate program only entails E^+, *Popper* terminates.

Popper searches through a finite hypothesis space, parameterized by features of the language bias (i.e. number of body predicates, variables, etc.). Importantly, if an optimal solution is present in this parameterized hypothesis space, Popper will find it (Theorem 11 [CM21a]). Optimal is defined as the solution containing the fewest literals [CM21a].

An essential aspect of this *generate, test, constrain* loop is the choice of constraints. Depending on how a candidate program performs in the **test phase**, Popper introduces constraints pruning specializations and/or generalizations of the candidate program. Specialization/generalization is defined via Θ-subsumption [Plo70; Rey70]. *Popper* may also introduce **Elimination** Constraints pruning *seperable*[4] sets of clauses. Details concerning the benefits of this approach are presented in [CM21a]. Essentially, *Popper* refines the hypothesis space, not the program [Sri01; Mug95; QCJ93].

In addition to constraints introduced during the search, like the majority of ILP systems, *Popper* incorporates a form of *language bias* [NCWSC97], that is predefined syntactic and/or semantic restrictions of the hypothesis space. *Popper* minimally requires *predicate declarations*, i.e. whether a predicate can be used in the head or body of a clause, and with what arities the predicate may appear. Popper accepts *mode declaration*-like hypothesis constraints [Mug95] which declare, for each argument of a given predicate, the type and direction. Additional hypothesis constraints can be formulated as ASP programs (mentioned in Section 2.1).

Popper implements the *generate, test, constrain* loop using a *multi-shot solving* framework [GKKS19] and an encoding of both definite logic programs and constraints within the *ASP* [Lif19] paradigm. The language bias together with generated constraints are encoded as an ASP program. The ASP

[3] $\mathtt{front}(A,B) \vdash \mathtt{reverse}(A,C),\mathtt{tail}(C,D),\mathtt{reverse}(D,B)$.

[4] No head literal of a clause in the set occurs as a body literal of a clause in the set.

solver is run on this program and the resulting model (if one exists) is an encoding of a candidate program.

3 Theoretical Framework

We provide a brief overview of logic programming. Our exposition is far from comprehensive. We refer the interested reader to a more detailed source [Llo87].

3.1 Preliminaries

Let $\mathcal{P}$ be a countable set of *predicate symbols* (denoted by $p, q, r, p_1, q_1, \cdots$), $\mathcal{V}_f$ be a countable set of *first-order (FO) variables* (denoted by $A, B, C, \cdots$) , and $\mathcal{V}_h$ be a countable set of *HO variables* (denoted by $P, Q, R, \cdots$). Let $\mathcal{T}$ denote the set of FO terms constructed from a finite set of function symbols and $\mathcal{V}_f$ (denoted by $s, t, s_1, t_1, \cdots$).

An *atom* is of the form $p(T_1, \cdots, T_m, t_1, \cdots, t_n)$. Let us denote this atom by a, then $sy(a) = p$ is the *symbol of the atom*, $ag_h(a) = \{T_1, \cdots, T_m\}$ are its *HO-arguments*, and $ag_f(a) = \{t_1, \cdots, t_n\}$ are its *FO-arguments*. When $ag_h(a) = \emptyset$ and $sy(a) \in \mathcal{P}$ we refer to a as *FO*, when $ag_h(a) \subset \mathcal{P}$ and $sy(a) \in \mathcal{P}$ we refer to a as *HO-ground*, otherwise it is *HO*. A *literal* is either an atom or its negation. A literal is *HO* if the atom it contains is *HO*. [5]

A *clause* is a set of literals. A *Horn* clause contains at most one positive literal while a definite clause must have exactly one positive literal. The atom of the positive literal of a definite clause c is referred to as the *head* of c (denoted by $hd(c)$), while the set of atoms of negated literals is referred to as the *body* (denoted by $bd(c)$). A *function-free definite (f.f.d) clause* only contains variables as FO arguments. We refer to a finite set of clauses as a *theory*. A theory is considered *FO* if all atoms are *FO*. Replacing variables $P_1, \cdots, P_n, A_1, \cdots, A_m$ by predicate symbols $p_1, \cdots, p_n$ and terms $t_1 \cdots t_m$ is a *substitution* $(\theta, \sigma, \cdots)$ denoted by $\theta = \{P_1 \mapsto p_1, \cdots, P_n \mapsto p_n, A_1 \mapsto t_1, \cdots, A_m \mapsto t_m\}$. A substitution θ *unifies* two atoms when $a\theta = b\theta$.

