Generalisation through Negation and Predicate Invention

David M. Cerna[1], **Andrew Cropper**[2]
[1]Czech Academy of Sciences Institute of Computer Science (CAS ICS), Prague, Czechia
[2]University of Oxford, United Kingdom
dcerna@cs.cas.cz, andrew.cropper@cs.ox.ac.uk

Abstract

The ability to generalise from a small number of examples is a fundamental challenge in machine learning. To tackle this challenge, we introduce an inductive logic programming (ILP) approach that combines negation and predicate invention. Combining these two features allows an ILP system to generalise better by learning rules with universally quantified body-only variables. We implement our idea in NOPI, which can learn normal logic programs with predicate invention, including Datalog programs with stratified negation. Our experimental results on multiple domains show that our approach can improve predictive accuracies and learning times.

Introduction

Zendo is a game where one player, the *teacher*, creates a hidden rule for structures. The other players, the *students*, aim to discover the rule by building structures. The teacher provides feedback by marking which structures follow or break the rule without further explanation. The students continue to guess the rule. The first student to correctly guess the rule wins. For instance, consider the examples shown in Figure 1. A possible rule for these examples is *"there are two red cones"*.

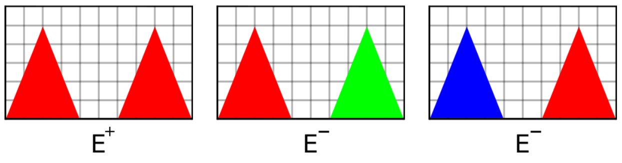

Figure 1: Positive (E^+) and negative (E^-) Zendo examples.

Suppose we want to use machine learning to play Zendo, i.e. to learn rules from examples. Then we need an approach that can (i) learn explainable rules, and (ii) generalise from a small number of examples. Although crucial for many problems, these requirements are difficult for standard machine learning techniques (Cropper et al. 2022).

Inductive logic programming (ILP) (Muggleton 1991) is a form of machine learning that can learn explainable rules from a small number of examples. For instance, an ILP system could learn the following hypothesis (a set of logical rules) from the examples in Figure 1:

$$\{\, f(S) \leftarrow cone(S,A), red(A), cone(S,B), red(B), all\_diff(A,B) \,\}$$

This hypothesis says that the relation f holds for the state S when there are two distinct red cones A and B, i.e. this hypothesis says there are two red cones.

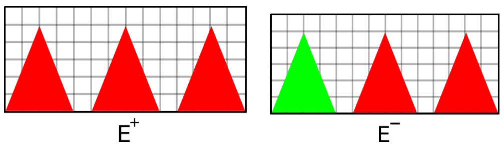

Figure 2: Additional Zendo examples.

Suppose we are given the two new examples shown in Figure 2. Our previous hypothesis does not correctly explain the new examples as it entails the new negative example. To correctly explain all the examples, we need a disjunctive hypothesis that says *"[there are exactly two red cones] or [there are exactly three red cones]"*. Given a new positive example with four red cones and a negative example with three red and one green cone, we would need to learn yet another rule that says *"there are exactly four red cones"*.

As is hopefully clear, we would struggle to generalise beyond the training examples using this approach because we need to learn a rule for each number of cones. Rather than learn a rule for each number of cones, we would ideally learn a single rule that says *"all the cones are red"*. However, most ILP approaches struggle to learn rules of this form because they only learn Datalog or definite programs and thus only learn rules with *existentially* quantified body-only variables (Apt and Blair 1991; Dantsin et al. 2001).

To overcome this limitation, we combine *negation as failure* (NAF) (Clark 1977) and *predicate invention* (PI) (Stahl 1995) to learn rules with *universally* quantified body-only variables. The main reason to combine NAF and PI is that many concepts can only be expressed in this more expressive language (Stahl 1995; Dantsin et al. 2001). For instance, for the Zendo scenario, our approach, which combines negation and PI, learns the hypothesis:

$$\left\{ \begin{array}{l} f(S) \leftarrow scene(S),\ not\ inv_1(S) \\ inv_1(S) \leftarrow cone(S,P),\ not\ red(P) \end{array} \right\}$$

This hypothesis says *"all the cones are red"*. The predicate symbol inv_1 is not provided as input and is invented by our approach. The rule defined by inv_1 says *"there is a cone that is not red"*. The rule defined by f negates this rule and says *"it is not true that there is a cone that is not red"*. The hypothesis, therefore, states *"there does not exist a cone that is not red"* which by the equivalences of first-order logic ($\forall \equiv$ **not** $\exists$ **not**) is the same as *"all the cones are red"*.

To combine negation and PI, we build on *learning from failures* (LFF) (Cropper and Morel 2021). An LFF learner continually generates and tests hypotheses, from which it infers constraints. For instance, if a hypothesis is too general, i.e. entails a negative example, an LFF learner, such as POPPER, builds a generalisation constraint to prune more general hypotheses from the hypothesis space. We extend LFF from learning definite (monotonic) programs to learning *polar* programs, a fragment of normal (non-monotonic) programs. The key benefit of polar programs is that we can efficiently reason about the subsumption relation between them in our learning algorithm. Furthermore, we show (Theorem 2) that polar programs capture Datalog with stratified negation (Dantsin et al. 2001). We implement our idea in NOPI, which, as it builds on POPPER, supports learning recursive and optimal programs.

Novelty, impact, and contributions. The key novelty of our approach is *the ability to learn normal logic programs with invented predicate symbols*. We expand on this novelty in the *Related Work* section. The impact is that our approach can learn programs that existing approaches cannot. Specifically, we claim that combining negation and PI can improve learning performance by allowing us to learn rules with universally quantified body-only variables. Our experiments on multiple domains support our claim and show that our approach leads to vastly improved predictive accuracies and learning times.

Overall, we make the following contributions:

1. We introduce *polar* programs, a fragment of *stratified* logic programs. We show (Theorem 2) that this fragment of normal logic programs can capture Datalog with stratified negation (Dantsin et al. 2001).
2. We introduce LFF constraints for this non-monotonic setting and prove their soundness (Propositions 1 and 2).
3. We introduce NOPI, an ILP system that can learn normal logic programs with PI and recursion, such as Datalog programs with stratified negation.
4. We empirically show on multiple domains that (i) NOPI can outperform existing approaches and (ii) our non-monotonic constraints can reduce learning times.

