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

This work is licensed under a CC BY 4.0 license.
Learning Higher-Order Programs without Meta-Interpretive Learning

Stanisław J. Purgał¹ ‡ † , David M. Cerna²§ ‡ , Cezary Kaliszyk¹ ‡

¹University of Innsbruck, Innsbruck, Austria
²Czech Academy of Sciences Institute of Computer Science (CAS ICS), Prague, Czechia
³Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria

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.

\begin{align*}
\text{allSeq}(N,L) &\leftarrow \text{iterate}(\text{succ}, 0, N, A), \text{map}(p, A, L).
\end{align*}

\begin{align*}
p(A, B) &\leftarrow \text{iterate}(\text{succ}, 0, A, B).
\end{align*}

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].

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] (Popper), 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, MetagolHO [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 binary, poses a challenge for some systems, and in the case of HEXMIL-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: \( p_1 \leq_0 (\vdash) p_2 \Rightarrow H(p_1) \leq_0 (\vdash) H(p_2) \) 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 constructs. 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 [KE18]. We also introduce Popper [CM21a], the system Popper is based on. Additionally, a detailed survey of the current state of ILP research has recently been published [CDM21].

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:

\[
\text{reverse}(A, B) \leftarrow \text{empty}(C), \text{fold}(p, C, A, B).
\]

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], Prolog [Mug95], Tilde [Bli09], 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 lists and trees.

The best performing systems with respect to the aforementioned tasks are Metagol [CM16] and HEXMIL [KE18]; 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 \( A_0 \leftarrow A_1, \ldots, A_n \), where \( A_i \) is a literal \( P(T_1, \ldots, T_m) \), 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. Popper, our extension, has drastically improved performance when compared with Popper. Popper 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 Popper 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\(_{HO} \) is the inclusion of interpreted background knowledge \( BK_{in} \). For example, map/3 as \( BK_{in} \) takes the form:

\[
\text{ibk}((\text{map},[],[[],[]]),[]).
\]

\[
\text{ibk}((\text{map},[A|As],[B|Bs],F),[[F,A,B],[\text{map},As,Bs,F]]).
\]

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 \( \text{half}_{fast}(A,B) \), which computes the last half of a list\(^2\), illustrates why this may be problematic:

\[
p(A,B) \leftarrow q(A,C), r(C,B).
\]

\[
\text{half}_{fast}([1,2],[2]), \text{half}_{fast}([1,2,3],[3]), \text{half}_{fast}([1,2],[1]), \text{half}_{fast}([1,2,3],[1,2]).
\]

\(^1\) body = \( \{P(A,B) \leftarrow q(A,C), r(C,B)\} \)

\(^2\) body = \( \{P(A,B) \leftarrow q(A,C), r(C,B)\} \)
The HO predicate `case1st(p1, P[H|T], C, B)` can be subdivided into the following cases:

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_{1</td>
<td>a}(A) = \emptyset(A) )</td>
</tr>
<tr>
<td>( p_{[H</td>
<td>T]}(A, B, C) = \emptyset(B), \emptyset(C) )</td>
</tr>
<tr>
<td>( p_{[H</td>
<td>T]}(A, B, C) = \front(B, D) )</td>
</tr>
<tr>
<td>( \text{case1st}(p_{1</td>
<td>a}, P[H</td>
</tr>
</tbody>
</table>

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. Popper 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 MetagolHO (see Section 4.2 of [CMM20]). Popper has similar issues which are discussed in Section 3.4.

**Higher-order HEMXIL**

HEXMIL is an ASP encoding of Meta-interpretive Learning [KIE18]. Given that ASP can be quite restrictive, HEMXIL exploits the HEX formalism for encoding MiL. HEX allows the ASP solver to interface with external resources [ERS16]. Unlike Metagol, HEMXIL is restricted to forward-chained metarules:

**Definition 2** A forward-chained metarule is of the form:

\[
P(A, B) \equiv Q_1(A, C_1), Q_2(C_1, C_2), \ldots, Q_n(C_{n-1}, B), R_1(D_1), \ldots, R_m(D_m),
\]

where \( D_i \in \{ A, C_1, \ldots, C_{n-1}, B \} \).

