Learning Higher-Order Logic Programs From Failures

Stanislaw J. Purgal1, 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.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. Our theoretical framework captures a class of higher-order definitions preserving soundness of existing subsumption-based pruning methods.

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 lists 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{verbatim}
allSeqN(N,L):- iterate(succ, 0, N, A), map(p, A, L).
   p(A,B):- iterate(succ, 0, A, B).
\end{verbatim}

The first iterate$^1$ 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 toward short solutions, with the ability to use HO definitions can result in improved performance. We developed an HO-enabled Popper [CM20a] (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 on 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 (w.r.t. FO arguments), poses a challenge for some systems, and in the case of HEXMIL$^\text{HO}$ [CMM20], 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 [CM21a], 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 definitions into MIIL-based systems is quite seamless as HO definitions are essentially a special type of metarule. Thus, HO enabling MIIL 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 (See page 813 of [CM21a]).

We avoid these soundness issues by indirectly adding HO definitions. Popper 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 and entailment; \( p_1 \subseteq_0 (\vdash) p_2 \Rightarrow H(p_1) \subseteq_0 (\vdash) H(p_2) \) 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). 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 [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. 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 HEMIL [KEI18]. Also, we introduce Popper [CM21a], the system Popper is based on. For interested readers, 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 fo1d/4 can be used together with PI to provide a succinct program for reversing a list:

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

\[
p(A, B, C) :- \text{head}(C, B), \text{tail}(C, A).
\]

Including \( p \) in the background knowledge is unintuitive. It is reasonable to expect the synthesizer to produce it. Many of the well known, non-MIIL based ILP frameworks do not support predicate invention, Foil [Qui90], Prolog [Mug95], Tilde [Bli99], and Aleph [Str01] 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 \( \delta \)ILP [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 HEMIL [KEI18]; both are based on Meta-interpretive Learning (MiIL) [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 Meta-rules [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\(^2\) (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 MIIL-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 HEMIL

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

Higher-order Metagol

In short, Metagol is a MIIL-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 \( BK_c \), 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 of 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\(_{HO}^2\) is the inclusion of interpreted background knowledge \( BK_{in} \). For example, \text{map}/3 as \( BK_{in} \) takes the form:

\[
\begin{align*}
\text{ibk}((\text{map}, [1], [1], [1])). \\
\text{ibk}((\text{map}, [A|\text{As}], [B|\text{Bs}], F), ([F, A, B], [\text{map}, \text{As}, \text{Bs}, F])).
\end{align*}
\]

Metagol handles \( BK_{in} \) as it handles metarules. When used, Metagol attempts to prove the body of \text{map}, i.e. \( F(A, B) \). Either \( F \) is substituted by a predicate contained in \( BK_c \) or

\( ^2 \)\( p(A, B) : - q(A, C), r(C, B) \).
replaced by an invented predicate that becomes the goal atom and is proven using metarules or BK

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

\[
\text{half\text{last}}(A,B) :- \text{reverse}(A,C), \\
\text{case\text{last}}(p_1, p_{[H|T]}, C, B). \\
p_{[1]}(A) :- \text{empty}(A). \\
p_{[H|T]}(A, B, C) :- \text{empty}(B), \text{empty}(C). \\
p_{[H|T]}(A, B, C) :- \text{front}(B, D), D, E. \\
\text{append}(E, A, C).
\]

The HO predicate case\text{last}($p_1, p_{[H|T]}, A, B$) calls $p_{[1]}$ if $A$ is empty and $p_{[H|T]}$ otherwise. Our definition of half$\text{last}$($A$,$B$) cannot be found using the standard search procedure as every occurrence of case\text{last} results in a call to the meta-interpreter's proof procedure. The underlined call to case\text{last} results in PI for $p_{[H|T]}$ ad infinitum. Similarly, the initial goal cannot be substituted unless it's explicitly specified.

As with half$\text{last}$($A$,$B$), the following program defining issubtree($A$,$B$), which computes whether $B$ is a subtree of $A$, requires recursively calling issubtree through any

\[
is\text{subtree}(A,B) :- A = B. \\
is\text{subtree}(A,B) :- \text{children}(A,C), \text{any}(\text{cond}, C, B). \\
\text{cond}(A,B) :- A = B.
\]

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.

Negation of invented predicates (HO arguments of BK definitions), to the best of our knowledge, is not fully supported by Metagol$_{HO}$ (See Section 4.2 of [CMM20]). Popper 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].

HEXMIL is restricted to forward-chained metarules:

**Definition 2** Forward-chained metarules are of the form:

\[
P(A,B) :- Q_1(A,C_1), Q_2(C_1, C_2), \ldots, Q_n(C_{n-1}, B), R_1(D_1), \ldots, R_m(D_m) \text{ where } D_i \in \{A, C_1, \ldots, C_{n-1}, B\}.
\]

Thus, only Dyadic learning task may be handled. Furthermore, many useful metarules are not of this form, i.e. $P(A,B) :- Q(A,B), R(A,B)$. HEXMIL$_{HO}$ incorporates HO definitions into the forward-chained structure of Definition 2. For details concerning the encoding see Section 4.4 of [CMM20]. The authors of [CMM20] illustrated HEXMIL$_{HO}$'s poor performance on list manipulation tasks and its limitations make application to tasks of interest difficult. Thus, we focus on Metagol$_{HO}$ in Section 4.

**2.3 Popper: Learning From Failures (LFF)**

The LFF paradigm together with Popper provides 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. Cropp 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 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.

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 [Pl67, Rey70]. Popper may also introduce elimination Constraints pruning separable sets of clauses. Details concerning the benefits of this approach are presented in [CM21a]. Essentially, Popper refines the hypothesis space, not the program [Sp01, Mu95, QC93].