Related Work

Program synthesis. ILP is a form of program synthesis (Shapiro 1983), which attracts a broad community of researchers (Evans and Grefenstette 2018; Ellis et al. 2018; Silver et al. 2022). Many recent approaches synthesise monotonic Datalog programs (Si et al. 2019; Raghothaman et al. 2020; Bembenek, Greenberg, and Chong 2023). We differ in many ways, including by learning non-monotonic programs.

Negation. Many ILP approaches learn non-monotonic programs (Quinlan 1990; Srinivasan, Muggleton, and Bain 1992; Dimopoulos and Kakas 1995; Sakama 2001; Sakama and Inoue 2009; Ray 2009). Most use negation to handle exceptions such as *"birds fly except penguins"* and thus require negative examples. For instance, Inoue and Kudoh (1997) learn normal logic programs by first learning a program that covers the positive examples and then adding exceptions (using NAF) to account for the negative examples. By contrast, we combine NAF and PI to improve generalisation and do not need negative examples – as Bekker and Davis (2020) state, it is sometimes necessary to learn from positive examples alone. Moreover, most approaches build on inverse entailment (Muggleton 1995), so they struggle to learn recursive and optimal programs. By contrast, our approach can learn recursive and optimal programs because we build on LFF. As Fogel and Zaverucha (1998) state, learning non-monotonic programs is difficult because the standard subsumption relation does not hold in general for normal programs. To overcome this challenge, the authors introduce a subsumption relation for normal programs based on the dependency graph of predicate symbols in a program. We differ because we introduce a general fragment of normal logic programs related to stratified logic programs. Moreover, our approach supports PI.

Predicate invention. Although crucial for many tasks, such as planning (Silver et al. 2022) and learning complex algorithms (Cropper and Muggleton 2019), most ILP systems do not support PI (Muggleton 1995; Srinivasan 2001; Blockeel and De Raedt 1998; Corapi, Russo, and Lupu 2011; Zeng, Patel, and Page 2014; Inoue, Ribeiro, and Sakama 2014). Approaches that support PI usually need metarules to restrict the syntax of hypotheses (Muggleton, Lin, and Tamaddoni-Nezhad 2015; Evans and Grefenstette 2018; Kaminski, Eiter, and Inoue 2019; Hocquette and Muggleton 2020; Dai and Muggleton 2021; Glanois et al. 2022), which, in some cases, are impossible to provide (Cropper and Tourret 2020). By contrast, we do not need metarules.

Negation and predicate invention. Ferilli (2016) describe an approach that specialises a theory to account for a misclassified negative example. If a negative example is misclassified, they introduce a conjunction of negated preconditions, where each precondition is an invented predicate. Their approach only works in a Datalog setting, cannot learn recursive programs, and only works when a negative example is misclassified. We differ because we (i) do not need negative examples, (ii) learn recursive programs, (iii) learn normal logic programs, and (iv) learn optimal programs. Siebers and Schmid (2018) learn recursive programs with negation and PI. Their approach first learns a program for the positive examples and allows some negative examples to be covered. It then flips the examples (false positives from the previous iteration are positive examples, and the previous true positives are now negative examples) and tries to learn again. We differ because we do not need negative examples or metarules. ILASP (Law, Russo, and Broda 2014) can learn non-monotonic programs with invented predicate symbols if a user tells it which symbols to invent. By contrast, NOPI does not need this information. Moreover, ILASP precomputes every possible rule in the hypothesis space, which is often infeasible. For instance, our Zendo4 experiment has approximately 10^{10} rules in the hypothesis space.

Problem Setting

We assume familiarity with logic programming (Lloyd 2012) and ASP (Gebser et al. 2012) but have included summaries in Appendix A For clarity, we define some key terms. A *normal* rule is of the form $h \leftarrow b_1, \cdots, b_n, \textbf{not}\ b_{n+1}, \cdots, \textbf{not}\ b_{n+m}$ where h is the *head* atom, each b_i is a literal, and $b_1, \cdots, b_n, \textbf{not}\ b_{n+1}, \cdots, \textbf{not}\ b_{n+m}$ is the *body*. The symbol **not** denotes *negation as failure* (Clark 1977). A literal is an atom (a non-negated literal) or the negation of an atom (a negated literal). A *normal* logic program is a set of normal rules. A clause is a set of literals. A *definite* clause is a clause with exactly one non-negated literal. A *substitution* $\theta = \{v_1/t_1, ..., v_n/t_n\}$ is the simultaneous replacement of each variable v_i by its corresponding term t_i. A clause C_1 *subsumes* a clause C_2 if and only if there exists a substitution θ such that $C_1\theta \subseteq C_2$ (Plotkin 1971). A definite theory P subsumes a definite theory Q ($P \preceq_\theta Q$) if and only if $\forall r_2 \in Q, \exists r_1 \in P$ such that r_1 subsumes r_2. A definite theory P is a *specialisation* of a definite theory Q if and only if $Q \preceq_\theta P$. A definite theory P is a *generalisation* of a definite theory Q if and only if $P \preceq_\theta Q$.

Polar Programs

To learn normal programs, we need to go beyond definite programs and standard subsumption. To do so, we introduce *polar programs*, which are normal programs where predicate symbols have polarities. We first define *top symbols*, which are head predicate symbols that are only used positively in a program:

Definition 1 (**Top symbols**). *Let P be a normal program. Then $top(P)$ is the inclusion-maximal subset of the head predicate symbols occurring in P satisfying the following two conditions:*

- *if $p \in top(P)$, then p does not occur in a negated literal in P*
- *if $p \in top(P)$ and p is in the body of a rule r then the head predicate symbol of r is in $top(P)$*

We define *defs(P)* as the set of all head predicate symbols in P that are not in *top(P)*.

Example 1. *To illustrate top symbols, consider the program:*

$$P = \left\{ \begin{array}{l} p \leftarrow g, q \\ p \leftarrow w, \textbf{not}\ s \\ g \leftarrow q, p \\ s \leftarrow l \end{array} \right\}$$

In this program, $top(P) = \{p, g\}$ and $defs(P) = \{s\}$.