This restricts us to Dyadic learning task. Furthermore, many useful metarules are not of this form, i.e. \( P(A, B) \Rightarrow Q(A, B), R(A, B) \). HEMXILHO incorporates HO definitions into the forward-chained structure of Definition 2.

**Definition 3** A forward-chained HO definition consists of clauses of the form: \( P(A, B) \Rightarrow k_1(A, C_1, P_1, \ldots, P_{k_1}), \ldots, k_n(C_{n-1}, B, P_n, \ldots, P_{k_n}) \), where every atom in this clause can have \( 0 \leq k_i \) 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 HEMXILHO's

\[
\text{half1st}(A, B) \vdash \text{reverse}(A, C),
\text{case1st}(p_{1|a}, P[H|T], C, B),
\text{p}_{1|a}(A) = \emptyset(A),
\text{p}_{[H|T]}(A, B, C) = \emptyset(B), \emptyset(C),
\text{p}_{[H|T]}(A, B, C) = \front(B, D)^3,
\text{case1st}(p_{1|a}, P[H|T], D, E),
\text{append}(E, A, C).
\]

The HO predicate `case1st(p_{1|a}, P[H|T], A, B)` calls `p_{1|a}` if `A` is empty and `p_{[H|T]}` otherwise. Our definition of \( \text{half1st}(A, B) \) cannot be found using the standard search procedure as every occurrence of `case1st` results in a call to the meta-interpreter's proof procedure. The underlined call to `case1st` results in PI for `p_{[H|T]}` ad infinitum. Similarly, the initial goal cannot be substituted unless it's explicitly specified.

As with `half1st(A, B)`, the following program defining `issubtree(A, B)`, which computes whether `B` is a subtree of `A`, requires recursively calling `issubtree` through any.

\[
\text{issubtree}(A, B) \vdash A = B.
\text{issubtree}(A, B) \vdash \text{children}(A, C), \text{any}(\text{cond}, C, B).
\text{cond}(A, B) \vdash \text{issubtree}(A, B).
\]

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. Popper successfully learns these predicates without any significant drawbacks.

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. Crompton 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 II [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 \( \Theta \)-subsumption [Plo70; Rey70]. Popper may also introduce Elimination Constraints pruning separable\(^3\) 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 [NCWCG07], 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 [GKK97] 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

\(^3\)No head literal of a clause in the set occurs as a body literal of a clause in the set.
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 [Lio87].

3.1 Preliminaries

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

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

A clause is a set of literals. A Horn clause contains at most one positive literal while a definite clause must have otherwise it is a definite clause.

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 (ff,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 \( P_1, \ldots, P_n, \Theta_1, \ldots, \Theta_m \) by predicate symbols \( p_1, \ldots, p_n \) and terms \( t_1, \ldots, t_m \) is a substitution \( \theta \) denoted by \( \{ P_1 \mapsto p_1, \ldots, P_n \mapsto p_n, \Theta_1 \mapsto t_1, \ldots, \Theta_m \mapsto t_m \} \). A substitution \( \theta \) unifies two atoms when \( \theta \theta = \theta \).  

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 \( sy(a) \notin \mathcal{V}_h \), 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 \( \mathcal{P}_{P_1} \subset \mathcal{P} \) be a set of predicate symbols reserved for invented predicates.

Definition 5 A ff,d theory \( \mathcal{T} \) is interpretable if \( \forall c \in \mathcal{T} \),\( ag_h(hd(c)) = \emptyset \) and \( \forall c' \in bd(c), l_i \) is higher-order ground, \( a) \) if \( ag_h(l) \neq \emptyset \), then \( l c \in \mathcal{T}, sy(hd(c')) \neq sy(l) \), \( b) \) if \( \forall p \in ag_h(l), \exists c' \in \mathcal{T}, s.t. sy(hd(c')) = p \in \mathcal{P}_{P_1} \).