In addition to constraints introduced during the search, like the majority of ILP systems, Popper incorporates a form of language bias [NCW97], 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 [Mu95] 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 [Li17] paradigm. The language bias together with the

\footnote{No head literal of a clause in the set occurs as a body literal of a clause in the set.}
generated constraints is encoded as an ASP program. The ASP 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 $p, q, r, p_1, q_1, \ldots$), $\mathcal{V}_f$ 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 $\text{sy}(a) = \theta$ is the symbol of the atom, $a_1$ is its HO-arguments, and $a_2$ is its FO-arguments. When $a_1 = \emptyset$ and $\text{sy}(a) \in \mathcal{P}$ we refer to $a$ as a FO, when $a_1 \subset \mathcal{P}$ and $\text{sy}(a) \in \mathcal{P}$ we refer to $a$ as a HO-ground, otherwise it is HO. A literal is either an atom or its negation.

A 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 $\text{hd}(c)$), while the set of atoms of negated literals is referred to as the body (denoted by $\text{bd}(c)$). A function-free definite (f.f.d) clause only contains variables as FO arguments. We refer to a clause body as the body of the clause and a set of higher-order definitions.

A definite clause $c$ is of the form $p, q, r, p_1, q_1, \ldots$. 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 clauseal theories encoding the relationship between certain literals and clauses and a set of higher-order definitions. Hopper generates and tests principal programs. This encoding preserves the soundness of the pruning mechanism presented in [CM21a]. 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 if $\text{ag}_h(\text{hd}(c))$ are pairwise distinct, $\text{ag}_h(\text{hd}(c)) \subset \mathcal{V}_h$, and $\forall a \in \text{bd}(c)$.\footnote{Similar to definitional HO of W. Wadge [Wad91].}

A finite set of proper clauses $d$ with the same head (denoted $\text{hd}(d)$) is referred to as a HO definition. A set of distinct HO definitions is a library. Let $\mathcal{P}_1 \subset \mathcal{P}$ be a set of predicate symbols reserved for invented predicates.

Definition 4 A f.f.d theory $\mathcal{T}$ is interpretable if $\forall c \in \mathcal{T}, \text{ag}_h(\text{hd}(c)) = \emptyset$ and $\forall l \in \text{bd}(c)$, $l$ is higher-order ground,

a) if $\text{ag}_h(l) \neq \emptyset$, then $\text{sy}(c') \neq \text{sy}(l)$,
b) $\forall p \in \text{ag}_h(l), \exists c' \in \mathcal{T}, \text{sy}(hd(c')) = p \in \mathcal{P}_1$.

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

Let $\mathcal{S}_1 = \{ p_i | p_i \in \text{ag}_h(l) \wedge a \in \text{ex}(\mathcal{T}) \}$, the set of predicates which need to be invented. During the generate phase we enforce invention of $\mathcal{S}_1$ 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 principal programs (an FO representation of interpretable theories).

Example 1 Consider reverse, half, isstartswith of Section 2.1 & 2.2. Each is an interpretable theory. The sets of external literals of these theories are $\{ \text{fold}(p, C, A, B) \}$, $\{ \text{case1start}(P_1, P_{\mid H\mid T}, C, B), \text{case1end}(P_1, P_{\mid H\mid T}, D, E) \}$, and $\{ \text{any}(\text{cond}, C) \}$, respectively.

Definition 5 Let $L$ be a library, and $\mathcal{T}$ an interpretable theory, $\mathcal{T}$ is $L$-compatible if $\exists l \in \text{ex}(\mathcal{T}), \exists l \in L$. s.t. $\text{hd}(d) = l$ for some substitution $\theta$. 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 $\mathcal{T}$ can be $L$-grounded. This requires replacing external literals of $\mathcal{T}$ by FO literals, i.e. removal of all HO arguments and replacing the predicate symbol of the external literals with fresh predicate symbols, resulting in $\mathcal{T}^*$, 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 $\mathcal{T}$, i.e. $\mathcal{T}^*$ (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:

| reverse(A, B) :- empty(C), fold_{a}(C, A, B).  
| fold_{p,a}(A, B, C) :- head(C, B), tail(C, A).  
| fold_{a}(A, B, C) :- fold(fold_{p,a}, A, B, C).  
| fold(P, A, B, C) :- empty(B), A = C.  
| tail(B, T), fold(P, D, T, C). |

The first two clauses form the principal program. Note that the external literal fold(fold_{p,a}, C, A, B) is replaced by fold_{a}(C, A, B).

If an L-compatible theory contains multiple external literals whose symbol is fold, i.e. fold(fold_{p,a}, C, A, B) and fold(fold_{p,b}, D, E, R), both are renamed to fold_{a}. However, if the higher-order arguments differ, i.e. fold_{p,a} and fold_{p,b}, then they are renamed to fold_{a} and fold_{b}, and an additional clause fold_{a}(A, B, C) :- fold(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 to HO definitions.

Note, the system requires the user to provide higher-order definitions, similar to MetagolH0. Additionally, these HO definitions may be of the form ho(P, Q, x, y) :- P(Q, x, y), essentially a higher-order definition template. While this is allowed by the formalism, we have not thoroughly investigated such constructions. This would amount to the invention of HO definitions.

3.3 Interpretable Theories and Constraints

The constraints of Section 2.3 are based on Θ-subsumption.

Definition 6 (Θ-subsumption) An FO theory T_1 subsumes an FO theory T_2, denoted by T_1 ≤_Θ T_2 iff ∀c_2 ∈ T_2∃c_1 ∈ T_1 s.t. c_1 ≤_Θ c_2, where c_1 ≤_Θ c_2 iff ∃θ s.t. c_1 θ ⊆ c_2.

Importantly, the following property holds:

Proposition 1 if T_1 ≤_Θ T_2, then T_1 |= 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 ≤_Θ T_2 (T_2 ≤_Θ T_1).

Given a library L and a 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 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 [CM21a]. 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:

| reverse(A, B) :- empty(C), fold(C, A, B).  
| p(A, B, C) :- head(C, B), tail(C, A). |

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 [CM21b].