In a normal program P, the polarity of every head predicate symbol p is *positive* ($pos(p)$) or *negative* ($neg(p)$). The polarity of a symbol in *top(P)* is positive. By contrast, the polarity of a symbol in *defs(P)* depends on whether the symbol is used positively or negatively:

Definition 2 (**Polarity**). *Let P be a normal program, r be a rule in P, p be the head predicate symbol of r, $body^+(r)$ and $body^-(r)$ be the predicate symbols that appear in non-negated and negated body literals in r respectively, and q in $defs(P)$ be a predicate symbol in the body of r. Then the polarity of q is as follows:*

(1) if $q \in body^+(r)$ and $pos(p)$ then $pos(q)$
(2) if $q \in body^-(r)$ and $pos(p)$ then $neg(q)$
(3) if $q \in body^+(r)$ and $neg(p)$ then $neg(q)$
(4) if $q \in body^-(r)$ and $neg(p)$ then $pos(q)$

Example 2. *Consider the program P from Example 1. The polarities of the head predicate symbols are $pos(p)$, $pos(g)$, and $neg(s)$.*

We define a *polar program*:

Definition 3 (**Polar program**). *A normal program P is polar if and only if the polarity of every head predicate symbol in P is exclusively positive or negative.*

Example 3. *The following program is not polar because the polarity of* odd *is neither positive nor negative:*

$$\{\ odd(\text{X}) \leftarrow \text{succ(Y,X)}, \textbf{not}\ odd(\text{Y})\}$$

Example 4. *The following stratified program is not polar because the polarity of inv_1 is positive and negative:*

$$\left\{ \begin{array}{l} \text{f} \leftarrow \text{inv}_1, \textbf{not}\ \text{inv}_1 \\ \text{inv}_1 \leftarrow \text{inv}_2 \\ \text{inv}_2 \leftarrow \text{w} \end{array} \right\}$$

Example 5. *The following program is polar because only $pos(unconnected)$ and $neg(inv_1)$ hold:*

$$\left\{ \begin{array}{l} r_1 : unconnected(A, B) \leftarrow \textbf{not}\ inv_1(\text{A,B}) \\ r_2 : inv_1(\text{A,B}) \leftarrow edge(\text{A,B}) \\ r_3 : inv_1(\text{A,B}) \leftarrow edge(\text{A,C}), inv_1(\text{C,B}) \end{array} \right\}$$

The rules of a polar program P are *positive* (P^+) or *negative* (P^-) depending on the polarity of their head symbols.

Example 6. *Consider the program P from Example 5. Then $P^+ = \{r_1\}$ and $P^- = \{r_2, r_3\}$.*

We can compare positive rules using standard subsumption. For negative rules, we need to flip the order of comparison. To do so, we introduce *polar subsumption*:

Definition 4 (**Polar subsumption**). *Let P and Q be polar programs. Then P polar subsumes Q ($P \preccurlyeq_\diamond Q$) iff $P^+ \preceq_\theta Q^+$ and $Q^- \preceq_\theta P^-$.*

Example 7. *To understand the intuition behind Definition 4, consider the following polar programs:*

$$P = \left\{ \begin{array}{l} r_1\text{: f} \leftarrow \textbf{not}\ inv_1 \\ r_2\text{: } inv_1 \leftarrow \text{a, b} \end{array} \right\} \quad Q = \left\{ \begin{array}{l} r_3\text{: f} \leftarrow \textbf{not}\ inv_1 \\ r_4\text{: } inv_1 \leftarrow \text{a} \end{array} \right\}$$

Note that $P \preccurlyeq_\diamond Q$ because $r_1 \preceq_\theta r_3$ where $r_1 \in P^+$ and $r_3 \in Q^+$, and $r_4 \preceq_\theta r_2$ where $r_2 \in P^-$ and $r_4 \in Q^-$.

$$P = \left\{ \begin{array}{l} r_1\text{: f} \leftarrow \textbf{not}\ inv_1 \\ r_2\text{: } inv_1 \leftarrow \text{b} \\ r_3\text{: } inv_1 \leftarrow \text{a} \end{array} \right\} \quad Q = \left\{ \begin{array}{l} r_4\text{: f} \leftarrow \textbf{not}\ inv_1 \\ r_5\text{: } inv_1 \leftarrow \text{b} \end{array} \right\}$$

Note that $Q \preccurlyeq_\diamond P$ because $r_4 \preceq_\theta r_1$ where $r_1 \in P^+$ and $r_4 \in Q^+$, and $r_2 \preceq_\theta r_5$ where $r_2 \in P^-$ and $r_5 \in Q^-$.

We show that polar subsumption implies entailment. Note, the properties of negation we require to prove the following two theorems hold in the commonly used semantics for NAF (such as stable and well-founded) when unstratified usage of negation does not occur. Both stratified and polar logic programs do not allow for unstratified usage of negation.

Theorem 1 (Entailment property). *Let P and Q be polar programs such that $P \preccurlyeq_\diamond Q$. Then $P \models Q$.*

Proof. For any r_2 in Q^+ the implication trivially follows from properties of $\leq_\theta$. Now let r_1 in P^- and r_2 in Q^- such that $r_2 \leq_\theta r_1$; this implies $r_2 \models r_1$. By contraposition, we derive $\textbf{not}\ r_1 \models \textbf{not}\ r_2$.

□

We also show that polar programs can express all concepts expressible as stratified programs:

Theorem 2. *Let S be a stratified logic program. Then there exists a polar program P such that for all $p \in top(S)$, $S \models \forall \vec{x}.p(\vec{x})$ iff $P \models \forall \vec{x}.p(\vec{x})$ (See Appendix C).*

Thus, we do not lose expressivity by learning polar programs rather than stratified programs.

Learning From Failures (LFF)

LFF searches a hypothesis space (a set of hypotheses) for a hypothesis that generalises examples and background knowledge. In the existing literature, an LFF hypothesis is a definite (monotonic) program. LFF uses *hypothesis constraints* to restrict the hypothesis space. Let $\mathcal{L}$ be a language that defines hypotheses. A *hypothesis constraint* is a constraint expressed in $\mathcal{L}$. Let C be a set of hypothesis constraints written in a language $\mathcal{L}$. A hypothesis H is *consistent* with C if, when written in $\mathcal{L}$, H does not violate any constraint in C. We denote as $\mathcal{H}_C$ the subset of the hypothesis space $\mathcal{H}$ which does not violate any constraint in C.

