Learning Higher-Order Logic Programs from Failures*

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

Abstract

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

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

$$\boxed{\begin{array}{l}\texttt{allSeqN}(N, L)\texttt{:- iterate}(succ, 0, N, A),\ \texttt{map}(p, A, L).\\ \qquad\quad \texttt{p}(A, B)\texttt{:- iterate}(succ, 0, A, B).\end{array}}$$

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 [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*$_{\text{HO}}$ [Cropper *et al.*, 2020]. See Section 4.

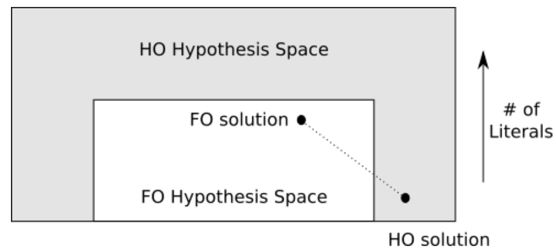

Figure 1: Inclusion of HO definitions increases the size of the search space, but can lead to the search space containing a shorter solutions.

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, `iterate`, being ternary (w.r.t. FO arguments), poses a challenge for some systems, and in the case of *HEXMIL*$_{\text{HO}}$ [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

*Authors contributed equally to the work described in this paper.
†Contact Author
[1]See Appendix of arXiv:2112.14603 for HO definitions.

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* and *entailment*; $p_1 \leq_\theta (\models)\ p_2 \Rightarrow H(p_1) \leq_\theta (\models)\ H(p_2)$ where p_1 and p_2 are logic programs, and $H(\cdot)$ is an HO definition incorporating parts of p_1 and p_2. Similar to classes considered in literature, our class excludes most cases of HO negation (see Section 3.4). 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 systems, namely, HO extensions of *Metagol* [Cropper and Muggleton, 2016] and *HEXMIL* [Kaminski *et al.*, 2018]. Also, we introduce Popper [Cropper and Morel, 2021a], the system *Hopper* is based on. For interested readers, a detailed survey of the current state of ILP research has recently been published [Cropper *et al.*, 2022].

2.1 Predicate Invention and HO Synthesis

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

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

Including p in the background knowledge is unintuitive. It is reasonable to expect the synthesizer to produce it. Many of the well known, non-MiL based ILP frameworks do not support predicate invention, *Foil* [Quinlan, 1990], *Progol* [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 $A_0 \leftarrow A_1, \cdots, A_n$, where A_i is a literal $P(T_1, \cdots, T_m)$, s.t. P is either a predicate symbol or a HO variable and each T_i is either a constant or a FO variable.*

For further discussion see Section 2.2. *Popper* [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. *Hopper*, our extension, has drastically improved performance when compared with *Popper*. *Hopper* also outperforms the state-of-the-art MiL-based ILP systems extended by HO definitions. For further discussion of *Popper* see Section 2.3, and for *Hopper* see Section 3.

2.2 *Metagol* and *HEXMIL*

We briefly summarize existing HO-capable ILP systems introduced by A. Cropper *et al.* [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 body_pred(P/n), sets of positive E^+ and negative E^- examples, *compiled* background knowledge BK_c, and a set of metarules M. The examples provide the arity and name of the goal predicate. Initially, *Metagol* attempts to satisfy E^+ using BK_c. If this fails, then *Metagol* attempts to unify the current goal atom with a metarule from $m \in M$. At this point *Metagol* tries to prove the body 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*$_{\text{HO}}$ is the inclusion of *interpreted* background knowledge BK_{in}. For example, map/3 as BK_{in} takes the form:

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

[2] $\texttt{P}(A,B)\texttt{:-}\ \texttt{Q}(A,C), \texttt{R}(C,B)$.

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

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

$$\begin{aligned} \texttt{half}_{\texttt{lst}}(A,B) &\texttt{:- } \texttt{reverse}(A,C), \\ &\quad \texttt{case}_{\texttt{list}}(p_{[\,]}, p_{[H|T]}, C, B). \\ \texttt{p}_{[\,]}(A) &\texttt{:- } \texttt{empty}(A). \\ \texttt{p}_{[H|T]}(A,B,C) &\texttt{:- } \texttt{empty}(B), \texttt{empty}(C). \\ \texttt{p}_{[H|T]}(A,B,C) &\texttt{:- } \texttt{front}(B,D)^4, \\ &\quad \underline{\texttt{case}_{\texttt{list}}(p_{[\,]}, p_{[H|T]}, D, E)}, \\ &\quad \texttt{append}(E,A,C). \end{aligned}$$

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

As with $half_{\texttt{lst}}(A, B)$, The following program defining $\texttt{issubtree}(A, B)$, which computes whether B is a subtree of A, requires recursively calling issubtree through any.