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:

\[
E^+ : f(b), \quad f(c).  
\]

\[
E^- : f(a).  
\]

\[
BK : \begin{cases}  
p(a), \quad p(b), \quad q(a) 
\end{cases}  
\]

\[
HO : N(P, A) :\neg P(A).  
\]

\[
\begin{array}{ll}
prog_p & f(A) :\neg N(p_1, A), 
\quad p_1(A) : p(A), 
\quad q(A).
\end{array}  
\]

\[
\begin{array}{ll}
prog & f(A) :\neg N(p_1, A), 
\quad p_1(A) : p(A).
\end{array}  
\]

The optimal solution is prog_p, prog is an incorrect hypothesis which Popper can generate prior to prog_p, and prog ≤_Θ prog_p. Note, prog_p |= \neg f(b) \land \neg f(a) \land f(c), it does not entail all of E^+. We should generalize prog_p to find a solution, i.e. add literals to p_1. The introduced constraints [CM21a] prune programs extending p_1, i.e. prog_p. Similar holds for specializations. Consider the ILP task and candidate programs:

\[
E^+ : f(a), \quad f(b), \quad f(c),  
\]

\[
E^- : f(c).  
\]

\[
BK : \{ p(d), q(c). \}  
\]

\[
HO : N(P, X) :\neg P(X).  
\]

\[
\begin{array}{ll}
prog_p & f(A) :\neg N(p_1, A),  
\quad p_1(A) : p(A),  
\quad q(A).  
\end{array}  
\]

\[
\begin{array}{ll}
prog & f(A) :\neg N(p_1, A),  
\quad p_1(A) : p(A).  
\end{array}  
\]

The optimal solution is prog_p, prog is an incorrect hypothesis which Popper can generate prior to prog_p, and prog ≤_Θ prog_p. Note, prog_p |= 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 prog_p. The introduced constraints [CM21a] prune programs that add clauses, i.e. prog_p.

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

The pruning mechanism will never prune the shortest solution. Thus, the important question when evaluating Popper, and 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 significantly outperforms Enumerate and existing ILP systems.

One way to improve the performance of LFF-based ILP systems, like Popper, is to introduce techniques that shorten or simplify the solution. The authors of [CM20], in addition to introducing Metagol$_{HO}$ and HEXMIL$_{HO}$, provided a comprehensive suite of experiments illustrating that the addition of HO predicates can improve accuracy and, most importantly, reduce learning time. Reduction in learning times results from HO predicates reducing the complexity/size of the solutions.

The experiments in [CM21a] thoroughly cover scalability issues and learning performance on simple list transformation tasks, but do not cover performance on complex tasks with large solutions. The experiments presented in [CM20] illustrate performance gains when a HO library is used to solve many simple tasks and how the addition of HO predicates allows 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 a set of metarules for Metagol$_{HO}$ without being overly descriptive or suffering from long learning times.

Our experiments illustrate that combining Popper with HO predicates [CM20] significantly improves Popper’s performance at learning complex programs. As in [CM21a], we use predicate declarations, i.e. type decls to force in-body, type declarations, i.e. type(head, (list, element)), direction declarations, i.e. direction(head, (in, out)), and the parameters required by Popper’s search mechanism, max_var, max_body, and max_clauses.

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.

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. any explanatory hypothesis which is both accurate and precise requires auxiliary concepts not present in the BK.

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

### Table 1: We ran Popper, Hopper, optimized Hopper, and Metagol$_{HO}$ on a single core with a timeout of 300 seconds. 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.

<table>
<thead>
<tr>
<th>Task</th>
<th>Popper (Opt)</th>
<th>#Literals</th>
<th>PT?</th>
<th>Hopper</th>
<th>Hopper (Opt)</th>
<th>#Literals</th>
<th>HO-Predicates</th>
<th>Metagol$_{HO}$</th>
<th>Metatypes?</th>
</tr>
</thead>
<tbody>
<tr>
<td>dropK</td>
<td>1.1s</td>
<td>7</td>
<td>no</td>
<td>0.5s</td>
<td>0.1s</td>
<td>4</td>
<td>iterate</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>allEven</td>
<td>0.2s</td>
<td>7</td>
<td>no</td>
<td>0.2s</td>
<td>0.1s</td>
<td>4</td>
<td>all</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>findDup</td>
<td>0.25s</td>
<td>7</td>
<td>no</td>
<td>--</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>0.1s</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>0.2s</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>46.3s</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>7.7s</td>
<td>0.5s</td>
<td>6</td>
<td>fold</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task</th>
<th>Popper (Opt)</th>
<th>#Literals</th>
<th>PT?</th>
<th>Hopper</th>
<th>Hopper (Opt)</th>
<th>#Literals</th>
<th>HO-Predicates</th>
<th>Metagol$_{HO}$</th>
<th>Metatypes?</th>
</tr>
</thead>
<tbody>
<tr>
<td>dropLastK</td>
<td>300.0s</td>
<td>10</td>
<td>no</td>
<td>300s</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>300s</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 (Opt)</th>
<th>#Literals</th>
<th>PT?</th>
<th>Hopper</th>
<th>Hopper (Opt)</th>
<th>#Literals</th>
<th>HO-Predicates</th>
<th>Metagol$_{HO}$</th>
<th>Metatypes?</th>
</tr>
</thead>
<tbody>
<tr>
<td>repeatN</td>
<td>5.0s</td>
<td>7</td>
<td>no</td>
<td>0.6s</td>
<td>0.1s</td>
<td>5</td>
<td>iterate</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>rotateN</td>
<td>300.0s</td>
<td>10</td>
<td>no</td>
<td>300s</td>
<td>2.6s</td>
<td>6</td>
<td>iterate</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>allSeqN</td>
<td>300.0s</td>
<td>25</td>
<td>yes</td>
<td>300s</td>
<td>5.0s</td>
<td>9</td>
<td>iterate, map</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>dropLastK</td>
<td>300.0s</td>
<td>17</td>
<td>yes</td>
<td>300s</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>300s</td>
<td>0.2s</td>
<td>9</td>
<td>iterateStep</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>lastHalf</td>
<td>300.0s</td>
<td>12</td>
<td>no</td>
<td>300s</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>300s</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>157s</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>300s</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>300s</td>
<td>25.9s</td>
<td>12</td>
<td>caseTree, any</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>isSubTree</td>
<td>2.9s</td>
<td>11</td>
<td>yes</td>
<td>1.0s</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>300s</td>
<td>1.4s</td>
<td>9</td>
<td>map, caselnT</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>mulFromSuc</td>
<td>300.0s</td>
<td>19</td>
<td>yes</td>
<td>300s</td>
<td>1.2s</td>
<td>7</td>
<td>iterate</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

