



Learning programs by learning from failures

Andrew Cropper¹  · Rolf Morel¹

Received: 5 May 2020 / Revised: 22 November 2020 / Accepted: 2 December 2020 /
Published online: 19 February 2021
© The Author(s) 2021

Abstract

We describe an inductive logic programming (ILP) approach called *learning from failures*. In this approach, an ILP system (the learner) decomposes the learning problem into three separate stages: *generate*, *test*, and *constrain*. In the generate stage, the learner generates a hypothesis (a logic program) that satisfies a set of *hypothesis constraints* (constraints on the syntactic form of hypotheses). In the test stage, the learner tests the hypothesis against training examples. A hypothesis *fails* when it does not entail all the positive examples or entails a negative example. If a hypothesis fails, then, in the constrain stage, the learner learns constraints from the failed hypothesis to prune the hypothesis space, i.e. to constrain subsequent hypothesis generation. For instance, if a hypothesis is too general (entails a negative example), the constraints prune generalisations of the hypothesis. If a hypothesis is too specific (does not entail all the positive examples), the constraints prune specialisations of the hypothesis. This loop repeats until either (i) the learner finds a hypothesis that entails all the positive and none of the negative examples, or (ii) there are no more hypotheses to test. We introduce Popper, an ILP system that implements this approach by combining answer set programming and Prolog. Popper supports infinite problem domains, reasoning about lists and numbers, learning textually minimal programs, and learning recursive programs. Our experimental results on three domains (toy game problems, robot strategies, and list transformations) show that (i) constraints drastically improve learning performance, and (ii) Popper can outperform existing ILP systems, both in terms of predictive accuracies and learning times.

Keywords Inductive logic programming · Program synthesis · Program induction

Editors: Nikos Katzouris, Alexander Artikis, Luc De Raedt, Artur d’Avila Garcez, Sebastijan Dumančić, Ute Schmid, Jay Pujara.

✉ Andrew Cropper
andrew.cropper@cs.ox.ac.uk

Rolf Morel
rolf.morel@cs.ox.ac.uk

¹ University of Oxford, Oxford, UK

1 Introduction

Inductive logic programming (ILP) (Muggleton 1991) is a form of machine learning. Given examples of a target predicate and background knowledge (BK), the ILP problem is to induce a hypothesis which, with the BK, correctly generalises the examples. A key characteristic of ILP is that it represents the examples, BK, and hypotheses as logic programs (sets of logical rules).

Compared to most machine learning approaches, ILP has several advantages (Cropper et al. 2020). ILP systems can generalise from small numbers of examples, often a single example (Lin et al. 2014). Because hypotheses are logic programs, they can be read by humans, crucial for explainable AI and ultra-strong machine learning (Michie 1988). Finally, because of their symbolic nature, ILP systems naturally support lifelong and transfer learning (Cropper 2020), which is considered essential for human-like AI (Lake et al. 2016).

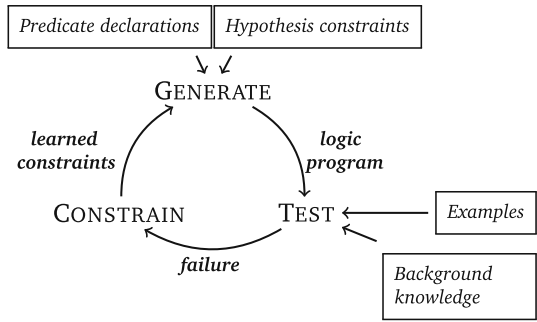
The fundamental problem in ILP is to efficiently search a large hypothesis space (the set of all hypotheses). A popular ILP approach is to use a set covering algorithm to learn hypotheses one clause at-a-time (Quinlan 1990; Muggleton 1995; Blockeel and De Raedt 1998; Srinivasan 2001; Ahlgren and Yuen 2013). Systems that implement this approach are often efficient because they are example-driven. However, these systems tend to learn overly specific solutions and struggle to learn recursive programs (Bratko 1999; Cropper et al. 2020). An alternative, but increasingly popular, approach is to encode the ILP problem as an answer set programming (ASP) problem (Corapi et al. 2011; Law et al. 2014; Schüller and Benz 2018; Kaminski et al. 2018; Evans et al. 2019). Systems that implement this approach can often learn optimal and recursive programs and can harness state-of-the-art ASP solvers, but often struggle with scalability, especially in terms of the problem domain size.

In this paper, we describe an ILP approach called *learning from failures* (LFF). In this approach, the learner (an ILP system) decomposes the ILP problem into three separate stages: *generate*, *test*, and *constrain*. In the generate stage, the learner generates a hypothesis (a logic program) that satisfies a set of *hypothesis constraints* (constraints on the syntactic form of hypotheses). In the test stage, the learner tests a hypothesis against training examples. A hypothesis *fails* when it does not entail all the positive examples or entails a negative example. If a hypothesis fails, then, in the constrain stage, the learner learns hypothesis constraints from the failed hypothesis to prune the hypothesis space, i.e. to constrain subsequent hypothesis generation.