3.2 Interpretable Theories and Groundings

Our hypothesis space consists a particular type of theory.

Definition 4 *A clause c is* proper[6] *if $ag_h(hd(c))$ are pairwise distinct, $ag_h(hd(c)) \subset \mathcal{V}_h$, and $\forall a \in bd(c)$,*

a) if $sy(a) \in \mathcal{V}_h$, then $sy(a) \in ag_h(hd(c))$, and

b) if $p \in ag_h(a)$ and $p \in \mathcal{V}_h$, then $p \in ag_h(hd(c))$.

A finite set of proper clauses d with the same head (denoted $hd(d)$) is referred to as a *HO definition*. A set of distinct HO definitions is a *library*. Let $\mathcal{P}_{PI} \subset \mathcal{P}$ be a set of predicate symbols reserved for invented predicates.

Definition 5 *A f.f.d theory $\mathfrak{T}$ is* interpretable *if $\forall c \in \mathfrak{T}$, $ag_h(hd(c)) = \emptyset$ and $\forall l \in bd(c)$, l is higher-order ground,*

a) if $ag_h(l) \neq \emptyset$, then $\forall c' \in \mathfrak{T}$, $sy(hd(c')) \neq sy(l)$,

b) $\forall p \in ag_h(l), \exists c' \in \mathfrak{T}$, s.t. $sy(hd(c')) = p \in \mathcal{P}_{PI}$.

Atoms s.t. $ag_h(l) \neq \emptyset$ are external. *The set of external atoms of an interpretable theory $\mathfrak{T}$ is denoted by $ex(\mathfrak{T})$.*

[5] $sy(l)$, $ag_h(l)$, and $ag_f(l)$ apply to literals with similar affect.

[6] Similar to definitional HO of W. Wadge [Wad91].

Let $S_{PI}(\mathfrak{T}) = \{p_i \mid p_i \in ag_h(a) \wedge a \in ex(\mathfrak{T})\}$, the set of predicates which need to be invented. During the **generate phase** we enforce invention of $S_{PI}(\mathfrak{T})$ by pruning programs which contain external literals, but do not contain clauses for their arguments. We discuss this in more detail in Section 3.3.

Otherwise, interpretable theories do not require significant adaption of *Popper*'s *generate, test, constrain loop* [CM21a]. The HO arguments of external literals are ignored by the ASP solver, which searches for so-called *principle programs* (an FO representation of interpretable theories).

Example 1 *Consider* `reverse`, $\texttt{half}_{\texttt{lst}}$, *and* `issubtree` *of Section 2.1 & 2.2. Each is an interpretable theory. The sets of external literals of these theories are* $\{\texttt{fold}(p, C, A, B)\}$, $\{\texttt{case}_{\texttt{list}}(p_{[\,]}, p_{[H|T]}, C, B), \texttt{case}_{\texttt{list}}(p_{[\,]}, p_{[H|T]}, D, E)\}$, *and* $\{\texttt{any}(\texttt{cond}, C, B)\}$, *respectively.*

Definition 6 *Let L be a library, and $\mathfrak{T}$ an interpretable theory. $\mathfrak{T}$ is L-*compatible *if $\forall l \in ex(\mathfrak{T}), \exists d \in L$. s.t. $hd(d)\sigma = l$ for some substitution σ. Let $df(L, l) = d$ and $\theta(L, l) = \sigma$.*

Example 2 *The program in Section 2.1 is L-compatible with*

$$\boxed{\begin{array}{l} \texttt{fold}(P, A, B, C) \vdash \texttt{empty}(B), C = A. \\ \texttt{fold}(P, A, B, C) \vdash \texttt{head}(B, H), \texttt{P}(H, D), \\ \qquad\qquad\qquad\qquad \texttt{tail}(B, T), \texttt{fold}(P, D, T, C). \end{array}}$$

Let $l = fold(p, C, A, B)$: $def(L, l) = \texttt{fold}(P, A, B, C)$ and $\theta(L, l) = \{P \mapsto p, A \mapsto C, B \mapsto A, C \mapsto B\}$.