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

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

Negation of invented predicates (HO arguments of BK_{in} definitions), to the best of our knowledge, is not fully supported by *Metagol*$_{\mathrm{HO}}$ (See Section 4.2 of [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:* $P(A, B)$:- $Q_1(A, C_1), Q_2(C_1, C_2), \cdots, Q_n(C_{n-1}, B), R_1(D_1), \cdots, R_m(D_m)$ *where* $D_i \in \{A, C_1, \cdots, C_{n-1}, B\}$.

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*$_{\mathrm{HO}}$, 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 *HEXMIL*$_{\mathrm{HO}}$'s poor performance on list manipulation tasks and its limitations make application to tasks of interest difficult. Thus, we focus on *Metagol*$_{\mathrm{HO}}$ in Section 4.

2.3 Popper: Learning From Failures (LFF)

The LFF paradigm together with *Popper* 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, *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 [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.

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 [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**, Popper introduces constraints pruning specializations and/or generalizations of the candidate program. Specialization/generalization is defined via Θ-subsumption [Plotkin, 1970; Reynolds, 1970]. *Popper* may also introduce **elimination** Constraints pruning *separable*[5] sets of clauses. Details concerning the benefits of this approach are presented in [Cropper and Morel, 2021a]. Essentially, *Popper* 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, *Popper* incorporates a form of *language bias* [Nienhuys-Cheng *et al.*, 1997], 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 [Mug-

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

[4] $\texttt{front}(A,B)$:- $\texttt{reverse}(A,C), \texttt{tail}(C,D), \texttt{reverse}(D,B)$.

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

gleton, 1995] 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 [Gebser *et al.*, 2019] and an encoding of both definite logic programs and constraints within the *ASP* [Lifschitz, 2019] paradigm. The language bias together with the 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 [Lloyd, 1987].

3.1 Preliminaries

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

An *atom* is of the form $p(T_1, \cdots, T_m, t_1, \cdots, t_n)$. Let us denote this atom by a, then $sy(a) = p$ is the *symbol of the atom*, $ag_h(a) = \{T_1, \cdots, T_m\}$ are its *HO-arguments*, and $ag_f(a) = \{t_1, \cdots, t_n\}$ are its *FO-arguments*. When $ag_h(a) = \emptyset$ and $sy(a) \in \mathcal{P}$ we refer to a as *FO*, when $ag_h(a) \subset \mathcal{P}$ and $sy(a) \in \mathcal{P}$ we refer to a as *HO-ground*, otherwise it is *HO*. A *literal* is either an atom or its negation. A literal is *HO* if the atom it contains is *HO*. [6]

A *clause* is a set of literals. A *Horn* clause contains at most one positive literal while a definite clause must have exactly one positive literal. The atom of the positive literal of a definite clause c is referred to as the *head* of c (denoted by $hd(c)$), while the set of atoms of negated literals is referred to as the *body* (denoted by $bd(c)$). A *function-free definite (f.f.d) clause* only contains variables as FO arguments. We refer to a finite set of clauses as a *theory*. A theory is considered *FO* if all atoms are *FO*.

Replacing variables $P_1, \cdots, P_n, A_1, \cdots, A_m$ by predicate symbols $p_1, \cdots, p_n$ and terms $t_1, \cdots, t_m$ is a *substitution* (denoted by $\theta, \sigma, \cdots$) $\{P_1 \mapsto p_1, \cdots, P_n \mapsto p_n, A_1 \mapsto t_1, \cdots, A_m \mapsto t_m\}$. A substitution θ *unifies* two atoms when $a\theta = b\theta$.