4 Experiments

We ran Popper, Hopper, optimized Hopper, and Metagol$_{HO}$ on a single core with a timeout of 300 seconds. 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.
A metatule defines a relation with a fixed number of arguments, and the body of a metatule is a clause and can be either literal or another metatule. The body of a metatule can be further expanded by metarules, which are defined in the body of the metatule. Metarules can be used to build a hypothesis space for a task, and the hypothesis space can be further expanded by metarules.

We attempted to solve each task using Metagol\textsubscript{HO}. Successful learning using Metagol\textsubscript{HO} 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. `metagol:type(\text{\text{Q}}, 2, \text{\text{head\_pred}})`, to the body of a metatule (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 Metagol\textsubscript{HO} 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 [CT20], the existence problem of a set of general metarules over which Metagol\textsubscript{HO}’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 extended the LFF-based ILP system Popper to effectively use user-provided HO definitions during learning. Our experiments show Hopper (when optimized) is capable of outperforming Popper on most tasks, especially the harder tasks we introduced in this work (Section 4). Hopper requires minimal guidance compared to the top-performing MiL-based ILP system Metagol\textsubscript{HO}. Our experiments test the theoretical possibility of Metagol\textsubscript{HO} finding a solution under significant guidance. However, given the sensitivity to metarule choice and the fact that many tasks have ternary and even 4-ary predicates, it is hard to properly compare these approaches.

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 \(ho(P, Q, x, y) :: P(Q, x, y)\). We can verify that Hopper can, in principal, finds the solution, but we have not managed to 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.

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 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 \( S_{P1}(\Sigma) \) 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

\[
\text{map}(P, [\_], [\_]).
\]
\[
\text{map}(P, [H_1|T_1], [H_2|T_2]) :- P(H_1, H_2), \text{map}(P, T_1, T_2).
\]

we automatically generate a more contextual version

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

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

\[
f(A, B, C) :- \text{map}(p_1, B, C, A).
\]
\[
p_1(A, B, C) :- \text{add}(A, C, B).
\]

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

The argument predicate is true for all elements of the list

\[
\text{all}(P, []).\]
\[
\text{all}(P, [H|T]) :- \text{call}(P, H), \text{all}(P, T).
\]

any/3

checks if for some element in a list and an input object if provided to a given predicate, then that predicate holds

\[
\text{any}(P, [H_1], B) :- \text{call}(P, H, B).
\]
\[
\text{any}(P, [], B) :- \text{any}(P, T, B).
\]

caseList/4

a deconstructor for a list, calling first or second predicate depending on whether the list is empty

\[
\text{caseList}(P, [], B) :- \text{call}(P, B).
\]
\[
\text{caseList}(P, [H|T], B) :- \text{call}(Q, H, T, B).
\]

caseList/5

a deconstructor for a list, calling the first, second, third predicate depending on whether the list is empty or singleton. Takes an additional argument which is passed to the predicate arguments.

\[
\text{caseList}(P, [], B) :- \text{call}(P, B).
\]
\[
\text{caseList}(P, [H], B) :- \text{call}(Q, H, B).
\]
\[
\text{caseList}(P, [H|T], B) :- \text{call}(R, H, T, B).
\]

caseTree/4

a deconstructor for a tree, calling first or second predicate depending on whether the tree is a leaf

\[
\text{caseTree}(P, [\_ t(R, [])], A) :- \text{call}(P, R, A).
\]
\[
\text{caseTree}(P, [\_ t(R, [H|T]), A]) :- \text{call}(Q, R, [H|T], A).
\]

caseInt/4

A deconstructor for a natural number, calling first or second predicate depending on whether the number is 0.

\[
\text{caseInt}(P, [], 0, X, Y) :- \text{call}(P, X, Y).
\]
\[
\text{caseInt}(P, [Q, N, B]) :- \text{less0}(N), \text{pred}(N, M), \text{call}(Q, M, X, Y).
\]
fold/4

combines all elements of a list using the argument predicate.

\[
\text{fold}(P, Acc, [H|T], Y) :- \text{call}(P, Acc, H, W), \text{fold}(P, W, T, Y).
\]
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.

\[
\begin{align*}
\text{map}(P, [], []). \\
\text{map}(P, [H_1[T_1], [H_2[T_2]]):- \text{call}(P, H_1, H_2), \text{map}(P, T_1, T_2).
\end{align*}
\]

try/3
checks whether at least one of argument predicates hold on the last argument

\[
\begin{align*}
\text{try}(P, \_ X):- \text{call}(P, X). \\
\text{try}(\_ Q, X):- \text{call}(Q, X).
\end{align*}
\]

condList/2
true if argument is an empty list, otherwise call argument predicate on the head and the tail.

\[
\begin{align*}
\text{condList}(P, []). \\
\text{condList}(P, [H[T]]):- \text{call}(P, H, T).
\end{align*}
\]

iterate/4
iterate the argument predicate \( n \) times

\[
\begin{align*}
\text{iterate}(\_, X, 0, X). \\
\text{iterate}(P, X, K, W):- \text{integer}(K), 0 < K, !, \\
\quad \text{pred}(K, R), \\
\quad \text{call}(P, X, H), \\
\quad \text{iterate}(P, H, R, W).
\end{align*}
\]

iterateStep/5
A variant of iterate/4, 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.