An L-compatible theory $\mathfrak{T}$ can be L-*grounded*. This requires adding instances of $d \in L$ to $\mathfrak{T}$ (one per external literal) and removal of all HO arguments. The principle program contains all clauses derived from $\mathfrak{T}$ (See Example 3).

Example 3 *Using the Library defined in Example 2 and the Program from Section 2.1, we get the following L-grounding:*

$$\boxed{\begin{array}{r l} \texttt{reverse}(A, B) \vdash & \texttt{empty}(C),\ \texttt{fold}(C, A, B). \\ \texttt{p}(A, B, C) \vdash & \texttt{head}(C, B),\ \texttt{tail}(C, A). \\ \texttt{fold}(A, B, C) \vdash & \texttt{empty}(B), A = C. \\ \texttt{fold}(A, B, C) \vdash & \texttt{head}(B, H), \texttt{p}(H, D), \\ & \texttt{tail}(B, T), \texttt{fold}(D, T, C). \end{array}}$$

The first two clauses form the principle program.

3.3 Interpretable Theories and Constraints

The constraints of Section 2.3 are based on Θ-*subsumption*:

Definition 7 (Θ-subsumption) *An FO theory T_1 subsumes an FO theory T_2, denoted by $T_1 \leq_\theta T_2$ iff, $\forall c_2 \in T_2 \exists c_1 \in T_1$ s.t. $c_1 \leq_\theta c_2$, where $c_1 \leq_\theta c_2$ iff, $\exists \theta$ s.t. $c_1\theta \subseteq c_2$.*

Importantly, the following property holds:

Proposition 1 *if $T_1 \leq_\theta T_2$, then $T_1 \models T_2$*

The pruning ability of *Popper*'s Generalization and specialization constraints follows from Proposition 1.

Definition 8 *An FO theory T_1 is a* generalization *(*specialization*) of an FO theory T_2 iff $T_1 \leq_\theta T_2$ ($T_2 \leq_\theta T_1$).*

Given a library L and a hypothesis space of L-compatible theories, we can compare L-groundings using Θ-subsumption and prune generalizations (specializations), based on the **Test phase**.

Groundings and Elimination Constraints
During the **generate phase**, elimination constraints prune separable programs (See Footnote 4). While L-groundings are non-separable, and thus avoid pruning in the presence of elimination constraints, it is inefficient to query the ASP solver for L-groundings. The ASP solver would have to know the library and how to include definitions. Furthermore, the library must to written in an ASP-friendly form [CM21a].

Instead we query the ASP solver for the *principle program*. The definitions from the library L are treated as BK[7]. Consider Example 3, during the **generate phase** the ASP solver may return an encoding of the following clauses:

$$\boxed{\begin{array}{r} \mathtt{reverse}(A,B) \vdash \mathtt{empty}(C),\ \mathtt{fold}(C,A,B). \\ \mathtt{p}(A,B,C) \vdash \mathtt{head}(C,B),\ \mathtt{tail}(C,A). \end{array}}$$

During the **test phase** the rest of the L-grounding is reintroduced. While this eliminates inefficiencies, the above program is now separable and may be pruned. To efficiently implement HO synthesis we relaxed the elimination constraint in the presence of a library. Instead we introduce so called *call graph constraints* defining the relationship between HO literals and auxiliary clauses.