Compared to other approaches that employ a generate/test/constrain loop (Law 2018), a key idea in this paper is to use theta-subsumption (Plotkin 1971) to translate a failed hypothesis into a set of constraints. For instance, if a hypothesis is too general (entails a negative example), the constraints prune generalisations of the hypothesis. If a hypothesis is too specific (does not entail all the positive examples), the constraints prune specialisations of the hypothesis. This loop repeats until either (i) the learner finds a *solution* (a hypothesis that entails all the positive examples and none of the negative examples), or (ii) there are no more hypotheses to test. Figure 1 illustrates this loop.

Example 1 (*Learning from failures*) To illustrate our approach, consider learning a *last/2* hypothesis to find the last element of a list. For simplicity, assume an initial hypothesis space \mathcal{H}_1 :

Fig. 1 The generate, test, and constrain loop



$$\mathcal{H}_1 = \left\{ \begin{array}{l} h_1 = \{ \text{last}(A, B):- \text{head}(A, B). \} \\ h_2 = \{ \text{last}(A, B):- \text{head}(A, B), \text{empty}(A). \} \\ h_3 = \{ \text{last}(A, B):- \text{tail}(A, C), \text{head}(C, B). \} \\ h_4 = \{ \text{last}(A, B):- \text{reverse}(A, C), \text{head}(C, B). \} \\ h_5 = \{ \text{last}(A, B):- \text{head}(A, B), \text{reverse}(A, C), \text{head}(C, B). \} \\ h_6 = \left\{ \begin{array}{l} \text{last}(A, B):- \text{tail}(A, C), \text{head}(C, B). \\ \text{last}(A, B):- \text{reverse}(A, C), \text{head}(C, B). \end{array} \right\} \\ h_7 = \left\{ \begin{array}{l} \text{last}(A, B):- \text{tail}(A, C), \text{head}(C, B). \\ \text{last}(A, B):- \text{tail}(A, C), \text{tail}(C, D), \text{head}(D, B). \end{array} \right\} \\ h_8 = \left\{ \begin{array}{l} \text{last}(A, B):- \text{reverse}(A, C), \text{tail}(C, D), \text{head}(D, B). \\ \text{last}(A, B):- \text{tail}(A, C), \text{reverse}(C, D), \text{head}(D, B). \end{array} \right\} \end{array} \right.$$

Also assume we have the positive (E^+) and negative (E^-) examples:

$$E^+ = \left\{ \begin{array}{l} \text{last}([l, a, u, r, a], a). \\ \text{last}([p, e, n, e, l, o, p, e], e). \end{array} \right\} \quad E^- = \left\{ \begin{array}{l} \text{last}([e, m, m, a], m). \\ \text{last}([j, a, m, e, s], e). \end{array} \right\}$$

In the generate stage, the learner generates a hypothesis:

$$h_1 = \{ \text{last}(A, B):- \text{head}(A, B). \}$$

In the test stage, the learner tests h_1 against the examples and finds that it *fails* because it does not entail any positive example and is therefore too *specific*. In the constrain stage, the learner learns hypothesis constraints to prune specialisations of h_1 (h_2 and h_5) from the hypothesis space. The hypothesis space is now:

$$\mathcal{H}_2 = \left\{ \begin{array}{l} h_3 = \{ \text{last}(A, B):- \text{tail}(A, C), \text{head}(C, B). \} \\ h_4 = \{ \text{last}(A, B):- \text{reverse}(A, C), \text{head}(C, B). \} \\ h_6 = \left\{ \begin{array}{l} \text{last}(A, B):- \text{tail}(A, C), \text{head}(C, B). \\ \text{last}(A, B):- \text{reverse}(A, C), \text{head}(C, B). \end{array} \right\} \\ h_7 = \left\{ \begin{array}{l} \text{last}(A, B):- \text{tail}(A, C), \text{head}(C, B). \\ \text{last}(A, B):- \text{tail}(A, C), \text{tail}(C, D), \text{head}(D, B). \end{array} \right\} \\ h_8 = \left\{ \begin{array}{l} \text{last}(A, B):- \text{reverse}(A, C), \text{tail}(C, D), \text{head}(D, B). \\ \text{last}(A, B):- \text{tail}(A, C), \text{reverse}(C, D), \text{head}(D, B). \end{array} \right\} \end{array} \right.$$