LFFN

We extend LFF to learn polar programs, which we call the *learning from failures with negation (LFFN)* setting. We define the LFFN input:

Definition 5 (LFFN input). *A* LFFN *input is a tuple $(E^+, E^-, B, \mathcal{H}, C)$ where E^+ and E^- are sets of ground atoms denoting positive and negative examples respectively; B is a normal logic program denoting background knowledge; $\mathcal{H}$ is a hypothesis space of polar programs, and C is a set of hypothesis constraints.*

To be clear, an LFFN hypothesis is a polar (non-monotonic) program and the hypothesis space is a set of polar programs. We define an LFFN solution:

Definition 6 (LFFN solution). *Given an input tuple $(E^+, E^-, B, \mathcal{H}, C)$, a hypothesis $H \in \mathcal{H}_C$ is a* solution *when H is* complete *($\forall e \in E^+,\ B \cup H \models e$) and* consistent *($\forall e \in E^-,\ B \cup H \not\models e$).*

If a hypothesis is not a solution then it is a *failure*. A hypothesis is *incomplete* when $\exists e \in E^+,\ H \cup B \not\models e$; *inconsistent* when $\exists e \in E^-,\ H \cup B \models e$; *partially complete* when $\exists e \in E^+,\ H \cup B \models e$; and *totally incomplete* when $\forall e \in E^+,\ H \cup B \not\models e$.

Let $cost : \mathcal{H} \mapsto \mathbb{N}$ be a function that measures the cost of a hypothesis. We define an *optimal* solution:

Definition 7 (Optimal solution). *Given an input tuple $(E^+, E^-, B, \mathcal{H}, C)$, a hypothesis $H \in \mathcal{H}_C$ is* optimal *when (i) H is a solution and (ii) $\forall H' \in \mathcal{H}_C$, where H' is a solution, $cost(H) \leq cost(H')$.*

Our cost function is the number of literals in the hypothesis.

LFFN Constraints

An LFF learner learns hypothesis constraints from failed hypotheses. Cropper and Morel (2021) introduce hypothesis constraints based on subsumption. A *specialisation* constraint prunes specialisations of a hypothesis. A *generalisation* constraint prunes generalisations. The existing LFF constraints are only sound for monotonic programs, i.e. they can incorrectly prune optimal solutions when learning non-monotonic programs, and are thus unsound for the LFFN setting. The reason for unsoundness is that entailment is not a consequence of subsumption in a non-monotonic setting, even in the propositional case, as the following examples illustrate.

Example 8. *Consider the programs:*

$$P = \left\{ \begin{array}{l} \text{a.} \\ \text{f} \leftarrow \textbf{not}\ inv_1 \\ inv_1 \leftarrow \text{b} \\ inv_1 \leftarrow \text{a} \end{array} \right\} Q = \left\{ \begin{array}{l} \text{a.} \\ \text{f} \leftarrow \textbf{not}\ inv_1 \\ inv_1 \leftarrow \text{b} \end{array} \right\}$$

Note that $P \preceq_\theta Q$ and $Q \models f$ but $P \not\models f$. Similarly, we have the following:

$$P' = \left\{ \begin{array}{l} \text{a.} \\ \text{f} \leftarrow \textbf{not}\ inv_1 \\ inv_1 \leftarrow \text{a, b} \end{array} \right\} Q' = \left\{ \begin{array}{l} \text{a.} \\ \text{f} \leftarrow \textbf{not}\ inv_1 \\ inv_1 \leftarrow \text{a} \end{array} \right\}$$

Note that $Q' \preceq_\theta P'$ and $P' \models f$ but $Q' \not\models f$.

To overcome this limitation, we introduce constraints that are optimally sound for polar programs (Definition 2) based on polar subsumption (Definition 4). In the LFFN setting, Theorem 1 implies the following propositions:

Proposition 1 (Generalisation soundness). *Let $(E^+, E^-, B, \mathcal{H}, C)$ be a LFFN input, $H_1, H_2 \in \mathcal{H}_C$, H_1 be inconsistent, and $H_2 \preccurlyeq_\diamond H_1$. Then H_2 is not a solution.*

Proposition 2 (Specialisation soundness). *Let $(E^+, E^-, B, \mathcal{H}, C)$ be a LFFN input, $H_1, H_2 \in \mathcal{H}_C$, H_1 be incomplete, and $H_1 \preccurlyeq_\diamond H_2$. Then H_2 is not a solution.*

To summarise, *polar subsumption* allows us to soundly prune the hypothesis space when learning non-monotonic programs. In this next section, we introduce an algorithm that uses *polar subsumption* to efficiently learn polar programs.

Algorithm

We now describe our NOPI algorithm. To aid our explanation, we first describe POPPER (Cropper and Morel 2021).

POPPER. POPPER takes as input an LFF input[1] and learns hypotheses as definite programs without NAF. To generate hypotheses, POPPER uses an ASP program P where each model (answer set) of P represents a hypothesis. POPPER follows a *generate*, *test*, and *constrain* loop to find a solution. First, it generates a hypothesis as a solution to P with the ASP system Clingo (Gebser et al. 2019). Then, POPPER tests this hypothesis given the background knowledge against the training examples, typically using Prolog. If the hypothesis is a solution, POPPER returns it. Otherwise, the hypothesis is a failure: POPPER identifies the kind of failure and builds constraints accordingly. For instance, if the hypothesis is inconsistent, POPPER builds a generalisation constraint. POPPER adds these constraints to P to constrain subsequent *generate* steps. This loop repeats until the solver finds a solution or there are no more models of P.

NOPI

NOPI builds on POPPER and follows a *generate*, *test*, and *constrain* loop. The two key novelties of NOPI are its ability to (i) learn polar programs and (ii) use non-monotonic generalisation and specialisation constraints to efficiently prune the hypothesis space. We describe these advances in turn.

Polar Programs To learn polar programs, we extend the *generate* ASP program to generate normal logic programs, i.e. programs with negative literals. To only generate polar programs, we add the rules and constraints of Definitions 1 and 2 to the ASP program to eliminate models where a predicate symbol has multiple polarities. The complete ASP encoding is in Appendix F, but we briefly explain it at a high-level. If a predicate symbol p occurs in the body of a rule with head symbol q we say q *calls* p, which we name the *call relation*. A predicate can be called positively or negatively. We compute the transitive closure of the call relation tracking the number of negative calls on each path. If a symbol has an even number of negative calls on a path to a top symbol we say its associated rules are positive, otherwise they are negative. If any rule is labeled both positive and negative then the program is non-polar. We ignore background knowledge predicates when computing the call relation.

