Learning Higher-Order Logic Programs from Failures

Stanisław J. Purgał1, David M. Cerna2,3†, 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
stanislaw.purga@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. Our theoretical framework captures a class of higher-order definitions preserving soundness of existing subsumption-based pruning methods.

1 Introduction
Inductive Logic Programming, abbreviated ILP, [Muggleton, 1991; Nienhuys-Cheng et al., 1997] 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.
Naïvely, learning a logic program which takes a positive integer \( n \) and returns a list of list of the form \([[[1], [2], \ldots, [1, \ldots, n]]]]\) would not come across as a formidable learning task. A logic program is easily constructed using conventional higher-order (HO) definitions.

\[
\text{allSeqN}(N, L) :- \text{iterate}(\text{succ}, 0, N, A), \text{map}(p, A, L).
\]

The first \text{iterate}1 produces the list \([1, \ldots, N]\) and \text{map} applies a functionally equivalent \text{iterate} to each member of \([1, \ldots, N]\), 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 [Cropper et al., 2022].

Excessively large BK can, in many cases, lead to performance loss [Cropper, 2020; Srinivasan et al., 2003]. 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 toward short solutions, with the ability to use HO definitions can result in improved performance. We developed an HO-enabled Popper [Cropper and Morel, 2021a] (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_HO [Cropper et al., 2020]. See Section 4.
Existing HO-enabled ILP systems are based on Meta-interpretive Learning (MiL) [Muggleton et al., 2014]. 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, \text{iterate}, being ternary (w.r.t. FO arguments), poses a challenge for some systems, and in the case of HEXMIL_HO [Cropper et al., 2020], this definition cannot be considered as only binary definitions are allowed (w.r.t. FO arguments).

Limiting human participation when fine-tuning the search space is an essential step toward strong symbolic machine learning. The novel Learning from Failures (LFF) paradigm [Cropper and Morel, 2021a], realized through Popper, prunes the search space as part of the learning process. Not only does this decrease human guidance, but 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

Figure 1: Inclusion of HO definitions increases the size of the search space, but can lead to the search space containing a shorter solutions.
change to the theoretical foundations. In the case of LFF learners, like Popper, the pruning mechanism influences which HO definitions may be soundly used (See page 813 of [Cropper and Morel, 2021a]).

We avoid these soundness issues by indirectly adding HO definitions. Hopper uses FO instances of HO definitions each of which is associated with a set of unique predicates symbols denoting the HO arguments of the definition. These predicates symbols occur in the head literal of clauses occurring in the candidate program iff their associated FO instance occurs in the candidate program. Thus, only programs with matching structure may be pruned. We further examine this issue in Section 3 and provide a construction encapsulating the accepted class of HO definitions.

Succinctly, we work within the class of HO definitions that are monotone with respect to subsumption; \( p_1 \preceq_0 (\models) p_2 \Rightarrow H(p_1) \preceq_0 (\models) H(p_2) \) 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). However, 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 [Cropper et al., 2020] (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. We provide a brief summary of this survey and focus on introducing the state-of-the-art MiL-based ILP systems extended by HO definitions. 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 see Section 2.2. Popper [Cropper and Morel, 2021a], does not directly support PI, though, it is possible to enforce PI through the language bias (Poppi is an PI-enabled extension [Cropper and Morel, 2021b]). Popper’s language bias, while partially fixed, is essentially an arbitrary ASP program. The authors of [Cropper and Morel, 2021a] illustrate this by providing ASP code emulating the chain metarule\(^2\) (see Appendix A of [Cropper and Morel, 2021a]). We exploit this feature to extend Popper, allowing it to construct programs containing instances of HO definitions.

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:

\[
\begin{align*}
\text{reverse}(A, B) :- & \text{empty}(C), \text{fold}(p, C, A, B). \\
\text{p}(A, B, C) :- & \text{head}(C, B), \text{tail}(C, A).
\end{align*}
\]

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 [Quinlan, 1990], Prolog [Muggleton, 1995], Tilde [Blockeel, 1999], and Aleph [Srinivasan, 2001] 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” [Muggleton et al., 2012]. Since then, there have been a few successful approaches. Both ILASP [Law et al., 2014] and δILP [Evans and Grefenstette, 2018] 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 [Cropper and Muggleton, 2016] and HEXMIL [Kaminski et al., 2018]; both are based on Meta-interpretive Learning (MiL) [Muggleton et al., 2014], 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 [Cropper and Muggleton, 2014], a restricted form of HO horn clauses.