\[
\begin{align*}
\text{iterate}([\_, \_, 0, []]). \\
\text{iterate}(P, Q, X, K, [H[T]]):- \text{integer}(K) \\
\quad \text{call}(P, K, R), \\
\quad \text{call}(Q, X, H, W), \\
\quad \text{iterate}(P, Q, W, R, T).
\end{align*}
\]

C Learning tasks
In this section we provide the experimental set up for all learning task discussed above. For each task we provide the following information:

- \textbf{FO Background}: \( BK \) used for the first order learning task.
- \textbf{FO Parameters}: The values for the essential parameters of \textit{Popper} and any switches which were enabled during FO learning.
- \textbf{FO Solution}: The solution found by \textit{Popper} or the solution we expected \textit{Popper} to find if given enough time.

- \textbf{HO Optimal Background}: \( BK \) used for the Higher order learning task when ran in the optimal configuration. When ran in the non-optimal configuration we used the union of the FO \( BK \) with the HO optimal \( BK \) minus any HO predicates used for predicate invention during FO learning.
- \textbf{HO Optimal Parameters}: The values for the essential parameters of \textit{Hopper} and any switches which were enabled during HO optimal learning. In the non-optimal case we used the setting of the first order learning task minus switches activated specifically for predicate invention during the first order task.
- \textbf{HO Solution}: The solution found by \textit{Hopper} or the solution we expected \textit{Hopper} to find if given enough time.

C.1 dropK/3
drop first \( k \) elements from a list

- \textbf{FO Background}: \( \text{succ}, \text{pred}, \text{zero}, \text{tail}, \text{head}, \text{eq} \)
- \textbf{FO Parameters}:
  - \( \text{max-vars}: 5 \)
  - \( \text{max-body}: 3 \)
  - \( \text{max-clause}: 2 \)
  - \( \text{Switches}: \text{enable-recursion} \)
- \textbf{FO Solution}:

\[
\begin{align*}
\text{dropK}(A, B, C):- \text{zero}(A), \text{eq}(B, C). \\
\text{dropK}(A, B, C):- \text{tail}(B, D), \text{pred}(A, E), \text{dropK}(E, D, C).
\end{align*}
\]

- \textbf{HO Optimal Background}: \( \text{succ}, \text{zero}, \text{tail}, \text{head} \)
- \textbf{HO Optimal Parameters}:
  - \( \text{max-vars}: 3 \)
  - \( \text{max-body}: 2 \)
  - \( \text{max-clause}: 2 \)
  - \( \text{Switches}: \)
- \textbf{HO Solution}:

\[
\begin{align*}
\text{dropK}(A, B, C):- \text{iterate_a}(B, A, C). \\
\text{iterate_p_a}(A, B):- \text{tail}(A, B).
\end{align*}
\]

C.2 allEven/1
check whether all elements on a list are even

- \textbf{FO Background}: \( \text{succ}, \text{zero}, \text{even}, \text{tail}, \text{head}, \text{empty} \)
- \textbf{FO Parameters}:
  - \( \text{max-vars}: 3 \)
  - \( \text{max-body}: 4 \)
  - \( \text{max-clause}: 2 \)
  - \( \text{Switches}: \text{enable-recursion} \)
- \textbf{FO Solution}:
C.3 findDup/2

check whether an element is present on a list at least twice

- **FO Background**: member, head, tail, empty
- **FO Parameters**:
  - max_vars: 4
  - max_body: 4
  - max_clause: 2
- **Switches**: enable_recursion
- **FO Solution**:

  ```prolog
  findDup(A, B):- tail(A, C), empty(B, C), head(A, B).
  findDup(A, B):- tail(A, C), findDup(C, B).
  ```

- **HO Optimal Background**: empty, eq, member
- **HO Optimal Parameters**:
  - max_vars: 3
  - max_body: 2
  - max_clause: 4
- **Switches**: non_datalog, allow_singletons
- **HO Solution**:

  ```prolog
  findDup(A, B):- caseList.a(A, B).
  caseList_a(A, B):- empty(B, member(A, B)).
  caseList_q_a(A):- member(C, B), eq(C, A).
  caseList_q_a(A):- caseList.a(B, C).
  ```

C.4 length/2

Finds the length of a list.

- **FO Background**: head, tail, succ, empty, zero.
- **FO Parameters**:
  - max_vars: 4
  - max_body: 3
  - max_clause: 2
- **Switches**: enable_recursion

- **FO Solution**:

  ```prolog
  length(A, B):- empty(A), zero(B).
  length(A, B):- tail(A, C), length(C, D), suc(D, B).
  ```

- **HO Optimal Background**: succ, zero, fold.
- **HO Optimal Parameters**:
  - max_vars: 3
  - max_body: 2
  - max_clause: 2
- **Switches**: enable_recursion
- **HO Solution**:

  ```prolog
  length(A, B):- zero(C), fold_a(C, A, B).
  fold_p_a(A, B, C):- suc(A, C).
  ```

C.5 member/2

check whether an element is on a list

- **FO Background**: head, tail, empty.
- **FO Parameters**:
  - max_vars: 3
  - max_body: 2
  - max_clause: 2
- **Switches**: enable_recursion
- **FO Solution**:

  ```prolog
  member(A, B):- head(A, B).
  member(A, B):- tail(A, C), member(C, B).
  ```

- **HO Optimal Background**: head, tail, empty, any.
- **HO Optimal Parameters**:
  - max_vars: 2
  - max_body: 2
  - max_clause: 2
- **Switches**: enable_recursion
- **HO Solution**:

  ```prolog
  member(A, B):- any_a(A, B).
  any_p_a(A, B):- head(A, B).
  ```
C.6  sorted/1