Polar Constraints NOPI uses two types of constraints to prune models and thus prune hypotheses. We refer to these constraints as *polar specialisation* and *polar generalisation* constraints. These constraints differ from those used by POPPER because (i) they use additional literals to assign polarity to rules, and (ii) they use *polar subsumption* (Definition 4) rather than standard subsumption. Polarity is important when learning polar programs because a *polar generalisation* constraint prunes generalisations of positive polarity rules and specialisations of negative polarity rules. A polar specialisation constraint prunes the specialisations of positive polarity rules and generalisations of negative polarity rules.

Example 9. *Reconsider the Zendo scenario from the introduction (Figures 1 and 2). The following hypothesis is incomplete as every positive example contains at least one cone:*

$$h_1 = \left\{ \begin{array}{l} r_1 : \mathrm{f(S)} \leftarrow scene(S), \textbf{not}\ inv_1(S) \\ r_2 :\ inv_1(S) \leftarrow cone(S,A) \end{array} \right\}$$

Since h_1 is incomplete, we can use a polar specialisation constraint to prune the hypothesis:

$$h_2 = \left\{ \begin{array}{l} r_1 : \mathrm{f(S)} \leftarrow scene(S), \textbf{not}\ inv_1(S) \\ r_2 :\ inv_1\mathrm{(S)} \leftarrow cone(S,A) \\ r_3 :\ inv_1\mathrm{(S)} \leftarrow contact(S,A,\_), red(A), \\ \qquad \textbf{not}\ blue(A) \end{array} \right\}$$

The hypothesis h_2 is a superset of h_1 as it includes the additional rule r_3. In h_2, the symbol inv_1 is negative because it is used negatively in r_1. Therefore, the rule r_3 implies that h_2 is a specialisation of h_1.

The polar specialisation constraint also prunes h_3:

$$h_3 = \left\{ \begin{array}{l} r_4 : \mathrm{f(S)} \leftarrow scene(S), \textbf{not}\ inv_1(S), \\ \qquad\qquad contact(S,A,\_), red(A), blue(A) \\ r_2 :\ inv_1\mathrm{(S)} \leftarrow cone(S,A) \end{array} \right\}$$

The rule r_4 in h_3 adds literals to r_1 in h_1, so h_3 is a specialisation of h_1. By contrast, a polar specialisation constraint does not prune the following hypothesis:

$$h_4 = \left\{ \begin{array}{l} r_1 : \mathrm{f(S)} \leftarrow scene(S), \textbf{not}\ inv_1(S) \\ r_5 :\ inv_1\mathrm{(S)} \leftarrow cone(S,A), \textbf{not}\ red(A) \end{array} \right\}$$

Notice that the rule r_5 has an additional negated body literal compared to r_2. The symbol of this literal is neither positive nor negative, so we can ignore the occurrence of **not**. *Thus, h_4 is a generalisation of h_1 as the new literal occurs in the body of r_2 whose head symbol has negative polarity.*

Experiments

To evaluate the impact of combining negation and PI, our experiments aim to answer the question:

Q1 Can negation and PI improve learning performance?

To answer **Q1**, we compare the performance of NOPI against POPPER, which cannot negate invented predicate symbols. Comparing NOPI against different systems with different biases will not allow us to answer the question, as we would be unable to identify the reason for any performance difference. To answer **Q1**, we use tasks where negation and PI should be helpful, such as learning the rules of Zendo (Bramley et al. 2018). We describe the tasks in the next section.

We introduced sound constraints for polar programs to prune non-optimal solutions from the hypothesis space. To evaluate whether these constraints improve performance, our experiments aim to answer the question:

Q2 Can polar constraints improve learning performance?

To answer **Q2**, we compare the performance of NOPI with and without these constraints.

We introduced NOPI to go beyond existing approaches by combining negation and PI. Our experiments, therefore, aim to answer the question:

Q3 How does NOPI compare against existing approaches?

[1]An LFF input is the same as an LFFN input except the hypothesis space contains only definite (monotonic) programs.

To answer **Q3**, we compare NOPI against POPPER, ALEPH, and METAGOL$_{SN}$. We describe these systems below.

Questions **Q1-Q3** focus on tasks where negation and PI should help. However, negation and PI are not always necessary. In such cases, can negation and PI be harmful? Our experiments try to answer the question:

Q4 Can negation and PI degrade learning performance?

To answer **Q4**, we evaluate NOPI on tasks where negation and PI should be unnecessary.

Reproducibility. Experimental code may be found in the following repository: *github.com/Ermine516/NOPI*

Domains We briefly describe our domains. The precise problems are found in Appendix D.

Basic (B). Non-monotonic learning tasks introduced by Siebers and Schmid (2018) and Purgał, Cerna, and Kaliszyk (2022), such as learning the definition of a leap year.

Zendo (Z). Bramley et al. (2018) introduce *Zendo* tasks similar to the one in the introduction.

Graphs (G). We use commonly used graph problems (Evans and Grefenstette 2018; Glanois et al. 2022), such as *dominating set*, *independent set*, and *connectedness*.

Sets (S). These set-based tasks include *symmetric difference*, *decomposition into subsets*, and *mutual distinctness*.

Systems We compare NOPI against POPPER (Cropper and Morel 2021; Cropper and Hocquette 2023), ALEPH (Srinivasan 2001), and METAGOL$_{SN}$ (Siebers and Schmid 2018). We give NOPI and POPPER identical input. The only experimental difference is the ability of NOPI to negate invented predicate symbols. ALEPH can learn normal logic programs but uses a different bias than NOPI so the comparison should be interpreted as indicative only. Also, we use the default ALEPH settings, but there are likely to be better settings on these datasets (Srinivasan and Ramakrishnan 2011). METAGOL$_{SN}$ can learn normal logic programs but requires metarules to define the hypothesis space. We use the metarules used by Siebers and Schmid (2018) supplemented with a general set of metarules (Cropper and Tourret 2020).

Experimental Setup We use a 300s learning timeout for each task and round accuracies and learning times to integer values. We plot 99% confidence intervals. Additional experimental details are in Appendix B.

Q1. We allow all the systems to negate the given background relations. For instance, in the Zendo tasks, each system can negate colours such *red*. Therefore, any improvements from NOPI are not from the use of negation but from the combination of negation and PI.