**Definition 1 ([Cropper and Tourret, 2020])** 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 [Cropper and Morel, 2021a], does not directly support PI, though, it is possible to enforce PI through the language bias (Poppi is an PI-enabled extension [Cropper and Morel, 2021b]). Popper’s language bias, while partially fixed, is essentially an arbitrary ASP program. The authors of [Cropper and Morel, 2021a] illustrate this by providing ASP code emulating the chain metarule\(^2\) (see Appendix A of [Cropper and Morel, 2021a]). We exploit this feature to extend Popper, allowing it to construct programs containing instances of HO definitions.

2.2 Metagol and HEXMIL

We briefly summarize existing HO-capable ILP systems introduced by A. Cropper et al. [Cropper et al., 2020].

Higher-order Metagol

In short, Metagol is a MiL-learner implemented using a Prolog meta-interpreter. As input, Metagol takes a set of predicate declarations \( PD \) of the form \( \text{body}_{\text{pred}}(P/n) \), sets of positive \( E^+ \) and negative \( E^- \) examples, compiled background knowledge \( BK_e \), and a set of metarules \( M \). The examples provide the arity and name of the goal predicate. Initially, Metagol attempts to satisfy \( E^+ \) using \( BK_e \). 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 of the metarule \( m \). If successful, the derivation provides a Prolog program that 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 the construction of a program.

The difference between Metagol and Metagol\(_{HO} \) is the inclusion of interpreted background knowledge \( BK_{in} \). For example, \( \text{map} \) as \( BK_{in} \) takes the form:

\[
\begin{align*}
\text{ibk}([\text{map}, [\text{[]}, [], [], [], [], []]),} \\
\text{ibk}([\text{map}, [\text{A[As]}, \text{B[Bs]}, \text{F}], \text{[F, A, B], [\text{map}, \text{As}, \text{Bs}, \text{F}]]}).
\end{align*}
\]

\(^2\)\( P(A, B) :- Q(A, C), R(C, B) \).
Metagol handles BK\textsubscript{in} as it handles metarules. When used, Metagol attempts to prove the body of \texttt{map}, i.e. \texttt{F(A, B)}. Either \texttt{F} is substituted by a predicate contained in BK\textsubscript{in} or replaced by an invented predicate that becomes the goal atom and is proven using metarules or BK\textsubscript{in}.

A consequence of this approach is that substituting the goal atom by a predicate defined as BK\textsubscript{in} cannot result in a derivation defining a Prolog program. Like with metarules, additional proof steps are necessary. The following program defining half\textsubscript{1st}(A, B), which computes the last half of a list\textsuperscript{3}, illustrates why this may be problematic:

\begin{align*}
\text{half}_{\text{1st}}(A, B) : & \text{- reverse}(A, C), \\
& \text{case}_{\text{1st}}(p_{\mid 1}, p_{\mid H[T]}, C, B), \\
& p_{\mid 1}(A) : \text{- empty}(A), \\
& p_{\mid H[T]}(A, B, C) : \text{- empty}(B), \text{empty}(C), \\
& p_{\mid H[T]}(A, B, C) : \text{- front}(B, D)\textsuperscript{4}, \\
& \text{case}_{\text{1st}}(p_{\mid 1}, p_{\mid H[T]}, D, E), \\
& \text{append}(E, A, B).
\end{align*}

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

As with half\textsubscript{1st}(A, B), the following program defining issubset\textsubscript{e}(A, B), which computes whether B is a subtree of A, requires recursively calling issubset\textsubscript{e} through any:

\begin{align*}
\text{isset}(A, B) & : A = B, \\
\text{isset}(A, B) & : \text{children}(A, C), \text{any}(\text{cond}, C, B), \\
& \text{cond}(A, B) : \text{isset}(A, B).
\end{align*}

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\textsubscript{in} definitions), to the best of our knowledge, is not fully supported by Metagol\textsubscript{HO} (See Section 4.2 of [Cropper et al., 2020]). Hopper has similar issues which are discussed in Section 3.4.

Higher-order HEXMIL

HEXMIL is an ASP encoding of Meta-interpretive Learning [Kaminski et al., 2018]. 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 [Eiter et al., 2016]. HEXMIL is restricted to forward-chained metarules:

Definition 2 Forward-chained metarules are of the form: \(P(A, B) : Q_1(A, C_1), Q_2(C_1, C_2), \ldots, Q_n(C_n-1, B), R_1(D_1), \ldots, R_m(D_m)\) where \(D_i \in \{A, C_1, \ldots, C_n-1, B\}\).