In the next generate stage, the learner generates another hypothesis:

$$h_3 = \{ \text{last}(A, B):- \text{tail}(A, C), \text{head}(C, B). \}$$

The learner tests h_3 against the examples and finds that it fails because it entails the negative example $\text{last}([e, m, m, a], m)$ and is therefore too *general*. The learner learns constraints to prune generalisations of h_3 (h_6 and h_7) from the hypothesis space. The hypothesis space is now:

$$\mathcal{H}_3 = \left\{ \begin{array}{l} h_4 = \{ \text{last}(A, B) :- \text{reverse}(A, C), \text{head}(C, B). \} \\ h_8 = \{ \text{last}(A, B) :- \text{reverse}(A, C), \text{tail}(C, D), \text{head}(D, B). \\ \text{last}(A, B) :- \text{tail}(A, C), \text{reverse}(C, D), \text{head}(D, B). \} \end{array} \right\}$$

The learner generates another hypothesis (h_4), tests it against the examples, finds that it does not fail, and returns it.

Whereas many ILP approaches iteratively refine a clause (Quinlan 1990; Muggleton 1995; De Raedt and Bruynooghe 1993; Blockeel and De Raedt 1998; Srinivasan 2001; Ahlgren and Yuen 2013) or refine a hypothesis (Shapiro 1983; Bratko 1999; Athakravi et al. 2013; Cropper and Muggleton 2016), our approach refines the *hypothesis space* through learned *hypothesis constraints*. In other words, LFF continually builds a set of constraints. The more constraints we learn, the more we reduce the hypothesis space. By reasoning about the hypothesis space, our approach can drastically prune large parts of the hypothesis space by testing a single hypothesis.

We implement our approach in Popper,¹ a new ILP system which combines ASP and Prolog. In the generate stage, Popper uses ASP to declaratively define, constrain, and search the hypothesis space. The idea is to frame the problem as an ASP problem where an answer set (a model) corresponds to a program, an approach also employed by other recent ILP approaches (Corapi et al. 2011; Law et al. 2014; Kaminski et al. 2018; Schüller and Benz 2018). By later learning hypothesis constraints, we eliminate answer sets and thus prune the hypothesis space. Our first motivation for using ASP is its declarative nature, which allows us to, for instance, define constraints to enforce Datalog and type restrictions, constraints to prune recursive hypotheses that do not contain base cases, and constraints to prune generalisations and specialisations of a failed hypothesis. Our second motivation is to use state-of-the-art ASP systems (Gebser et al. 2014) to efficiently solve our complex constraint problem. In the test stage, Popper uses Prolog to test hypotheses against the examples and BK. Our main motivation for using Prolog in this stage is to learn programs that use lists, numbers, and large domains. In the constrain stage, Popper learns hypothesis constraints (in the form of ASP constraints) from failed hypotheses to prune the hypothesis space, i.e. to constrain subsequent hypothesis generation. To efficiently combine the three stages, Popper uses ASP's multi-shot solving (Gebser et al. 2019) to maintain state between the three stages, e.g. to remember learned conflicts on the hypothesis space.

To give a clear overview of Popper, Table 1 compares Popper to Aleph (Srinivasan 2001), a classical ILP system, and Metagol (Cropper and Muggleton 2016), ILASP3 (Law 2018), and ∂ ILP (Evans and Grefenstette 2018), three state-of-the-art ILP systems based on Prolog, ASP, and neural networks respectively. Compared to Aleph, Popper can learn optimal and recursive programs.² Compared to Metagol, Popper does not need metarules (Cropper and Tourret 2020), so can learn programs with any arity predicates. Compared to ∂ ILP, Popper

¹ Popper is named after *Karl Popper*, whose idea of *falsification* (Popper 2005) inspired our approach, as it did Shapiro's MIS approach (Shapiro 1983). In fact, one can view our approach as Popper's idea of falsification, where a *failure* is a refutation/falsification. In other words, in our approach, a learner *deduces* what hypotheses *cannot* be true and prunes them from the hypothesis space, leaving only hypotheses not yet refuted.

² Aleph can learn recursive programs but struggles because it requires examples of both the base and inductive cases.

Table 1 A simplified comparison of ILP systems

	Aleph	Metagol	ILASP3	∂ ILP	Popper
Hypotheses	Normal	Definite	ASP	Datalog	Definite
Language bias	Modes	Metarules	Modes	Templates	Declarations
Predicate invention	No	Yes	Partly	Partly	No
Noise handling	Yes	No	Yes	Yes	No
Recursion	Partly	Yes	Yes	Yes	Yes
Optimality	No	Yes	Yes	Yes	Yes
Hypothesis constraints	No	No	No	No	Yes

Aleph can learn recursive programs but struggles because it requires examples of both the base and inductive cases. Metagol supports *automatic* predicate invention, whereas ILASP3 and ∂ ILP support *prescriptive* predicate invention (Law 2018), where the arity and argument types of an invented predicate must be specified by the given language bias

supports non-ground clauses as BK, so supports large and infinite domains. Compared to ILASP3, Popper does not need to ground a program, so scales better as the domain size grows (Sect. 5.2). Compared to all the systems, Popper supports hypothesis constraints, such as disallowing the co-occurrence of predicate symbols in a program, disallowing recursive hypotheses that do not contain base cases, or preventing subsumption redundant hypotheses.