3.2 Interpretable Theories and Groundings

Our hypothesis space consists 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. *Hopper* 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 $ag_h(hd(c))$ are pairwise distinct, $ag_h(hd(c)) \subset \mathcal{V}_h$, and $\forall a \in bd(c)$,*

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

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

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

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

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

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

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

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

Otherwise, interpretable theories do not require significant adaption of *Popper*'s *generate, test, constrain loop* [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, $\texttt{half}_{\texttt{lst}}$, *and* issubtree *of Section 2.1 & 2.2. Each is an interpretable theory. The sets of external literals of these theories are* $\{\texttt{fold}(p, C, A, B)\}$, $\{\texttt{case}_{\texttt{list}}(p_{[\,]}, p_{[H|T]}, C, B), \texttt{case}_{\texttt{list}}(p_{[\,]}, p_{[H|T]}, D, E)\}$, *and* $\{\texttt{any}(\texttt{cond}, C, B)\}$, *respectively.*

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

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

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

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

An L-compatible theory $\mathfrak{T}$ can be L*-grounded*. This requires replacing external literals of $\mathfrak{T}$ by FO literals, i.e. removal of all HO arguments and replacing the predicate symbol

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

[7] Similar to definitional HO of W. Wadge [Wadge, 1991].

of the external literals with fresh predicate symbols, resulting in $\mathfrak{T}^*$, 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 $\mathfrak{T}$, i.e. $\mathfrak{T}^*$ (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* $\mathtt{fold}_{p\_a}$ *for clarity purposes), we get the following L-grounding:*

$$
\begin{array}{l}
\mathtt{reverse}(A,B)\text{:- } \mathtt{empty}(C),\ \mathtt{fold}_a(C,A,B).\\
\mathtt{fold}_{p\_a}(A,B,C)\text{:- } \mathtt{head}(C,B),\ \mathtt{tail}(C,A).\\
\mathtt{fold}_a(A,B,C)\text{:- } \mathtt{fold}(\mathtt{fold}_{p\_a},A,B,C).\\
\mathtt{fold}(P,A,B,C)\text{:- } \mathtt{empty}(B), A=C.\\
\mathtt{fold}(P,A,B,C)\text{:- } \mathtt{head}(B,H), P(H,D),\\
\qquad\qquad \mathtt{tail}(B,T), \mathtt{fold}(P,D,T,C).
\end{array}
$$

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

If an *L-compatible* theory contains multiple external literals whose symbol is fold, i.e. $\mathtt{fold}(\mathtt{fold}_{p\_a},C,A,B)$ and $\mathtt{fold}(\mathtt{fold}_{p\_a},D,E,R)$, both are renamed to $\mathtt{fold}_a$. However, if the higher-order arguments differ, i.e. $\mathtt{fold}_{p\_a}$, and $\mathtt{fold}_{p\_b}$, then they are renamed to $\mathtt{fold}_a$ and $\mathtt{fold}_b$, and an additional clause $\mathtt{fold}_b(A,B,C)$:- $\mathtt{fold}(\mathtt{fold}_{p\_b},A,B,C)$ 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 *Metagol*$_{\mathrm{HO}}$. 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 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 Θ-*subsumption*:

Definition 6 (Θ-subsumption) *An FO theory T_1 subsumes an FO theory T_2, denoted by $T_1 \leq_\theta T_2$ iff, $\forall c_2 \in T_2 \exists c_1 \in T_1$ s.t. $c_1 \leq_\theta c_2$, where $c_1 \leq_\theta c_2$ iff, $\exists\theta$ s.t. $c_1\theta \subseteq c_2$.*

Importantly, the following property holds:

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

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

Definition 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 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 [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:

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

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

$$
\begin{array}{ll}
E^+ : f(b). \quad f(c). & \mathbf{E}^- : f(a).\\
BK : \left\{ \begin{array}{ll} p(a). & p(b).\\ q(a). & q(c). \end{array} \right\} & \mathbf{HO} : N(P,A)\text{:- } \neg\, P(A).
\end{array}
$$

$\underline{prog_s}$

$$
\begin{array}{l}
\mathtt{f}(A)\text{:- } \mathtt{N}(p_1,A).\\
\mathtt{p}_1(A)\text{:- } \mathtt{p}(A),\ \mathtt{q}(A).
\end{array}
$$

$\underline{prog_f}$

$$
\begin{array}{l}
\mathtt{f}(A)\text{:- } \mathtt{N}(p_1,A).\\
\mathtt{p}_1(A)\text{:- } \mathtt{p}(A).
\end{array}
$$

The optimal solution is $prog_s$, $prog_f$ is an *incorrect* hypothesis which *Hopper* can generate prior to $prog_s$, and $prog_f \leq_\theta prog_s$. Note, $prog_f \models \neg f(b) \wedge \neg f(a) \wedge f(c)$, it does not entail all of E^+. We should generalize $prog_f$ to find a solution, i.e. add literals to p_1. The introduced constraints [Cropper and Morel, 2021a] prune programs extending p_1, i.e. $prog_s$. Similar holds for specializations. Consider the ILP task and candidate programs:

$$
\begin{array}{ll}
E^+ : f(a). \quad f(b). & \mathbf{E}^- : f(c). \quad f(d).\\
BK : \{\ p(d). \quad q(c).\ \} & \mathbf{HO} : N(P,X)\text{:- } \neg\, P(X).
\end{array}
$$