Q2. We need a baseline to evaluate our polar constraints. As discussed in the *Problem Setting* section, POPPER uses unsound constraints when learning polar programs. If a program h is not a solution and has a negated invented symbol, the only sound option for POPPER is to prune h from the hypothesis space, but, importantly, not its generalisations or specialisations. To evaluate our polar constraints, we compare them against this simpler (banish) approach, which we call NOPI$_{bn}$. In other words, to answer **Q2**, we compare NOPI against NOPI$_{bn}$.[2]

Results

Q1. Can negation and PI improve performance? Table 1 shows the predictive accuracies of NOPI and POPPER. The results show that NOPI vastly outperforms POPPER regarding predictive accuracies. For instance, for *all red* (Z2) POPPER learns:

$$\left\{\begin{array}{l} \textit{z(A)}\leftarrow \textit{piece(A,B), contact(B,C), red(C), rhs(C)} \\ \textit{z(A)}\leftarrow \textit{piece(A,B), contact(B,C), upright(C), lhs(B)} \\ \textit{z(A)}\leftarrow \textit{piece(A,B), contact(B,C), lhs(C), lhs(B)} \\ \textit{z(A)}\leftarrow \textit{piece(A,B), coord1(B,C), size(B,C), upright(B)} \end{array}\right\}$$

By contrast, NOPI learns:

$$\left\{\begin{array}{l} \textit{z(A)}\leftarrow \textit{scene(A)}, \textbf{\textit{not}}\ inv_1(A) \\ inv_1(A) \leftarrow \textit{piece(A,B)}, \textbf{\textit{not}}\ \textit{red(B)} \end{array}\right\}$$

Table 2 shows the corresponding learning times. The results show that NOPI rarely needs more than 40s to learn a solution. One of the more difficult problems (30s to learn) is *largest is red* (Z6), which involves inventing two predicate symbols and having two layers of negation, which, as far as we are aware, goes beyond anything in the existing literature:

$$\left\{\begin{array}{l} \textit{zendo(A)} \leftarrow \textit{scene(A), piece(A,B)}, \textbf{\textit{not}}\ inv_1(B,A) \\ inv_1(A,B) \leftarrow \textit{piece(B,C), size(C,D)}, \textbf{\textit{not}}\ inv_2(D,A) \\ inv_2(A,B) \leftarrow \textit{size(B,C), red(B)},\ A \leq C \end{array}\right\}$$

POPPER sometimes terminates in less than a second. The reason is that on some problems, because of its highly efficient search, POPPER almost immediately proves that there is no monotonic solution.

Overall, the results from this section suggest that the answer to **Q1** is that combining negation and PI can drastically improve learning performance.

Q2. Can polar constraints improve performance? Table 3 shows the learning times of NOPI and NOPI$_{bn}$. The results show that NOPI has lower learning times than NOPI$_{bn}$. In other words, the results show that polar constraints can drastically reduce learning times. A *wilcoxon signed-rank test* confirms the significance of the differences at the $p < 10^{-8}$ value. For simpler tasks, there is little benefit from the polar constraints as the overhead of constructing and adding them to the solver negates the pruning benefits. For more difficult tasks, the difference is substantial. For instance, the learning times for NOPI and NOPI$_{bn}$ on the *sym. difference* (S4) task are 31s and 72s respectively, a 57% reduction. Overall, the results suggest that the answer to **Q2** is that our polar constraints can drastically reduce learning times.

Q3. How does NOPI compare against existing approaches? Table 1 shows the predictive accuracies of the systems. As is clear, NOPI overwhelmingly outperforms the other systems. This result is expected. Besides METAGOL$_{SN}$, the other systems cannot learn normal logic programs with PI. ALEPH can learn programs with NAF and sometimes learns reasonable solutions. However, ALEPH cannot perform PI so,

[2]Appendix Table 6 compares sound and unsound constraints.

Task	NOPI	POPPER	ALEPH	METAGOL$_{SN}$
B1	**100 ± 0**	82 ± 0	50 ± 0	0 ± 0
B2	**100 ± 0**	0 ± 0	50 ± 0	**100 ± 0**
B3	**100 ± 0**	82 ± 0	82 ± 0	**100 ± 0**
Z1	**100 ± 0**	0 ± 0	60 ± 0	0 ± 0
Z2	**100 ± 0**	55 ± 0	67 ± 0	0± 0
Z3	**100 ± 0**	0 ± 0	65 ± 0	0 ± 0
Z4	**100 ± 0**	55 ± 0	58 ± 0	0± 0
Z5	**100 ± 0**	0 ± 0	21 ± 0	0 ± 0
Z6	**100 ± 0**	0 ± 0	45 ± 0	0 ± 0
G1	**100 ± 0**	0 ± 0	50 ± 0	0 ± 0
G2	**100 ± 0**	24 ± 0	47 ± 0	0 ± 0
G3	**100 ± 0**	0 ± 0	12 ± 0	0 ± 0
G4	**100 ± 0**	20 ± 0	**100 ± 0**	0 ± 0
G5	**100 ± 0**	0 ± 0	50 ± 0	0 ± 0
G6	**100 ± 0**	0 ± 0	21 ± 0	0 ± 0
G7	**100 ± 0**	0 ± 0	50 ± 0	0 ± 0
G8	**100 ± 0**	0 ± 0	50 ± 0	0 ± 0
S1	**100 ± 0**	0 ± 0	50 ± 0	0 ± 0
S2	**100 ± 0**	0 ± 0	50 ± 0	0 ± 0
S3	**92 ± 0**	0 ± 0	57 ± 0	0 ± 0
S4	**100 ± 0**	0 ± 0	50 ± 0	0 ± 0
S5	**100 ± 0**	57 ± 0	23 ± 0	0 ± 0
S6	**100 ± 0**	0 ± 0	0 ± 0	0 ± 0

Table 1: Mean predictive accuracies (10 runs).

due to its restricted language, it struggles to generalise. In many cases, ALEPH simply memorises the training examples. Because it relies on user-supplied metarules, METAGOL$_{SN}$ can only learn normal logic programs of a very restricted syntactic structure and thus struggles on almost all our tasks. Overall, the results from this section suggest that the answer to **Q3** is that NOPI performs well compared to other approaches on problems that need negation and PI.

Q4. Can negation and PI degrade performance? The Appendix includes tables showing the predictive accuracies and learning times of the systems. The results show that NOPI performs worse than POPPER on these tasks. The Blumer bound (Blumer et al. 1987) helps explain why. According to the bound, given two hypotheses spaces of different sizes, searching the smaller space should result in higher predictive accuracy compared to searching the larger one if the target hypothesis is in both. NOPI considers programs with negation and PI and thus searches a drastically larger hypothesis space than POPPER and the other systems. The tasks in **Q4** do not need negation and PI, thus explaining the difference.