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

Let \( S_{P_1}(\mathcal{T}) = \{ p_i | p_i \in ag_h(a) \wedge a \in ex(\mathcal{T}) \} \), the set of predicates which need to be invented. During the generate phase we enforce invention of \( S_{P_1}(\mathcal{T}) \) 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, half1t, and issubtree of Section 2.1 & 2.2. Each is an interpretable theory. The sets of external literals of these theories are \( \{ \text{fold}(p, C, A, B), \{ \text{case1ist}(\{p_1, \ldots, p_n\}, C, B), \text{case1ist}(\{p_1, \ldots, p_n\}, D, E) \} \), and \( \{ \text{any}(C, B) \} \), respectively.

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

Example 2 The program in Section 2.1 is L-compatible with \( df(P, A, B, C) = A. \)

\( \text{fold}(p, A, B, C) \models \text{head}(B, H), p(H, D), \)  
\( \text{tail}(B, T), \text{fold}(P, D, T, C). \)

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

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

\( \text{reverse}(A, B) \models \text{empty}(C), \text{fold}(C, A, B). \)
\( p(A, B, C) \models \text{head}(C, B), \text{tail}(C, A). \)
\( \text{fold}(A, B, C) \models \text{empty}(B), A = C. \)
\( \text{fold}(A, B, C) \models \text{head}(B, H), p(H, D), \)  
\( \text{tail}(B, T), \text{fold}(D, T, C). \)

The first two clauses form the principle program.

3.3 Interpretable Theories and Constraints

The constraints of Section 2.3 are based on \( \Theta \)-subsumption:

Definition 7 (\( \Theta \)-subsumption) An FO theory \( T_1 \) subsumes an FO theory \( T_2 \), denoted by \( \Theta \leq_\Theta T_2 \), s.t. \( c_1 \leq_\Theta c_2 \), where \( c_1 \leq_\Theta c_2 \), if \( \exists \theta \) s.t. \( c_1 \theta \subseteq c_2 \).

Importantly, the following property holds:

Proposition 1 if \( \Theta \leq_\Theta T_2 \), 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 \( T_2 \) if \( T_1 \models T_2 \) (\( T_2 \models T_1 \)).

Given a library \( L \) and a hypothesis space of L-compatible theories, we can compare L-groundings using \( \Theta \)-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 be 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:

$$\begin{align*}
\text{reverse}(A, B) &\rightarrow \text{empty}(C), \text{fold}(C, A, B). \\
p(A, B, C) &\rightarrow \text{head}(C, B), \text{tail}(C, A).
\end{align*}$$

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:

$$\begin{align*}
E^+ &\rightarrow f(b) &f(c). \\
E^- &\rightarrow f(a). \\
BK &\rightarrow p(a), p(b), q(a), q(c). \\
\text{prog} &\rightarrow \text{HO} : N(P,A) \rightarrow \text{not} P(A).
\end{align*}$$

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

$$\begin{align*}
E^+ &\rightarrow f(a) &f(c). \\
E^- &\rightarrow f(c) &f(d). \\
BK &\rightarrow p(d), q(c). \\
\text{prog} &\rightarrow \text{HO} : N(P, X) \rightarrow \text{not} P(X).
\end{align*}$$

The optimal solution is $\text{prog}_s$ and $\text{prog}_s \leq \text{prog}_s$. Note, $\text{prog}_s \models f(a) \land f(b) \land f(c)$, it entails some of $E^-$. We should specialize $p_1$ to find a solution, i.e. add clauses to $\text{prog}_s$. The introduced constraints [CM21a] prune programs that add clauses, i.e. $\text{prog}_s$.