3.4 Negation, Generalization, and Specialization

Negation of HO literals can interfere with *Popper* constraints. Consider the ILP task and candidate programs:

$$E^+ : f(b).\quad f(c). \qquad \mathbf{E}^- : f(a).$$

$$BK : \left\{ \begin{array}{ll} p(a). & p(b). \\ q(a). & q(c). \end{array} \right\} \quad \mathbf{HO} : N(P,A) \vdash \mathtt{not}\ P(A).$$

$$\underline{prog_s} \qquad\qquad \underline{prog_f}$$

$$\boxed{\begin{array}{l} \mathtt{f}(A) \vdash \mathtt{N}(p_1,A). \\ \mathtt{p_1}(A) \vdash \mathtt{p}(A),\ \mathtt{q}(A). \end{array}} \qquad \boxed{\begin{array}{l} \mathtt{f}(A) \vdash \mathtt{N}(p_1,A). \\ \mathtt{p_1}(A) \vdash \mathtt{p}(A). \end{array}}$$

The optimal solution is $prog_s$ and $prog_f \leq_\theta prog_s$. Note, $prog_f \models f(b) \wedge \neg f(a) \wedge \neg f(c)$, it does not entail all of E^+. We should generalize $prog_f$ to find a solution, i.e. add literals to p_1. The introduced constraints [CM21a] prune programs extending p_1, i.e. $prog_s$. Similar holds for specializations. Consider the ILP task and candidate programs:

$$E^+ : f(a).\quad f(b). \qquad \mathbf{E}^- : f(c).\quad f(d).$$

$$BK : \{\ p(d).\quad q(c).\ \} \quad \mathbf{HO} : N(P,X) \vdash \mathtt{not}\ P(X).$$

$$\underline{prog_s} \qquad\qquad \underline{prog_f}$$

$$\boxed{\begin{array}{l} \mathtt{f}(A) \vdash \mathtt{N}(p_1,A). \\ \mathtt{p_1}(A) \vdash p(A). \\ \mathtt{p_1}(A) \vdash \mathtt{q}(A). \end{array}} \qquad \boxed{\begin{array}{l} \mathtt{f}(A) \vdash \mathtt{N}(p_1,A). \\ \mathtt{p_1}(A) \vdash \mathtt{p}(A). \end{array}}$$

The optimal solution is $prog_s$ and $prog_s \leq_\theta prog_f$. Note, $prog_f \models f(a) \wedge f(b) \wedge f(c)$, it entails some of E^-. We should specialize p_1 to find a solution, i.e. add clauses to $prog_f$. The introduced constraints [CM21a] prune programs that add clauses, i.e. $prog_s$.

[7] Our HO definitions are similar to the BK_{in} of *Metagol*$_{\text{HO}}$.

Handling negation of invented predicates is feasible, but non-trivial as it would require significant changes to the constraint construction procedure. We leave it to future work.

4 Experiments

A possible, albeit very weak, program synthesizer is an enumeration procedure which orders all possible programs constructable from the BK by size, testing each until a solution is found. In [CM21a], this procedure was referred to as *Enumerate*. *Popper* is an extension of *Enumerate* that prunes the hypothesis space based on the performance of previously tested programs.

The pruning mechanism will never prune the shortest solution. Thus, the important question to ask when evaluating *Popper*, and in our case *Hopper*, is not if *Popper* will find a solution, nor is it a high quality solution, but rather how long it takes *Popper* to find the solution. An extensive suite of experiments was presented in [CM21a], illustrating that *Popper* not only significantly outperforms *Enumerate*, but also existing ILP systems.

One way to improve the performance of *LFF*-based ILP systems, like *Popper*, is to introduce techniques which shorten or simplify the solution. The authors of [CMM20], in addition to introducing *Metagol*$_{\text{HO}}$ and *HEXMIL*$_{\text{HO}}$, provided a comprehensive suite of experiments illustrating that the addition of HO predicates can improve accuracy and, most importantly, reduce learning time. The reduction in learning times results from HO predicates reducing the complexity and size of the solutions.

The experiments presented in [CM21a] thoroughly cover scalability issues and learning performance with respect to simple list transformation task, but do not cover performance on more complex task with large solutions. The experiments presented in [CMM20] illustrate performance gains when a HO library is used to solve many simple task and how the addition of HO predicates allow the synthesis of relatively complex predicates such as `Droplast`. When the solution is large *Popper*'s performance degrades significantly. When the solution requires complex interaction between predicates and clauses it becomes exceedingly difficult to find an appropriate set of metarules without being overly descriptive or suffering from long learning times.