\footnote{half\textsubscript{1st}(\{1, 2\}, \{2\}), half\textsubscript{1st}(\{1, 2, 3\}, \{3\}), half\textsubscript{1st}(\{1, 2, 3\}, \{1, 2\}).}

\footnote{front(A, B) : reverse(A, C), tail(C, D), reverse(D, B).}

Thus, only Dyadic learning task may be handled. Furthermore, many useful metarules are not of this form, i.e. \(P(A, B) : Q(A, B), R(A, B)\). HXEMIL\textsubscript{HO}, incorporates HO definitions into the forward-chained structure of Definition 2. For details concerning the encoding see Section 4.4 of [Cropper et al., 2020]. The authors of [Cropper et al., 2020] illustrated HXEMIL\textsubscript{HO}’s poor performance on list manipulation tasks and its limitations make application to tasks of interest difficult. Thus, we focus on Metagol\textsubscript{HO} in Section 4.

2.3 Hopper: Learning From Failures (LFF)

The LFF paradigm together with Hopper provides a novel approach to inductive logic programming, based on counterexample-guided inductive synthesis (CEGIS) [Solar-Lezama, 2008]. Both LFF and the system implementing it were introduced by A. Cropper and R. Morel [Cropper and Morel, 2021a]. As input, Hopper 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 [Raedt, 2008].

During the generate phase, candidate programs are chosen from the viable hypothesis space, i.e. the space of programs that 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.

Hopper 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 [Cropper and Morel, 2021a]). Optimal is defined as the solution containing the fewest literals [Cropper and Morel, 2021a].

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, Hopper introduces constraints pruning specializations and/or generalizations of the candidate program. Specialization/generalization is defined via \(\Theta\)-subsumption [Plotkin, 1970; Reynolds, 1970]. Hopper may also introduce elimination Constraints pruning separable\textsuperscript{5} sets of clauses. Details concerning the benefits of this approach are presented in [Cropper and Morel, 2021a]. Essentially, Hopper refines the hypothesis space, not the program [Srinivasan, 2001; Muggleton, 1995; Quinlan and Cameron-Jones, 1993].

In addition to constraints introduced during the search, like the majority of ILP systems, Hopper incorporates a form of language bias [Nienhuys-Cheng et al., 1997], that is predefined syntactic and/or semantic restrictions of the hypothesis space. Hopper 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. Hopper accepts mode declaration-like hypothesis constraints [Mug-

\footnote{\textsuperscript{5}No head literal of a clause in the set occurs as a body literal of a clause in the set.}
We provide a brief overview of logic programming. Our exposition is far from comprehensive. We refer the interested reader to a more detailed source [Lloyd, 1987].

3 Theoretical Framework

We describe a brief overview of logic programming. Our exposition is far from comprehensive. We refer the interested reader to a more detailed source [Lloyd, 1987].

3.1 Preliminaries

Let \( P \) be a countable set of predicate symbols (denoted by \( p, q, r, p_1, q_1, \cdots \)), \( V_f \) be a countable set of first-order (FO) variables (denoted by \( A, B, C, \cdots \)), and \( V_h \) be a countable set of higher-order variables (denoted by \( P, Q, R, \cdots \)). Let \( T \) denote the set of FO terms constructed from a finite set of function symbols and \( V_f \) (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 \( \mathcal{sy}(a) = p \) is the symbol of the atom, \( \mathcal{ag}(a) = \{ T_1, \cdots, T_m \} \) are its FO-arguments, and \( \mathcal{ag}_h(a) = \{ t_1, \cdots, t_n \} \) are its FO-arguments. When \( \mathcal{ag}_h(a) = \emptyset \) and \( \mathcal{sy}(a) \in P \) we refer to \( a \) as FO, when \( \mathcal{ag}_h(a) \in P \) and \( \mathcal{sy}(a) \in P \) 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.

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 \( h(c) \)), while the set of atoms of negated literals is referred to as the body (denoted by \( b(c) \)). 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 (denoted by \( \theta \)) to \( \{ P_1 \mapsto p_1, \cdots, P_n \mapsto p_n, A_1 \mapsto t_1, \cdots, A_m \mapsto t_m \} \). A substitution \( \theta \) unifies two atoms when \( a\theta = b\theta \).