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 [CM20], in addition to introducing *Metagol*$_{LH0}$ and *HEXXILH0*, 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 [CM20] 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 [CM20] significantly improve *Popper*’s performance at learning complex programs. Similar to the tasks considered in [CM21a], our language bias consist of predicate declarations $\text{body} \_ \text{pred}(\text{head}, 2)$. type declarations $\text{type} \_ \text{decl}(\text{head}, (\text{list}, \text{element}))$, and direction declarations $\text{direction} \_ \text{decl}(\text{head}, (\text{in}, \text{out}))$. Additionally, we provide the parameters required for *Popper*’s search mechanism, $\text{max}_\text{var}$, $\text{max}_\text{body}$, and $\text{max}_\text{clauses}$. Also, we may need to enable recursion or disable the datalog restriction.

Concering the individual tasks, we reevaluated 7 of the tasks presented in [CM21a] and 2 presented in [CM20]. 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
Learning Programs by learning from Failures [CM21a]

<table>
<thead>
<tr>
<th>Task</th>
<th>Popper</th>
<th>#Literals</th>
<th>PI?</th>
<th>Hopper (ours)</th>
<th>#Literals</th>
<th>HO-Predicates</th>
<th>Metagol\textsubscript{HO}</th>
<th>Metatypes?</th>
</tr>
</thead>
<tbody>
<tr>
<td>dropK</td>
<td>1.1s</td>
<td>7</td>
<td>no</td>
<td>0.1s</td>
<td>4</td>
<td>iterate2</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>allEven</td>
<td>0.2s</td>
<td>7</td>
<td>no</td>
<td>0.1s</td>
<td>4</td>
<td>all</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>findDup</td>
<td>0.5s</td>
<td>8</td>
<td>no</td>
<td>0.5s</td>
<td>10</td>
<td>caseList</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>length</td>
<td>0.1s</td>
<td>7</td>
<td>no</td>
<td>0.2s</td>
<td>5</td>
<td>fold</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>member</td>
<td>0.1s</td>
<td>5</td>
<td>no</td>
<td>0.1s</td>
<td>4</td>
<td>any</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>sorted</td>
<td>65.0s</td>
<td>9</td>
<td>no</td>
<td>0.4s</td>
<td>6</td>
<td>fold</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>reverse</td>
<td>11.2s</td>
<td>8</td>
<td>no</td>
<td>0.5s</td>
<td>6</td>
<td>fold</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

Learning Higher-Order Logic Programs [CMM20]

<table>
<thead>
<tr>
<th>Task</th>
<th>Popper</th>
<th>#Literals</th>
<th>PI?</th>
<th>Hopper (ours)</th>
<th>#Literals</th>
<th>HO-Predicates</th>
<th>Metagol\textsubscript{HO}</th>
<th>Metatypes?</th>
</tr>
</thead>
<tbody>
<tr>
<td>dropLastK</td>
<td>300.0s</td>
<td>10</td>
<td>no</td>
<td>2.9s</td>
<td>6</td>
<td>map</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>encryption</td>
<td>300.0s</td>
<td>12</td>
<td>no</td>
<td>1.2s</td>
<td>7</td>
<td>map</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task</th>
<th>Popper</th>
<th>#Literals</th>
<th>PI?</th>
<th>Hopper (ours)</th>
<th>#Literals</th>
<th>HO-Predicates</th>
<th>Metagol\textsubscript{HO}</th>
<th>Metatypes?</th>
</tr>
</thead>
<tbody>
<tr>
<td>repeatN</td>
<td>5.0s</td>
<td>7</td>
<td>no</td>
<td>0.1s</td>
<td>5</td>
<td>iterate2</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>rotateN</td>
<td>300.0s</td>
<td>10</td>
<td>no</td>
<td>2.6s</td>
<td>6</td>
<td>iterate2</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>allSeqN</td>
<td>300.0s</td>
<td>25</td>
<td>yes</td>
<td>5.0s</td>
<td>9</td>
<td>iterate2, map</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>dropLastK</td>
<td>300.0s</td>
<td>17</td>
<td>yes</td>
<td>37.7s</td>
<td>11</td>
<td>map</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>firstHalf</td>
<td>300.0s</td>
<td>14</td>
<td>yes</td>
<td>0.5s</td>
<td>9</td>
<td>iterate2Step</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>lastHalf</td>
<td>300.0s</td>
<td>12</td>
<td>no</td>
<td>155.2s</td>
<td>12</td>
<td>caseList</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>of1And2</td>
<td>300.0s</td>
<td>13</td>
<td>no</td>
<td>6.9s</td>
<td>13</td>
<td>try</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>isPalindrome</td>
<td>300.0s</td>
<td>11</td>
<td>no</td>
<td>2.4s</td>
<td>9</td>
<td>condList</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>depth</td>
<td>300.0s</td>
<td>14</td>
<td>yes</td>
<td>10.1s</td>
<td>8</td>
<td>fold</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>isBranch</td>
<td>300.0s</td>
<td>17</td>
<td>yes</td>
<td>25.9s</td>
<td>12</td>
<td>caseTree, any</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>isSubTree</td>
<td>18.4s</td>
<td>11</td>
<td>yes</td>
<td>0.9s</td>
<td>7</td>
<td>any</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>addN</td>
<td>300.0s</td>
<td>15</td>
<td>yes</td>
<td>1.4s</td>
<td>9</td>
<td>map, caseInt</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>mulFromSuc</td>
<td>300.0s</td>
<td>19</td>
<td>yes</td>
<td>1.2s</td>
<td>7</td>
<td>iterate2, iterate3</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