Our experiments illustrate that combining *Popper* with HO predicates [CMM20] significantly improve *Popper*'s performance at learning complex programs. Similar to the tasks considered in [CM21a], our language bias consist of predicate declarations `body_pred(head,2)`, type declarations `type(head,(list,element))`, and direction declarations `direction(head,(in,out))`. Additionally, we provide the parameters required for *Popper*'s search mechanism, `max_var`, `max_body`, and `max_clauses`. Also, we may need to enable recursion or disable the datalog restriction.

Concerning the individual tasks, we reevaluated 7 of the tasks presented in [CM21a] and 2 presented in [CMM20]. Additionally, we added 8 list manipulation tasks, 3 tree manipulation tasks, and 2 arithmetic tasks (separated by type in Table 1). Our additional tasks are significantly harder than the tasks evaluated in previous work. Concerning particular

Task	*Popper*	#Literals	PI?	***Hopper* (ours)**	#Literals	HO-Predicates	*Metagol*$_{\mathrm{HO}}$	Metatypes?
Learning Programs by learning from Failures [CM21a]								
dropK	1.1s	7	no	**0.1s**	4	iterate2	no	no
allEven	0.2s	7	no	**0.1s**	4	all	yes	no
findDup	0.5s	8	no	**0.5s**	10	caseList	no	yes
length	<u>0.1s</u>	7	no	**<u>0.2s</u>**	5	fold	yes	no
member	0.1s	5	no	**0.1s**	4	any	yes	no
sorted	65.0s	9	no	**0.4s**	6	fold	yes	no
reverse	11.2s	8	no	**0.5s**	6	fold	yes	no
Learning Higher-Order Logic Programs [CMM20]								
dropLast	300.0s	10	no	**2.9s**	6	map	yes	no
encryption	300.0s	12	no	**1.2s**	7	map	yes	no
Additional Tasks								
repeatN	5.0s	7	no	**0.1s**	5	iterate2	yes	no
rotateN	300.0s	10	no	**2.6s**	6	iterate2	yes	no
allSeqN	300.0s	25	yes	**5.0s**	9	iterate2, map	yes	no
dropLastK	300.0s	17	yes	**37.7s**	11	map	no	no
firstHalf	300.0s	14	yes	**0.5s**	9	iterate2Step	yes	no
lastHalf	300.0s	12	no	**155.2s**	12	caseList	no	yes
of1And2	300.0s	13	no	**6.9s**	13	try	no	no
isPalindrome	300.0s	11	no	**2.4s**	9	condList	no	yes
depth	300.0s	14	yes	**10.1s**	8	fold	yes	yes
isBranch	300.0s	17	yes	**25.9s**	12	caseTree, any	no	yes
isSubTree	18.4s	11	yes	**0.9s**	7	any	yes	yes
addN	300.0s	15	yes	**1.4s**	9	map, caseInt	yes	no
mulFromSuc	300.0s	19	yes	**1.2s**	7	iterate2, iterate3	yes	no

Table 1: All three systems were run on a single core with a timeout of 300 second. Times denote the average of 5 runs. Evaluation time for *Popper* and *Hopper* was set to a thousandth of a second, sufficient time for all task involved.

task: *isPalindrome* does not include `reverse` in the BK, and *mulFromSuc* requires deriving multiplication from `suc`, `zero`. For the most part, the remaining task have self-descriptive titles or were discused earlier in this paper.

For each task, an optimal solution is present in the hypothesis space. Thus, our only concern is how long it takes *Popper* and *Hopper* to find the solution. In some cases, the tasks cannot be solved by *Popper* without *Predicate Invention* (See Column *PI?* of Table 1). For the most part, we used conventional HO constructions. The predicates used for a particular task are listed in Column *HO-Predicate* of Table 1. One exception is *iterate2Step*, a variation of iterate2 with an arbitrary step predicate.