$\underline{prog_s}$

$$
\begin{array}{l}
\mathtt{f}(A)\text{:- } \mathtt{N}(p_1,A).\\
\mathtt{p}_1(A)\text{:- } \mathtt{p}(A).\\
\mathtt{p}_1(A)\text{:- } \mathtt{q}(A).
\end{array}
$$

$\underline{prog_f}$

$$
\begin{array}{l}
\mathtt{f}(A)\text{:- } \mathtt{N}(p_1,A).\\
\mathtt{p}_1(A)\text{:- } \mathtt{p}(A).
\end{array}
$$

The optimal solution is $prog_s$, $prog_f$ is an *incorrect* hypothesis which *Hopper* can generate prior to $prog_s$, and

$prog_s \leq_\theta prog_f$. Note, $prog_f \models f(a) \wedge f(b) \wedge f(c)$, it entails some of E^-. We should specialize p_1 to find a solution, i.e. add clauses to $prog_f$. The introduced constraints [Cropper and Morel, 2021a] prune programs that add clauses, i.e. $prog_s$.

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

4 Experiments

A possible, albeit very weak, program synthesizer is an enumeration procedure that orders all possible programs constructible from the BK by size, testing each until a solution is found. In [Cropper and Morel, 2021a], this procedure was referred to as *Enumerate*. *Popper* extends *Enumerate* by pruning the hypothesis space based on 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 [Cropper and Morel, 2021a], illustrating that *Popper* 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 [Cropper *et al.*, 2020], in addition to introducing *Metagol*$_{\text{HO}}$ and *HEXMIL*$_{\text{HO}}$, provided a comprehensive suite of experiments illustrating that the addition of HO predicates can improve accuracy and, most importantly, reduce learning time. Reduction in learning times results from a reduction in the complexity/size of the solutions.

The experiments in [Cropper and Morel, 2021a] 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 [Cropper *et al.*, 2020] 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*$_{\text{HO}}$without being overly descriptive or suffering from long learning times.

Our experiments illustrate that the combination of *Popper* and HO predicates [Cropper *et al.*, 2020] significantly improves *Popper*'s performance at learning complex programs. Similar to [Cropper and Morel, 2021a], we use predicate declarations, i.e. `body_pred(head,2)`, 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 [Cropper and Morel, 2021a] and 2 presented in [Cropper *et al.*, 2020]. 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. a 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 *Metagol*$_{\text{HO}}$. Successful learning using *Metagol*$_{\text{HO}}$ 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(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

Task	*Popper* (**Opt**)	#Literals	PI?	*Hopper*	***Hopper* (Opt)**	#Literals	HO-Predicates	*Metagol*$_{\mathrm{HO}}$	Metatypes?
Learning Programs by learning from Failures [Cropper and Morel, 2021a]									
dropK	1.1s	7	no	0.5s	**0.1s**	4	iterate	no	no
allEven	0.2s	7	no	0.2s	**0.1s**	4	all	yes	no
findDup	0.25s	7	no	−−	**0.5s**	10	caseList	no	yes
length	0.1s	7	no	0.2s	**0.1s**	5	fold	yes	no
member	0.1s	5	no	0.2s	**0.1s**	4	any	yes	no
sorted	65.0s	9	no	46.3s	**0.4s**	6	fold	yes	no
reverse	11.2s	8	no	7.7s	**0.5s**	6	fold	yes	no
Learning Higher-Order Logic Programs [Cropper *et al.*, 2020]									
dropLast	300.0s	10	no	300s	**2.9s**	6	map	yes	no
encryption	300.0s	12	no	300s	**1.2s**	7	map	yes	no
Additional Tasks									
repeatN	5.0s	7	no	0.6s	**0.1s**	5	iterate	yes	no
rotateN	300.0s	10	no	300s	**2.6s**	6	iterate	yes	no
allSeqN	300.0s	25	yes	300s	**5.0s**	9	iterate, map	yes	no
dropLastK	300.0s	17	yes	300s	**37.7s**	11	map	no	no
firstHalf	300.0s	14	yes	300s	**0.2s**	9	iterateStep	yes	no
lastHalf	300.0s	12	no	300s	**155.2s**	12	caseList	no	yes
of1And2	300.0s	13	no	300s	**6.9s**	13	try	no	no
isPalindrome	300.0s	11	no	157s	**2.4s**	9	condlist	no	yes
depth	300.0s	14	yes	300s	**10.1s**	8	fold	yes	yes
isBranch	300.0s	17	yes	300s	**25.9s**	12	caseTree, any	no	yes
isSubTree	2.9s	11	yes	1.0s	**0.9s**	7	any	yes	yes
addN	300.0s	15	yes	300s	**1.4s**	9	map, caseInt	yes	no
mulFromSuc	300.0s	19	yes	300s	**1.2s**	7	iterate	yes	no