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 \textit{Popper} and \textit{Hopper} was set to a thousandth of a second, sufficient time for all task involved.

task: \textit{isPalindrome} does not include \textit{reverse} in the \textit{BK}, and \textit{mulFromSuc} requires deriving multiplication from \textit{succ}, \textit{zero}. For the most part, the remaining task have self-descriptive titles or were discussed 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 \textit{Popper} and \textit{Hopper} to find the solution. In some cases, the tasks cannot be solved by \textit{Popper} without \textit{Predicate Invention} (See Column \textit{PI}? of Table 1). For the most part, we used conventional HO constructions. The predicates used for a particular task are listed in Column \textit{HO-Predicate} of Table 1. One exception is \textit{iterate2Step}, a variation of \textit{iterate2} with an arbitrary step predicate.

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

In addition we attempted to solve each task with \textit{Metagol\textsubscript{HO}}. Successful learning using \textit{Metagol\textsubscript{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. \texttt{metagol\_type(Q,2,head\_pred)}, to the body of a metarule (denoted by \texttt{metatype} in Table 1). These choices amount to significant human-guidance, and thus simplifies learning. Nonetheless, \textit{Metagol\textsubscript{HO}} still fails to solve 7 tasks. Without Guidance \textit{Metagol\textsubscript{HO}} would fair much worse.

5 Conclusion and Future Work

We introduced an extension of the LFF-based ILP system \textit{Popper} that effectively uses user provided HO definitions during learning. Our experiments show that \textit{Hopper} outperforms \textit{Popper} on most tasks, especially the harder tasks we introduced in this work (Section 4). Additionally, \textit{Hopper} requires minimal guidance when compared to the top-performing MiL-based ILP system \textit{Metagol\textsubscript{HO}}. Our experiments only test the theoretical possibility of \textit{Metagol\textsubscript{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 $\text{ho}(P, Q, x, y) \rightarrow P(Q, x, y)$. We can verify that \textit{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


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 section 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 $S_{P1}(\Sigma)$ 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 [CM20] 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

\[
\text{map}(P, []). \vdash P(H_1, H_2, \text{map}(P, T_1, T_2)).
\]

we automatically generate a more contextual version

\[
\text{map}(P, []). \vdash P(H_1[T_1], H_2[T_2], \text{map}(P, T_1, T_2), V)
\]

which allows for construction of a program that adds a number to every element of a list using \texttt{map}

\[
\begin{align*}
\text{map}(P, []). \vdash P(A, B, C) \leftarrow \text{map}(P_1, B, C, A), \\
\text{add}([], A, C). \\
\end{align*}
\]

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

all12 – 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
allEven1/1 – 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
dropLastK/3 – 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 ≤ 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 consists 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)