Check whether a list is sorted (non-decreasing)

- FO Background: empty, geq, head, tail, suc, zero, pred.
- FO Solution:

\[
\text{sorted}(A) :- \text{tail}(A, B), \text{empty}(B).
\]
\[
\text{sorted}(A) :- \text{head}(A, D), \text{tail}(A, B),
\]
\[
\text{head}(B, C), \text{geq}(C, D),
\]
\[
\text{sorted}(B).
\]

- HO Background: empty, geq, head, tail, suc, zero, pred.
- HO Solution:

\[
\text{sorted}(A) :- \text{zero}(C), \text{fold}_a(C, A, B).
\]
\[
\text{fold}_p_a(A, B, C) :- \text{pred}(B, C), \text{geq}(C, A).
\]

C.7  reverse/2

Reverse a list

- FO Background: empty, app, head, tail.
- FO Parameters:
  - max_vars: 5
  - max_body: 5
  - max_clause: 2
  - Switches: enable_recursion
- FO Solution:

\[
\text{reverse}(A, B) :- \text{empty}(A), \text{empty}(B).
\]
\[
\text{reverse}(A, B) :- \text{head}(A, D), \text{tail}(A, E),
\]
\[
\text{reverse}(E, C), \text{app}(C, D, B).
\]

- HO optimal Background: empty, head, tail, fold.
- HO optimal Parameters:
  - max_vars: 5
  - max_body: 5
  - max_clause: 2
  - Switches: enable_recursion
- HO Solution:

\[
\text{reverse}(A, B) :- \text{empty}(C),
\]
\[
\text{fold}_a(C, A, B).
\]
\[
\text{fold}_p_a(A, B, C) :- \text{head}(C, B), \text{tail}(C, A).
\]

C.8  dropLast/2

given a list of lists, drop the last element from each list

- FO Background: con, tail, empty, reverse
- FO Parameters:
  - max_vars: 8
  - max_body: 6
  - max_clause: 2
  - Switches: enable_recursion
- FO Solution:

\[
\text{dropLast}(A, B) :- \text{empty}(A), \text{empty}(B).
\]
\[
\text{dropLast}(A, B) :- \text{con}(A, B, C), \text{reverse}(C, E),
\]
\[
\text{tail}(E, F), \text{reverse}(F, G),
\]
\[
\text{con}(B, G, H), \text{dropLast}(D, H).
\]

- HO optimal Background: reverse, tail, map
- HO optimal Parameters:
  - max_vars: 5
  - max_body: 3
  - max_clause: 3
  - Switches:
- HO Solution:

\[
\text{dropLast}(A, B) :- \text{map}(\text{map}_p_a, A, B).
\]
\[
\text{map}_p_a(A, B) :- \text{reverse}(A, C), \text{tail}(C, D),
\]
\[
\text{reverse}(D, B).
\]

C.9  encryption/2

convert characters to integers, add 2 to each of them, then convert them back

- FO Background: pred, char_to_int, int_to_char, head, con, tail, empty
- FO Parameters:
  - max_vars: 9
  - max_body: 10
  - max_clause: 2
  - Switches: enable_recursion
- FO Solution:

\[
\text{encryption}(A, B) :- \text{empty}(A), \text{empty}(B).
\]
\[
\text{encryption}(A, B) :- \text{head}(A, C), \text{head}(A, D),
\]
\[
\text{cons}(B, E, F),
\]
\[
\text{char}_\text{to}_\text{int}(C, G), \text{pred}(G, H),
\]
\[
\text{pred}(H, I), \text{int}_\text{to}_\text{char}(I, E),
\]
\[
\text{encryption}(D, F).
\]

- HO optimal Background: char_to_int, int_to_char, pred, map
- HO optimal Parameters:
C.10 repeatN/3
Construct a list made of the input list argument repeated n times.

- **FO Background**: empty, app1, zero, pred.
- **FO Parameters**:
  max_vars: 5
  max_body: 3
  max_clause: 2
  Switches: enable_recursion, non_datalog, allow_singletons

- **FO Solution**:
  \[
  \text{repeatN}(A, B, C) :- \text{empty}(C), \text{zero}(B).
  \text{repeatN}(A, B, C) :- \text{pred}(B, E), \text{repeatN}(A, E, D),
  \text{app2}(D, A, C).
  \]

- **HO Solution**:
  \[
  \text{encryption}(A, B) :- \text{map}(A, B).
  \text{map}(A, B) :- \text{char_to_int}(A, E), \text{pred}(E, D),
  \text{pred}(D, C), \text{int_to_char}(C, B).
  \]

C.11 rotateN/3
Move first element of a list to the end n times.

- **FO Background**: empty, app2, head, tail, less0, zero, eq, pred.
- **FO Parameters**:
  max_vars: 7
  max_body: 6
  max_clause: 2
  Switches: enable_recursion

- **FO Solution**:
  \[
  \text{rotateN}(A, B, C) :- \text{eq}(B, C), \text{zero}(A).
  \text{rotateN}(A, B, C) :- \text{head}(B, D), \text{pred}(A, G),
  \text{less0}(A), \text{tail}(B, E),
  \text{app}(E, D, F), \text{rotateN}(G, F, C).
  \]

- **HO Solution**:
  \[
  \text{rotateN}(A, B, C) :- \text{iterate}(A, B, C).
  \text{iterate}(A, B, C) :- \text{head}(A, D), \text{tail}(A, C),
  \text{app2}(C, D, B).
  \]

C.12 allSeqN/2
Construct a list of lists, consisting of all sequences from 1 to i with i ≤ n.