Conclusions and Limitations

We have introduced an approach that combines negation and PI. Our approach can learn *polar* programs, including stratified Datalog programs (Theorem 2). We introduced generalisation and specialisation constraints for this non-monotonic fragment and showed that they are optimally sound (Theorem 1). We introduced NOPI, an ILP system that can learn normal logic programs with PI, including recursive programs. We have empirically shown on multiple domains that (i) NOPI can outperform existing approaches, and (ii) our non-monotonic constraints can reduce learning times.

Task	NOPI	POPPER	ALEPH	METAGOL$_{SN}$
B1	20 ± 0	*timeout*	20 ± 0	*timeout*
B3	3 ± 0	0 ± 0	18 ± 2	1 ± 0
Z1	2 ± 0	0 ± 0	7± 1	*timeout*
Z2	12 ± 0	1 ± 0	95 ± 2	*timeout*
Z3	2 ± 0	0 ± 0	27 ± 1	*timeout*
Z4	22 ± 1	0 ± 0	20 ± 1	*timeout*
Z5	15 ± 1	0 ± 0	24 ± 1	*timeout*
Z6	67 ± 4	0 ± 0	149 ± 24	*timeout*
G1	4 ± 0	0 ± 0	32 ± 2	*timeout*
G2	2 ± 0	0 ± 0	0 ± 0	*timeout*
G3	9 ± 0	16 ± 0	1 ± 0	*timeout*
G4	12 ± 0	0 ± 0	1 ± 0	*timeout*
G5	8 ± 0	0 ± 0	0 ± 0	*timeout*
G6	19 ± 3	0 ± 0	12 ± 1	*timeout*
G7	58 ± 8	0 ± 0	12 ± 1	*timeout*
G8	71 ± 9	0 ± 0	38 ± 1	*timeout*
S2	3 ± 0	0 ± 0	1 ± 0	*timeout*
S4	28 ± 2	0 ± 0	0 ± 2	0 ± 0
S5	43 ± 3	0 ± 0	23 ± 3	*timeout*
S6	3 ± 0	0 ± 0	1 ± 0	*timeout*

Table 2: Mean learning times (10 runs). We show tasks where the times of NOPI and POPPER differ by more than 1 second.

Task	NOPI	NOPI$_{bn}$	Change
B3	2 ± 0	4 ± 0	**-50%**
Z4	11 ± 1	49 ± 2	**-78%**
Z5	13 ± 1	29 ± 1	**-55%**
Z6	30 ± 1	115 ± 9	**-74%**
G2	1 ± 0	9 ± 0	**-89%**
G3	18 ± 1	23 ± 1	**-22%**
G4	20 ± 3	68 ± 4	**-71%**
G5	3 ± 0	11 ± 0	**-73%**
G6	23 ± 1	33 ± 3	**-27%**
G7	56 ± 5	93 ± 9	**-40%**
G8	63 ± 5	103 ± 9	**-39%**
S3	3 ± 0	13 ± 0	**-77%**
S4	31 ± 2	72 ± 9	**-57%**
S5	35 ± 2	53 ± 5	**-34%**
S6	4 ± 0	8 ± 1	**-50%**

Table 3: Mean learning time (125 runs). We show tasks where the times differ by more than 1 second.

Limitations and Future Work

Inefficient constraints. NOPI sometimes spends 30% of learning time building polar constraints. This inefficiency is an implementation issue rather than a theoretical one. Therefore, our empirical results likely underestimate the performance of NOPI, especially the improvements from using polar constraints.

Unnecessary negation and PI. Our results show that combining negation and PI allows NOPI to learn programs that other approaches cannot. However, the results also show that this increased expressivity can be detrimental when the combination of negation and PI is unnecessary. Thus, the main limitation of this work and direction for future work is to automatically detect when a problem needs negation and PI.

Acknowledgements

The first author is supported by 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 Czech Science Foundation Grant No. 22-06414L, *PANDAFOREST*. The second author is supported by the EPSRC fellowship *The Automatic Computer Scientist* (EP/V040340/1). We thank Filipe Gouveia, Céline Hocquette, and Oghenejokpeme Orhobor for feedback on the paper.

References

Apt, K. R.; and Blair, H. A. 1991. Arithmetic classification of perfect models of stratified programs. *Fundam. Informaticae*, 14(3): 339–343.

Bekker, J.; and Davis, J. 2020. Learning from positive and unlabeled data: a survey. *Mach. Learn.*, 109(4): 719–760.

Bembenek, A.; Greenberg, M.; and Chong, S. 2023. From SMT to ASP: Solver-Based Approaches to Solving Datalog Synthesis-as-Rule-Selection Problems. *Proc. ACM Program. Lang.*, 7(POPL): 185–217.

Blockeel, H.; and De Raedt, L. 1998. Top-Down Induction of First-Order Logical Decision Trees. *Artif. Intell.*, 101(1-2): 285–297.

Blumer, A.; Ehrenfeucht, A.; Haussler, D.; and Warmuth, M. K. 1987. Occam's Razor. *Inf. Process. Lett.*, 24(6): 377–380.

Bramley, N.; Rothe, A.; Tenenbaum, J.; Xu, F.; and Gureckis, T. M. 2018. Grounding Compositional Hypothesis Generation in Specific Instances. In *Proceedings of the 40th Annual Meeting of the Cognitive Science Society, CogSci 2018.*

Clark, K. L. 1977. Negation as Failure. In *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977*, 293–322. New York.

Corapi, D.; Russo, A.; and Lupu, E. 2011. Inductive Logic Programming in Answer Set Programming. In *Inductive Logic Programming - 21st International Conference*, volume 7207, 91–97.

Cropper, A.; Dumancic, S.; Evans, R.; and Muggleton, S. H. 2022. Inductive logic programming at 30. *Mach. Learn.*, 111(1): 147–172.

Cropper, A.; and Hocquette, C. 2023. Learning Logic Programs by Combining Programs. In *ECAI 2023 - 26th European Conference on Artificial Intelligence*, volume 372 of *Frontiers in Artificial Intelligence and Applications*, 501–508. IOS Press.

Cropper, A.; and Morel, R. 2021. Learning programs by learning from failures. *Mach. Learn.*, 110(4): 801–856.