3.2 Interpretable Theories and Groundings

Our hypothesis space consists of a particular type of theory which we refer to as interpretable. From these theories, one can derive so-called, principle programs, FO clausal theories encoding the relationship between certain literals and clauses and a set of higher-order definitions. Popper generates and tests principal programs. This encoding preserves the soundness of the pruning mechanism presented in [Cropper and Morel, 2021a]. Intuitively, the soundness follows from each principal program encoding a unique HO program. A consequence of this approach is that each HO program may be encoded by multiple principal programs, some of which may not be in a subsumption relation to each other, i.e. not mutually prunable. This results in a larger, though more expressive hypothesis space.

Definition 3 A clause \( c \) is proper\(^7\) if \( \mathcal{ag}_h(h(c)) \) are pairwise distinct, \( \mathcal{ag}_h(h(c)) \subset V_h \), and \( \forall v \in b(c) \).

Definition 4 A f.f.d theory \( \Sigma \) is interpretable if \( \forall c \in \Sigma \), \( \mathcal{ag}_h(h(c)) = \emptyset \) and \( \forall l \in b(c) \), \( l \) is higher-order ground,

\[ \begin{align*}
& a) \text{ if } \mathcal{ag}_h(l) \neq \emptyset, \text{ then } \exists v' \in \Sigma, \mathcal{sy}(h(c)) \neq \mathcal{sy}(l), \text{ and} \\
& b) \forall p \in \mathcal{ag}_h(l), \exists c' \in \Sigma, \text{s.t. } \mathcal{sy}(h(c')) = p \in \mathcal{P}_{f_1}.
\end{align*} \]

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

Let \( \mathcal{S}_{f_1}(\Sigma) = \{ p_i | p_i \in \mathcal{ag}_h(a) \land a \in \mathcal{ex}(\Sigma) \} \), the set of predicates which need to be invented. During the generate phase we enforce invention of \( \mathcal{S}_{f_1}(\Sigma) \) 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 [Cropper and Morel, 2021a]. The HO arguments of external literals are ignored by the ASP solver, which searches for so-called principal programs (an FO representation of interpretable theories).

Example 1 Consider reverse, half, and issubtree of Section 2.1 & 2.2. Each is an interpretable theory. The sets of external literals of these theories are \( \{ \text{fold}(p, C, A, B), \{ \text{case}_{l_1}(p_1), p_1, \text{case}_{l_2}(p_2), p_2, C \}, \{ \text{case}_{l_3}(p_3), p_3, D, E \}, \{ \text{any}(C, B) \} \} \), respectively.

Definition 5 Let \( L \) be a library, and \( \Sigma \) an interpretable theory. \( \Sigma \) is L-compatible if \( \forall l \in \mathcal{ex}(\Sigma), \exists d \in L, \text{s.t. } \mathcal{hd}(d) = \sigma = l \) for some substitution \( \sigma \). Let \( \text{df}(L, l) = d \) and \( \theta(L, l) = \sigma \).

Example 2 The program in Section 2.1 is L-compatible with the following library \( L = \)

\[
\begin{align*}
\text{fold}(P, A, B, C) & : \text{empty}(B), C = A. \\
\text{fold}(P, A, B, C) & : \text{head}(B, H), P(A, H, D). \\
\text{tail}(B, T), \text{fold}(P, D, T, C).
\end{align*}
\]

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

An L-compatible theory \( \Sigma \) can be L-grounded. This requires replacing external literals of \( \Sigma \) by FO literals, i.e. removal of all HO arguments and replacing the predicate symbol

\^6\( \mathcal{sy}(l), \mathcal{ag}_h(l), \) and \( \mathcal{ag}_f(l) \) apply to literals with similar affect.

\^7Similar to definitional HO of W. Wadge [Wadge, 1991].
of the external literals with fresh predicate symbols, resulting in \(T^*\), and adding clauses that associate the FO literals \(l\) with the appropriate \(d \in L\) and argument instantiations. Different occurrences of external literals with the same symbol and same HO arguments result in FO literals with the same predicate symbol. The principal program contains all clauses derived from \(T\), i.e. \(T^*\) (See Example 3).

Example 3 Using the library of Example 2 and a modified version of the program from Section 2.1 (\(p\) is replaced by \(fold_{p,a}\) for clarity purposes), we get the following \(L\)-grounding:

\[
\begin{align*}
\text{reverse}(A, B) & : \text{empty}(C), \text{fold}_a(C, A, B). \\
\text{fold}_{p,a}(A, B, C) & : \text{head}(C, B), \text{tail}(A, C). \\
\text{fold}(P, A, B, C) & : \text{empty}(B), A = C. \\
\text{fold}(P, A, B, C) & : \text{head}(B, H), P(H, D), \\
\text{tail}(B, T) & : \text{fold}(P, D, T, C).
\end{align*}
\]

\(\text{fold}_a(C, A, B)\) replaces \(\text{fold}(\text{fold}_{p,a}(C, A, B))\). The first two clauses form the principal program.