Table 1: We ran *Popper*, *Hopper*, *optimized Hopper* , and *Metagol*$_{\mathrm{HO}}$ on a single core with a timeout of 300 second. Times denote the average of 5 runs. Evaluation time for *Popper* and *Hopper* was set to a thousandth of a second, sufficient time for all task involved.

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*$_{\mathrm{HO}}$ 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 [Cropper and Tourret, 2020], the existence problem of a set of *general* metarules over which *Metagol*$_{\mathrm{HO}}$'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*$_{\mathrm{HO}}$. Our experiments test the theoretical possibility of *Metagol*$_{\mathrm{HO}}$ 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 notsuccessfully 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 Math$_{LP}$ 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_053/0017594 of the Ministry of Education, Youth and Sports of the Czech Republic.

References

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

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

[Cropper and Morel, 2021b] Andrew Cropper and Rolf Morel. Predicate invention by learning from failures. *CoRR*, abs/2104.14426, May 2021.

[Cropper and Muggleton, 2014] Andrew Cropper and Stephen H. Muggleton. Logical minimisation of metarules within meta-interpretive learning. In *Proceedings of the 24th International Conference on Inductive Logic Programming*, pages 62–75, Nancy, France, September 2014. Springer.

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

[Cropper and Tourret, 2020] Andrew Cropper and Sophie Tourret. Logical reduction of metarules. *Machine Learning*, 109(7):1323–1369, July 2020.

[Cropper *et al.*, 2020] Andrew Cropper, Rolf Morel, and Stephen Muggleton. Learning higher-order logic programs. *Machine Learning*, 109(7):1289–1322, July 2020.

[Cropper *et al.*, 2022] Andrew Cropper, Sebastijan Dumančić, Richard Evans, and Stephen H. Muggleton. Inductive logic programming at 30. *Machine Learning*, 111(1):147–172, Jan 2022.

[Cropper, 2020] Andrew Cropper. Forgetting to learn logic programs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(04):3676–3683, April 2020.

[Eiter *et al.*, 2016] Thomas Eiter, Christoph Redl, and Peter Schüller. Problem solving using the HEX family. In *Proceedings of Computational Models of Rationality*, pages 150–174. College Publications, June 2016.

[Evans and Grefenstette, 2018] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, 61:1–64, January 2018.

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

[Kaminski *et al.*, 2018] Tobias Kaminski, Thomas Eiter, and Katsumi Inoue. Exploiting answer set programming with external sources for meta-interpretive learning. *Theory and Practice of Logic Programming*, 18(3-4):571–588, October 2018.

[Law *et al.*, 2014] Mark Law, Alessandra Russo, and Krysia Broda. Inductive learning of answer set programs. In *Proceedings of Logics in Artificial Intelligence*, pages 311–325. Springer, August 2014.

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

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

[Muggleton *et al.*, 2012] Stephen Muggleton, Luc De Raedt, David Poole, Ivan Bratko, Peter A. Flach, Katsumi Inoue, and Ashwin Srinivasan. ILP turns 20 - biography and future challenges. *Machince Learning*, 86(1):3–23, January 2012.

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

[Muggleton, 1991] Stephen Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, February 1991.

[Muggleton, 1995] Stephen Muggleton. Inverse entailment and progol. *New Generation Computing*, 13(3&4):245–286, December 1995.

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

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

[Quinlan and Cameron-Jones, 1993] J. R. Quinlan and R. M. Cameron-Jones. Foil: A midterm report. In *Proceedings of the European Conference on Machine Learning*, pages 1–20, Vienna, Austria, April 1993. Springer.

[Quinlan, 1990] J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, August 1990.

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

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

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

[Srinivasan *et al.*, 2003] Ashwin Srinivasan, Ross D. King, and Michael E. Bain. An empirical study of the use of relevance information in inductive logic programming. *Journal of Machine Learning Research*, 4:369–383, December 2003.

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

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