Cropper, A.; and Muggleton, S. H. 2019. Learning efficient logic programs. *Mach. Learn.*, 108(7): 1063–1083.

Cropper, A.; and Tourret, S. 2020. Logical reduction of metarules. *Mach. Learn.*, 109(7): 1323–1369.

Dai, W.; and Muggleton, S. 2021. Abductive Knowledge Induction from Raw Data. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, 1845–1851.

Dantsin, E.; Eiter, T.; Gottlob, G.; and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3): 374–425.

Dimopoulos, Y.; and Kakas, A. C. 1995. Learning Non-Monotonic Logic Programs: Learning Exceptions. In *Machine Learning: ECML-95, 8th European Conference on Machine Learning 1995*, volume 912.

Ellis, K.; Morales, L.; Sablé-Meyer, M.; Solar-Lezama, A.; and Tenenbaum, J. 2018. Learning Libraries of Subroutines for Neurally-Guided Bayesian Program Induction. In *NeurIPS 2018*, 7816–7826.

Evans, R.; and Grefenstette, E. 2018. Learning Explanatory Rules from Noisy Data. *J. Artif. Intell. Res.*, 61: 1–64.

Ferilli, S. 2016. Predicate invention-based specialization in Inductive Logic Programming. *J. Intell. Inf. Syst.*, 47(1): 33–55.

Fogel, L.; and Zaverucha, G. 1998. Normal Programs and Multiple Predicate Learning. In Page, D., ed., *Inductive Logic Programming, 8th International Workshop, ILP-98, Madison, Wisconsin, USA, July 22-24, 1998, Proceedings*, volume 1446 of *Lecture Notes in Computer Science*, 175–184. Springer.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. *Answer Set Solving in Practice*. Morgan & Claypool Publishers.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.*, 19(1): 27–82.

Glanois, C.; Jiang, Z.; Feng, X.; Weng, P.; Zimmer, M.; Li, D.; Liu, W.; and Hao, J. 2022. Neuro-Symbolic Hierarchical Rule Induction. In *International Conference on Machine Learning, ICML 2022*, volume 162, 7583–7615. PMLR.

Hocquette, C.; and Muggleton, S. H. 2020. Complete Bottom-Up Predicate Invention in Meta-Interpretive Learning. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, 2312–2318.

Inoue, K.; and Kudoh, Y. 1997. Learning Extended Logic Programs. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*, 176–181.

Inoue, K.; Ribeiro, T.; and Sakama, C. 2014. Learning from interpretation transition. *Mach. Learn.*, 94(1): 51–79.

Kaminski, T.; Eiter, T.; and Inoue, K. 2019. Meta-Interpretive Learning Using HEX-Programs. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019*, 6186–6190.

Law, M.; Russo, A.; and Broda, K. 2014. Inductive Learning of Answer Set Programs. In *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014*, volume 8761, 311–325.

Lloyd, J. W. 2012. *Foundations of logic programming*. Springer Science & Business Media.

Muggleton, S. 1991. Inductive Logic Programming. *New Generation Computing*, 8(4): 295–318.

Muggleton, S. 1995. Inverse Entailment and Progol. *New Generation Comput.*, 13(3&4): 245–286.

Muggleton, S. H.; Lin, D.; and Tamaddoni-Nezhad, A. 2015. Meta-interpretive learning of higher-order dyadic Datalog: predicate invention revisited. *Mach. Learn.*, 100(1): 49–73.

Plotkin, G. 1971. *Automatic Methods of Inductive Inference*. Ph.D. thesis, Edinburgh University.

Purgał, S. J.; Cerna, D. M.; and Kaliszyk, C. 2022. Learning Higher-Order Logic Programs From Failures. In *IJCAI 2022*.

Quinlan, J. R. 1990. Learning Logical Definitions from Relations. *Mach. Learn.*, 5: 239–266.

Raghothaman, M.; Mendelson, J.; Zhao, D.; Naik, M.; and Scholz, B. 2020. Provenance-guided synthesis of Datalog programs. *Proc. ACM Program. Lang.*, 4(POPL): 62:1–62:27.

Ray, O. 2009. Nonmonotonic abductive inductive learning. *J. Applied Logic*, 7(3): 329–340.

Sakama, C. 2001. Nonmonotonic Inductive Logic Programming. In Eiter, T.; Faber, W.; and Truszczynski, M., eds., *Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001, Vienna, Austria, September 17-19, 2001, Proceedings*, volume 2173 of *Lecture Notes in Computer Science*, 62–80. Springer.

Sakama, C.; and Inoue, K. 2009. Brave induction: a logical framework for learning from incomplete information. *Mach. Learn.*, 76(1): 3–35.

Shapiro, E. Y. 1983. *Algorithmic Program DeBugging*. Cambridge, MA, USA: MIT Press. ISBN 0262192187.

Si, X.; Raghothaman, M.; Heo, K.; and Naik, M. 2019. Synthesizing Datalog Programs using Numerical Relaxation. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019*, 6117–6124.

Siebers, M.; and Schmid, U. 2018. Was the Year 2000 a Leap Year? Step-Wise Narrowing Theories with Metagol. In Riguzzi, F.; Bellodi, E.; and Zese, R., eds., *ILP 2018*, 141–156. Springer International Publishing.

Silver, T.; Chitnis, R.; Kumar, N.; McClinton, W.; Lozano-Perez, T.; Kaelbling, L. P.; and Tenenbaum, J. 2022. Predicate Invention for Bilevel Planning.

Srinivasan, A. 2001. The ALEPH manual. *Machine Learning at the Computing Laboratory, Oxford University*.

Srinivasan, A.; Muggleton, S.; and Bain, M. 1992. Distinguishing exceptions from noise in non-monotonic learning. In *Proceedings of the Second Inductive Logic Programming Workshop*, 97–107. Tokyo.

Srinivasan, A.; and Ramakrishnan, G. 2011. Parameter Screening and Optimisation for ILP using Designed Experiments. *J. Mach. Learn. Res.*, 12: 627–662.

Stahl, I. 1995. The Appropriateness of Predicate Invention as Bias Shift Operation in ILP. *Mach. Learn.*, 20(1-2): 95–117.

Zeng, Q.; Patel, J. M.; and Page, D. 2014. QuickFOIL: Scalable Inductive Logic Programming. *Proc. VLDB Endow.*, 8(3): 197–208.