If an \(L\)-compatible theory contains multiple external literals whose symbol is \(fold\), i.e. \(\text{fold}(\text{fold}_{p,a}, C, A, B)\) and \(\text{fold}(\text{fold}_{p,a}, D, E, R)\), both are renamed to \(\text{fold}_a\). However, if the higher-order arguments differ, i.e. \(\text{fold}_{p,a}\) and \(\text{fold}_{p,b}\), then they are renamed to \(\text{fold}_a\) and \(\text{fold}_b\), and an additional clause \(\text{fold}_{b}(A, B, C) : \text{fold}(\text{fold}_{p,b}, A, B, C)\) would be added to the \(L\)-grounding. When a definition takes more than one HO argument and arguments of instances partially overlap, duplicating clauses may be required during the construction of the \(L\)-grounding. Soundness of the pruning mechanism is preserved because the FO literals uniquely depend on the arguments fed to HO definitions.

Note, the system requires the user to provide higher-order definitions, similar to MetaHol90. Additionally, these HO definitions may be of the form \(\text{ho}(P, Q, x, y) : P(Q, x, y)\), essentially a higher-order definition template. While allowed by the formalism, we have not thoroughly investigated such constructions. This amounts to the invention of HO definitions.

3.3 Interpretable Theories and Constraints

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

Definition 6 (\(\Theta\)-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 \(\exists b \text{ s.t. } c_1 b \leq c_2\).

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 7 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 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 5). 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 [Cropper and Morel, 2021a]. Instead we query the ASP solver for the principal program. The definitions from the library \(L\) are treated as \(BK\). Consider Example 3, during the generate phase the ASP solver may return an encoding of the following clauses:

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

During the test phase the rest of the \(L\)-grounding is re-introduced. 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. This is similar to the dependency graph introduced in [Cropper and Morel, 2021b].

3.4 Negation, Generalization, and Specialization

Negation (under classical semantics) of HO literals can interfere with Popper constraints. Consider the ILP task and candidate programs:

\[
\begin{align*}
E^+ & : f(b). \quad f(c). \\
E^- & : f(a). \\
BK & : \{ p(a), \quad p(b), \quad q(a), \quad q(c) \}. \quad \text{HO : } N(P, A):= \neg P(A). \\
\text{prog}_a & : f(A):= N(p_1, A). \\
p_2(A):= p(A). \\
\text{prog}_a & \quad \text{prog}_a
\end{align*}
\]

The optimal solution is \(\text{prog}_a\). \(\text{prog}_a\) is an incorrect hypothesis which Popper can generate prior to \(\text{prog}_a\) and \(\text{prog}_a \leq_\Theta \text{prog}_a\). Note, \(\text{prog}_a \vdash \neg f(b) \land \neg f(a) \land f(c)\); it does not entail all of \(E^+\). We should generalize \(\text{prog}_a\) to find a solution, i.e. add literals to \(p_1\). The introduced constraints [Cropper and Morel, 2021a] prune programs extending \(p_1\), i.e. \(\text{prog}_a\). Similar holds for specializations. Consider the ILP task and candidate programs:

\[
\begin{align*}
E^+ & : f(a). \quad f(b). \\
E^- & : f(c). \quad f(d). \\
BK & : \{ p(d), \quad q(c) \}. \quad \text{HO : } N(P, X):= \neg P(X). \\
\text{prog}_a & : f(A):= N(p_1, A). \\
p_2(A):= p(A). \\
\text{prog}_a & \quad \text{prog}_a
\end{align*}
\]

The optimal solution is \(\text{prog}_a\). \(\text{prog}_a\) is an incorrect hypothesis which Popper can generate prior to \(\text{prog}_a\), and
For each task, we guarantee that the optimal solution is present in the hypothesis space and record how long Popper and Hopper take to find it. We ran Popper using optimal settings and minimal BK. In some cases, the tasks cannot be solved by Popper without Predicate Invention (See Column PI of Table 1), i.e. an explanatory hypothesis which is both accurate and precise requires auxiliary concepts.