- **FO Background**: suc, pred, zero, head, head2, tail, tail2, empty, less0
- **FO Parameters**:
  max_vars: 7
  max_body: 7
  max_clause: 5
  Switches: enable_recursion, enable_ho_recursion, non_datalog, allow_singletons

- **FO Solution**:
  \[
  \text{allSeqN}(A, B) :- \text{zero}(C), \text{h_a}(C, A, D), \text{g_a}(D, B).
  \text{g_a}(A, B) :- \text{empty}(A), \text{empty}(B).
  \text{g_a}(A, B) :- \text{head}(A, C), \text{tail}(A, F), \text{zero}(E),
  \text{h_p_a}(E, C, D), \text{g_p_a}(F, G),
  \text{tail2}(B, G), \text{head2}(B, D).
  \text{h_p_a}(A, B, C) :- \text{zero}(B), \text{empty}(C).
  \text{h_p_a}(A, B, C) :- \text{pred}(B, D), \text{suc}(A, E), \text{less0}(B),
  \text{h_p_a}(E, D, F), \text{tail2}(C, F),
  \text{head}(C, E).
  \]

- **HO Solution**:
  \[
  \text{allSeqN}(A, B) :- \text{iterate}(A, B, C).
  \text{iterate}(A, B, C) :- \text{head}(A, D), \text{tail}(A, C),
  \text{app2}(C, D, B).
  \]

- **HO Optimal Background**: empty, app2, head, tail, iterate, zero, eq, pred.
- **HO Optimal Parameters**:
  max_vars: 4
  max_body: 3
  max_clause: 3
  Switches:
• HO Solution:

```
allSeqN(A, B):- zero(C), iterate_a(C, A, D),
    map_a(D, B).
iterate_p_a(A, B):- suc(A, B).
map_p_a(A, B):- zero(C), iterate_a(C, A, B).
```

C.13 dropLastK/3

Given a list of lists, drop the last $k$ elements from each list

- FO Background: pred, eq, zero, tail, reverse, cons, empty
- FO Parameters:
  - max_vars: 8
  - max_body: 6
  - max_clause: 4
  - Switches: enable_recursion, enable_ho_recursion
- FO Solution:

```
dropLastK(A, B, C):- zero(A), eq(C, B), eq(B, C).
dropLastK(A, B, C):- pred(A, E), g_a(B, D),
    dropLastK(E, D, C).
    g_p_a(A, B):- empty(A), empty(B).
    g_p_a(A, B):- cons(A, C, D), reverse(C, E),
        tail(E, F), reverse(F, G),
        cons(B, G, H), g_p_a(D, H).
```

- HO Optimal Background: pred, eq, zero, tail, reverse, map
- HO Optimal Parameters:
  - max_vars: 5
  - max_body: 3
  - max_clause: 3
  - Switches: enable_recursion
- HO Solution:

```
dropLastK(A, B, C):- eq(C, B), zero(A).
dropLastK(A, B, C):- pred(A, E), map(B, D),
    dropLastK(E, D, C).
    map_p_a(A, B):- reverse(A, D), tail(D, C),
        reverse(C, B).
```

C.14 firstHalf/2

Check whether the second argument is equal to the first half of the first argument

- FO Background: head, tail, cons, app, true, empty, suc, pred, zero, len
- FO Parameters:
  - max_vars: 8
  - max_body: 7
- FO Solution:

```
    firstHalf(A, B):- len(A, C), g_a(A, C, B).
    g_p_a(A, B, C):- true(A), zero(B), empty(C).
    g_p_a(A, B, C):- pred(B, D), head(A, F),
        tail(A, G), pred(D, E),
        g_a(G, E, H), cons(C, F, H).
```

- HO Optimal Background: head, tail, pred, len, iterateStep
- HO Optimal Parameters:
  - max_vars: 3
  - max_body: 2
  - max_clause: 3
  - Switches: enable_recursion
- HO Solution:

```
    firstHalf(A, B):- len(A, C),
          iterateStep_a(A, C, B).
    iterateStep_p_a(A, B):- pred(A, C), pred(C, B),
                    iterateStep_q_a(A, B, C):- head(A, B), tail(A, C).
```

C.15 lastHalf/2

Check whether the second argument is equal to the last half of the first argument

- FO Background: empty, tail, front, last, app
- FO Parameters:
  - max_vars: 6
  - max_body: 5
  - max_clause: 3
  - Switches: enable_recursion
- FO Solution:

```
    lastHalf(A, B):- empty(B), empty(A).
    lastHalf(A, B):- tail(A, B), empty(B).
    lastHalf(A, B):- front(A, C), last(A, F),
          tail(C, E), lastHalf(E, D),
          app(D, F, B).
```

- HO Optimal Background: front, app2, reverse, empty, caseList
- HO optimal Parameters:
  - max_vars: 5
  - max_body: 3
  - max_clause: 4
  - Switches: enable_recursion
- HO Solution:
C.16 of1And2/1
check whether a list consists only of 1s and 2
• FO Background: suc, zero, cons1, empty.
• FO Parameters:
  max_vars: 5
  max_body: 5
  max_clause: 3
  Switches: enable_recursion
• FO Solution:
  of1And2(A):- empty(A).
  of1And2(A):- cons1(A, B, C), zero(D), suc(D, B), of1And2(2).
  of1And2(A):- cons1(A, B, C), zero(D), suc(D, E), suc(E, B), of1And2(2).
• HO Optimal Background: suc, zero, cons, empty, try.
• HO Optimal Parameters:
  max_vars: 3
  max_body: 4
  max_clause: 3
  Switches: enable_recursion
• HO Solution:
  of1And2(A):- empty(A).
  of1And2(A):- cons(A, C, B), try_a(C), of1And2(2).
  try_p_a(A):- zero(B), suc(B, A).
  try_q_a(A):- zero(B), suc(B, C), suc(C, A).