We run *Popper* and *Hopper* on each tasks 5 times and recorded the average running time. For 13 of the 22 tasks, *Popper* timed out (300 seconds elapsed), while *Hopper* solved every tasks. Furthermore, *Popper* was slower than *Hopper* on all but 3 of the tasks and was faster than *Hopper* on 1 tasks (insignificantly faster). Even when the HO solution required the same number of literals, *Hopper* performed better.

In addition we attempted to solve each task with *Metagol*$_{\mathrm{HO}}$. Successful learning using *Metagol*$_{\mathrm{HO}}$ is highly dependent on the choice of the metarules, thus, to simplify matters, our metarules mimic the clauses found in the solution. In some cases, this requires explicitly limiting how certain variables are instantiated by adding declarations, i.e. `metagol:type(Q,2,head_pred)`, to the body of a metarule (denoted by *metatype* in Table 1). These choices amount to significant human-guidance, and thus simplifies learning. Nonetheless, *Metagol*$_{\mathrm{HO}}$ still fails to solve 7 tasks. Without Guidance *Metagol*$_{\mathrm{HO}}$ would fair much worse.

5 Conclusion and Future Work

We introduced an extension of the LFF-based ILP system *Popper* that effectively uses user provided HO definitions during learning. Our experiments show that *Hopper* outperforms *Popper* on most tasks, especially the harder tasks we introduced in this work (Section 4). Additionally, *Hopper* requires minimal guidance when compared to the top-performing MiL-based ILP system *Metagol*$_{\mathrm{HO}}$. Our experiments only test the theoretical possibility of *Metagol*$_{\mathrm{HO}}$ finding a solution as we provide significant guidance. However, given the sensitivity to metarules choice and the fact that many tasks have ternary and even 4-ary predicates, it is unreasonable to expect as detailed of an analysis as in the dyadic case.

Additionally, we provide a theoretical framework encapsulating the accepted HO definitions and discuss the limitations of this framework. We succinctly describe the accepted definitions as those which are monotone over subsumption and entailment, but provide a detailed account in Section 3. The main limitation of this framework concerns HO-negation which we leave to future work. Our framework also allows for invention of HO predicates during learning through constructions of the form $ho(P, Q, x, y) \vdash P(Q, x, y)$. We can verify that *Hopper* can, in principle, find the solution, but we have not managed to successfully invent an HO predicates during learning. We plan for further investigation of this problem.

References

[Blo99] Hendrik Blockeel. Top-down induction of first order logical decision trees. *AI Commun.*, 12(1–2):119–120, jan 1999.

[CDEM21] Andrew Cropper, Sebastijan Dumančić, Richard Evans, and Stephen H. Muggleton. Inductive logic programming at 30. *Machine Learning*, 2021.

[CM14] Andrew Cropper and Stephen H. Muggleton. Logical minimisation of meta-rules within meta-interpretive learning. In Jesse Davis and Jan Ramon, editors, *Inductive Logic Programming - 24th International Conference, ILP 2014, Nancy, France, September 14-16, 2014, Revised Selected Papers*, volume 9046 of *Lecture Notes in Computer Science*, pages 62–75. Springer, 2014.

[CM16] Andrew Cropper and Stephen H. Muggleton. Metagol system. https://github.com/metagol/metagol, 2016.

[CM21a] Andrew Cropper and Rolf Morel. Learning programs by learning from failures. *Mach. Learn.*, 110(4):801–856, 2021.

[CM21b] Andrew Cropper and Rolf Morel. Predicate invention by learning from failures, 2021.

[CMM20] Andrew Cropper, Rolf Morel, and Stephen Muggleton. Learning higher-order logic programs. *Mach. Learn.*, 109(7):1289–1322, 2020.

[Cro20] Andrew Cropper. Forgetting to learn logic programs. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 3676–3683. AAAI Press, 2020.

[CT20] Andrew Cropper and Sophie Tourret. Logical reduction of metarules. *Mach. Learn.*, 109(7):1323–1369, 2020.

[EG18] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *J. Artif. Intell. Res.*, 61:1–64, 2018.