We ran Hopper in two modes, Column Hopper concerns running Hopper with the same settings and a superset of the BK used by Popper (minus constructions used to force invention), while Column Hopper (Opt) concerns running Hopper with optimal settings and minimal BK. For Popper and Hopper, settings such as max_var significantly impact performance. Both systems search for the shortest program (by literal count) respecting the current constraints. Note, that hypothesizing a program with an additional clause w.r.t the previously generated programs requires increasing the literal count by at least two. Thus, the current search procedure avoids such hypotheses until all shorter programs have been pruned or tested. The parameters max_var and max_body have a significant impact on the size of the single clause hypothesis space. Given that the use of HO definitions always requires auxiliary clauses, using large values, for the above-mentioned parameter, will hinder their use. This is why Hopper performs significantly better post-optimization. Using a comparably large BK incurs an insignificant performance impact compared to using unintuitively large parameter settings (see Proposition 1, page 14 (Cropper and Morel, 2021a))

The predicates used for a particular task are listed in Column HO-Predicate of Table 1. Popper and Hopper timed out (300 seconds elapsed) when large clauses or many variables are required. Timing out means the optimal solution was not found in 300 seconds. Given that we know the solution to each task, Column #Literals provides the size of the known solution, not the size of the non-optimal solution found by the system in case of timeout.

Concerning the Optimizations, these runs of Hopper closely emulate how such a system would be used as it is pragmatic to search assuming smaller clauses and fewer variables are sufficient and expand the space as needed. Popper’s and Hopper’s performance degrades by just assuming the solution may be complex. For findDup, Hopper found the FO solution.

Overall, this optimization issue raises a question concerning the search mechanism currently used by both Popper and Hopper. While HO solutions are typically shorter than the corresponding FO solution, this brevity comes at the cost of a complex program structure. This trade-off is not considered by the current implementations of the LFF paradigm. Investigating alternative search mechanisms and optimality conditions (other than literal count) is planned future work.

We attempted to solve each task using MetagolHO. Successful learning using MetagolHO is highly dependent on the choice of the metarules. To simplify matters, we chose metarules that mimic the clauses found in the solution. In some cases, this requires explicitly limiting how certain variables are instantiated by adding declarations, i.e. meta1: type(Q, 2, head_pred), to the body of a metarule (denoted by metatype in Table 1). This amounts to significant human guidance, and thus, both simplifies learning and what
we can say comparatively about the system. Hence we limit ourselves to indicating success or failure only.

Under these experimental settings, every successful task was solved faster by MetagolHO than Hopper with optimal settings. Relaxing restrictions on the metarule set introduces a new variable into the experiments. Choosing a set of metarules that is general and covers every task will likely result in the failure of the majority of tasks. Some tasks require splitting rules such as $P(A;B):=Q(A,B)$, $R(A,B)$ which significantly increase the size of the hypothesis space. Choosing metarules per task, but without optimizing for success, leaves the question, which metarules are appropriate/acceptable for the given task? While this is an interesting question [Cropper and Tourret, 2020], the existence problem of a set of general metarules over which MetagolHO’s performance is comparable to, or even better than, Hopper’s only strengthens our argument concerning the chosen experimental setting as one will have to deduce/design this set.

### 5 Conclusion and Future Work

We provide a theoretical framework encapsulating the accepted HO definitions, a fragment of the definitions monotone over subsumption and entailment, and discuss the limitations of this framework. A detailed account is provided in Section 3. The main limitation of this framework concerns HO-negation which we leave to future work. Our framework also allows for the invention of HO predicates during learning through constructions of the form $\text{ho}(P, Q, x, y):= P(Q, x, y)$. We can verify that Hopper can, in principal, finds the solution, but we have not successfully invent an HO predicate during learning. We plan for further investigation of this problem.

As briefly mentioned, Hopper was tested twice during experimentation due to the significant impact system parameters have on its performance. This can be seen as an artifact of the current implementation of LFF which is bias towards programs with fewer, but longer, clauses rather than programs with many short clauses. An alternative implementation of LFF taking this bias into account, together with other bias, is left to future investigation.

### Acknowledgements

We would like to thank Rolf Morel and Andrew Cropper for their thorough commentary which helped us greatly improve a preliminary version of this paper. Supported by the ERC starting grant no. 714034 SMART, the MathILP project (LIT-2019-7-YOU-213) of the Linz Institute of Technology and the state of Upper Austria, Cost action CA20111 EuroProofNet, and project CZ.02.2.69/0.0/0.0/18_053/00017594 of the Ministry of Education, Youth and Sports of the Czech Republic.
References