C.17 isPalindrome/1
Check whether a list is a palindrome (the same read normally and in reverse)
• FO Background: head, tail, empty, front, last.
• FO Parameters:
  max_vars: 4
  max_body: 5
  max_clause: 3
  Switches: enable_recursion
• FO Solution:
  isPalindrome(A):- empty(A).
  isPalindrome(A):- head(A, D), tail(A, B), front(B, C), last(B, D), isPalindrome(C).
• HO Optimal Background: empty, any, front, last, condList.
• HO Optimal Parameters:
  max_vars: 3
  max_body: 3
  max_clause: 3
  Switches:
• HO Solution:
  isPalindrome(A):- condList_a(A).
  condList_p_a(A, B):- any(A), empty(B).
  condList_p_a(A, B):- last(B, A), front(B, C), condList_a(C).
• HO Optimal Background: children, max, zero, suc, fold
• HO Optimal Parameters:
  max_vars: 3
  max_body: 4
  max_clause: 2
  Switches: enable_recursion, enable_ho_recursion
• HO Solution:
  depth(A, B):- zero(D), children(A, C), g_a(D, C, E), suc(E, B).
  g_p_a(A, B, C):- eq(A, C), empty(B).
  g_p_a(A, B, C):- tail(B, E), head(B, D), depth(D, F), max(A, F, G), g_p_a(G, E, C).
• HO Optimal Background: children, max, zero, suc, fold
• HO Optimal Parameters:
  max_vars: 5
  max_body: 4
  max_clause: 2
  Switches: enable_recursion
• HO Solution:
  depth(A, B):- children(A, C), zero(E), fold_a(E, C, D), suc(D, B).
  fold_p_a(A, B, C):- depth(B, D), max(D, A, C).
C.19 isBranch/2
check whether a given list is a branch of the tree
- FO Background: children, root, head, head2, tail, g_a
- FO Parameters:
  max_vars: 5
  max_body: 5
  max_clause: 4
  Switches: enable_recursion, enable_ho_recursion
- FO Solution:

```
isBranch(A, B):- tail(B, D), children(A, D), root(A, C), head2(B, C).
```

- HO Optimal Background: head, tail, empty, any, caseTree
- HO Optimal Parameters:
  max_vars: 4
  max_body: 3
  max_clause: 4
  Switches: enable_recursion
- HO Solution:

```
isBranch(A, B):- casetree_a(A, B).
casetree_p_a(A, B):- tail(B, C), empty(C), head(B, A).
any_p_a(A, B):- tail(B, C), casetree_a(A, C).
casetree_q_a(A, B, C):- head(C, A), any_a(B, C).
```

C.20 isSubTree/2
check whether the second argument is a sub-tree of the first argument.
- FO Background: head, tail, empty, children, g_a.
- FO Parameters:
  max_vars: 4
  max_body: 2
  max_clause: 4
  Switches: enable_recursion, enable_ho_recursion
- FO Solution:

```
g_p_a(A, B):- children(B, A).
g_p_a(A, B):- head(A, B).
g_p_a(A, B):- tail(A, C), g_a(C, B).
isSubTree(A, B):- children(A, C), g_p_a(C, B).
```

- HO Optimal Background: children, any, eq1.
- HO Optimal Parameters:
  max_vars: 3
  max_body: 2
  max_clause: 3
  Switches: enable_recursion
- HO Solution:

```
isSubTree(A, B):- eq(A, B).
isSubTree(A, B):- children(A, C), any_a(C, B).
any_p_a(A, B):- isSubTree(A, B).
```

C.21 addN/3
add n to every element of a list (with no addition predicate in the background knowledge)
- FO Background: suc, pred, zero, eq, less0, cons, cons2, empty, g_a
- FO Parameters:
  max_vars: 6
  max_body: 4
  max_clause: 4
  Switches: enable_recursion, enable_ho_recursion
- FO Solution:

```
addN(A, B, C):- zero(A), eq(B, C).
addN(A, B, C):- less0(A), g_a(B, D), pred(A, E), addN(E, D, C).
g_p_a(A, B):- empty(B), empty(A).
g_p_a(A, B):- cons(A, C, D), suc(C, E), g_p_a(D, F), cons2(B, E, F).
```

- HO Optimal Background: suc, eq, caseint, map
- HO Optimal Parameters:
  max_vars: 4
  max_body: 2
  max_clause: 4
  Switches:
- HO Solution:

```
addN(A, B, C):- caseint_a(A, B, C).
map_p_a(A, B):- suc(A, B).
caseint_q_a(A, B, C):- map_a(B, D), caseint_a(D, C).
caseint_p_a(A, B):- eq(A, B).
```
C.22 mulFromSuc/3

Multiply two numbers (with no addition predicate in the background knowledge).

- FO Background: suc, pred, eq, zero, less0, g.a, h.a.
- FO Parameters:
  - max_vars: 6
  - max_body: 4
  - max_clause: 5
  - Switches: allow_singletons, non_datalog.

- FO Solution:

  \[
  \text{mulFromSuc}(A, B, C) :- \text{zero}(D), \text{g.a}(D, A, C, B). \\
  \text{g.p.a}(A, B, C, D) :- \text{eq}(A, C), \text{zero}(B). \\
  \text{g.p.a}(A, B, C, D) :- \text{pred}(B, E), \text{less0}(B), \\
  \quad \text{h.a}(A, D, F), \text{g.a}(F, E, C, D). \\
  \text{h.p.a}(A, B, C) :- \text{eq}(A, C), \text{zero}(B). \\
  \text{h.p.a}(A, B) :- \text{suc}(A, E), \text{less0}(B), \\
  \quad \text{pred}(B, D), \text{h.a}(E, D, C). \\
  \]

- HO Optimal Background: suc, zero1, iterate.
- HO Optimal Parameters:
  - max_vars: 4
  - max_body: 2
  - max_clause: 3
  - Switches:

- HO Solution:

  \[
  \text{mulFromSuc}(A, B) :- \text{zero1}(D), \\
  \quad \text{iterate.a}(D, A, C, B). \\
  \text{iterate.p.a}(A, B, C) :- \text{iterate.a}(A, C, B). \\
  \text{iterate.p.a}(A, B) :- \text{suc}(A, B). \\
  \]