[ERS16] Thomas Eiter, Christoph Redl, and Peter Schüller. Problem solving using the HEX family. In Christoph Beierle, Gerhard Brewka, and Matthias Thimm, editors, *Computational Models of Rationality, Essays dedicated to Gabriele Kern-Isberner on the occasion of her 60th birthday*, pages 150–174. College Publications, 2016.

[GKKS19] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot asp solving with clingo. *Theory and Practice of Logic Programming*, 19(1):27–82, 2019.

[KEI18] Tobias Kaminski, Thomas Eiter, and Katsumi Inoue. Exploiting answer set programming with external sources for meta-interpretive learning. *Theory Pract. Log. Program.*, 18(3-4):571–588, 2018.

[Law18] Mark Law. *Inductive learning of answer set programs*. PhD thesis, Imperial College London, UK, 2018.

[Lif19] Vladimir Lifschitz. *Answer Set Programming*. Springer, 2019.

[Llo87] John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[MLPT14] Stephen H. Muggleton, Dianhuan Lin, Niels Pahlavi, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning: application to grammatical inference. *Mach. Learn.*, 94(1):25–49, 2014.

[MRP+12] Stephen Muggleton, Luc De Raedt, David Poole, Ivan Bratko, Peter A. Flach, Katsumi Inoue, and Ashwin Srinivasan. ILP turns 20 - biography and future challenges. *Mach. Learn.*, 86(1):3–23, 2012.

[Mug91] Stephen Muggleton. Inductive logic programming. *New Gener. Comput.*, 8(4):295–318, 1991.

[Mug95] Stephen Muggleton. Inverse entailment and progol. *New Gener. Comput.*, 13(3&4):245–286, 1995.

[NCWSC97] Shan-Hwei Nienhuys-Cheng, Ronald de Wolf, J. Siekmann, and J. G. Carbonell. *Foundations of Inductive Logic Programming*. Springer-Verlag, Berlin, Heidelberg, 1997.

[NdW97] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Computer Science*. Springer, 1997.

[Plo70] Gordon D. Plotkin. A note on inductive generalization. *Machine Intell.*, 5(1):153–163, 1970.

[QCJ93] J. R. Quinlan and R. M. Cameron-Jones. Foil: A midterm report. In Pavel B. Brazdil, editor, *Machine Learning: ECML-93*, pages 1–20, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[Qui90] J. Ross Quinlan. Learning logical definitions from relations. *Mach. Learn.*, 5:239–266, 1990.

[Rae08] Luc De Raedt. *Logical and relational learning*. Cognitive Technologies. Springer, 2008.

[Rey70] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intel.*, 5(1):135–151, 1970.

[SKB03] Ashwin Srinivasan, Ross D. King, and Michael E. Bain. An empirical study of the use of relevance information in inductive logic programming. *J. Mach. Learn. Res.*, 4(null):369–383, dec 2003.

[SL08] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California at Berkeley, USA, 2008.

[Sri01] Ashwin Srinivasan. The aleph manual. Technical report, Machine Learning at the Computing Laboratory, Oxford University, 2001.

[Wad91] William W. Wadge. Higher-order horn logic programming. In Vijay A. Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, Oct. 28 - Nov 1, 1991*, pages 289–303. MIT Press, 1991.

A Implementation

We implement our method by building on top of code provided by [CM21a]. The changes we applied include:

Processing HO predicates We allow user to declare some background knowledge predicates to be HO. Based on these declarations we generate ASP constraints discussed in subsection 3.2 and Prolog code that allows execution of programs generated with those constraints.

Generating context-passing versions of HO predicates Sometimes the HO argument predicates (referred to as $S_{PI}(\mathfrak{T})$ in subsection 3.2) require context that exists in the predicate that calls them, however is inaccessible to them in our framework. To make it accessible we support automating generation of *more contextual* versions of HO predicates. These predicates have higher arity and take more FO arguments. These arguments are only used in HO calls, and are simply passed as arguments to HO predicate calls. In [CMM20] this process is referred to as „currying" (though it is somewhat different to what *currying* is usually considered to be).

Example 4 *From a HO* `map` *predicate*

$$\begin{array}{l} \mathtt{map}(P,[\,],[\,]). \\ \mathtt{map}(P,[H_1|T_1],[H_2|T_2]) \vdash P(H_1,H_2), \mathtt{map}(P,T_1,T_2). \end{array}$$

we automatically generate a more contextual version

$$\begin{array}{l} \mathtt{map}(P,[\,],[\,],V). \\ \mathtt{map}(P,[H_1|T_1],[H_2|T_2],V) \vdash P(H_1,H_2,V), \\ \qquad \mathtt{map}(P,T_1,T_2,V). \end{array}$$

which allows for construction of a program that adds a number to every element of a list using `map`

$$\begin{array}{l} \mathtt{f}(A,B,C) \vdash \mathtt{map}(\mathtt{p_1},B,C,A). \\ \mathtt{p_1}(A,B,C) \vdash \mathtt{add}(A,C,B). \end{array}$$

Force all generated code to be used Since ASP can now generate many different predicates, some of them might not even be called in the main predicate. To avoid such useless code we make ASP keep track of a *call graph* – which predicates call which other predicates, and add a constraint that forces every defined defined predicate to be called (possibly indirectly) by the main predicate. This not only removes many variations of effectively the same program, but also significantly prunes the hypothesis space, pruning programs ignored by other constraints (explained in sec. 3.3).

Changes to separability and recursion We add a few small changes to solve the problems that appear when generating multiple predicates. We make sure that clauses that call HO predicates (and thus different predicates from the program) are not considered *separable*. We also change how recursion is handled – otherwise recursion would allow all invented arguments to be called everywhere in the program, needlessly increasing search space.

B Experimental details

Here we describe all tasks and HO predicates presented in Section 4.

B.1 Higher-Order predicates

`all/2` – true that the argument predicate is true for all elements of the list
`caseList/4` – a deconstructor for a list, calling first or second predicate depending on whether the list is empty
`caseTree/4` – a deconstructor for a tree, calling first or second predicate depending on whether the tree is a leaf
`caseInt/4` – a deconstructor for a natural number, calling first or second predicate depending on whether the number is 0.
`fold/4` – combines all elements of a list using the argument predicate.
`map/3` – checks that the output is a list of the same length as input, such that the argument predicate holds between all their elements.
`try/3` – checks whether at least one of argument predicates hold on the last argument
`iterate2/3` – iterate the argument predicate n times
`iterate2Step/5` – a variant of `iterate2/3`, but the iterator instead of being modified by 1 is modified using an argument predicate. Also the output here is a list of all intermediate values.
`iterate3/5` – „curried" version of `iterate2`

B.2 Learning tasks

`dropK/3` – drop first k elements from a list
`allEven/1` – check whether all elements on a list are even
`findDup/2` – check whether an element is present on a list at least twice
`length/2` – find length of a list
`member/2` – check whether an element is on a list
`sorted/1` – check whether a list is sorted (non-decreasing)
`reverse/2` – reverse a list
`dropLast/2` – given a list of lists, drop the last element from each list
`encryption/2` – convert characters to integers, add 2 to each of them, then convert them back
`repeatN/3` – construct a list made of the input argument repeated n times
`rotateN/3` – move first element of a list to the end n times
`allSeqN/2` – construct a list of lists, consisting of all sequences from 1 to i with $i \leq n$.
`dropLastK/3` – given a list of lists, drop the last k elements from each list
`firstHalf/2` – check whether the second argument is equal to the first half of the first argument
`lastHalf/2` – check whether the second argument is equal to the last half of the first argument
`of1And2/1` – check whether a list constists only of 1s and 2s
`isPalindrome/1` – check whether a list is a palindrome (the same read normally and in reverse)
`depth/2` – find the depth of a tree
`isBranch/2` – check whether a given list is a branch of the tree
`isSubTree/2` – check whether the second argument is a subtree of the first argument
`addN/3` – add n to every element of a list (with no addition predicate in the background knowledge)

`mulFromSuc/3` – multiply two numbers (with no addition predicate in the background knowledge)