ILASP3 (Law 2018) is the most similar ILP approach and also employs a generate/test/constrain loop. We discuss in detail the differences between ILASP3 and Popper in Sect. 2.6 but briefly summarise them now. ILASP3 learns ASP programs and can handle noise, whereas Popper learns Prolog programs and cannot currently handle noise. ILASP3 pre-computes every rule in the hypothesis space and therefore struggles to learn rules with many body literals (Sect. 5.1). By contrast, Popper does not pre-compute every rule, which allows it to learn rules with many body literals. With each iteration, ILASP3 finds the best hypothesis it can. If the hypothesis does not cover one of the examples, ILASP3 finds a reason why and then generates constraints to guide subsequent search.³ The constraints are boolean formulas over the rules in the hypothesis space, an approach that requires a set of pre-computed rules and the computation of which can be very expensive. Another way of viewing ILASP3 is that it uses a counter-example guided (Solar-Lezama et al. 2008) approach and translates an uncovered example e into a constraint that is satisfied if and only if e is covered. By contrast, the key idea of Popper is that when a hypothesis fails, Popper uses theta-subsumption (Plotkin 1971) to translate the *hypothesis* itself into a set of *hypothesis constraints* to rule out generalisations and specialisations of it, which does not need a set of pre-computed rules and which is substantially quicker to compute.

Overall our specific contributions in this paper are:

- We define the LFF problem, determine the size of the LFF hypothesis space, define hypothesis *generalisations* and *specialisations* based on theta-subsumption and show that they are sound with respect to optimal solutions (Sect. 3).
- We introduce Popper, an ILP system that learns definite programs (Sect. 4). Popper support types, learning optimal (textually minimal) solutions, learning recursive programs, reasoning about lists and infinite domains, and hypothesis constraints.

³ This statement covers the noiseless ILASP3 setting. Things are slightly more complicated in noisy tasks where examples are given penalties and ILASP3 may return a hypothesis that does not cover all examples, but is optimal with respect to the penalties.

- We experimentally show (Sect. 5) on three domains (toy game problems, robot strategies, and list transformations) that (i) constraints drastically reduce the hypothesis space, (ii) Popper scales well with respect to the optimal solution size, the number of background relations, the domain size, the number of training examples, and the size of the training examples, and (iii) Popper can substantially outperform existing ILP systems both in terms of predictive accuracies and learning times.

2 Related work

2.1 Inductive program synthesis

The goal of inductive program synthesis is to induce a program from a partial specification, typically input/output examples (Shapiro 1983). This topic interests researchers from many areas of computer science, notably machine learning (ML) and programming languages (PL). The major⁴ difference between ML and PL approaches is the generality of solutions (synthesised programs). PL approaches often aim to find *any* program that fits the specification, regardless of whether it generalises. Indeed, PL approaches rarely evaluate the ability of their systems to synthesise solutions that generalise, i.e. they do not measure predictive accuracy (Feser et al. 2015; Polikarpova et al. 2016; Albarghouthi et al. 2017; Feng et al. 2018; Raghothaman et al. 2020). By contrast, the major challenge in ML is learning hypotheses that *generalise* to unseen examples. Indeed, it is often trivial for an ML system to learn an overly specific solution for a given problem. For instance, an ILP system can trivially construct the bottom clause (Muggleton 1995) for each example. Because of this major difference, in the rest of this section, we focus on ML approaches to inductive program synthesis. We first, however, briefly cover two PL approaches, which share similarities to our learning from failures idea.

Neo (Feng et al. 2018) synthesises non-recursive programs using SMT encoded properties and a three staged loop. Neo inherently requires SMT encoded properties for domain specific functions (i.e. its background knowledge). For instance, their property for *head*, taking an *input* list and returning an *output* list, is the formula $input.size \geq 1 \wedge output.size = 1 \wedge output.max \leq input.max$. Neo's first stage builds up partially constructed programs. Its second stage uses SMT-based deduction on the properties of a partial program to detect inconsistency. The third stage determines related partial programs who must be inconsistent and can therefore be pruned. As it typically uses over-approximate properties, Neo can fail to detect inconsistency with the examples, in which case no programs get pruned. In contrast, our approach does not need any properties of background predicates. We only check whether a hypothesis entails the examples, always pruning specialisations and/or generalisations when the hypothesis fails. Neo cannot synthesise recursive programs, nor is it guaranteed to synthesise optimal (textually minimal) programs. By contrast, Popper can learn optimal and recursive logic programs.

ProSynth (Raghothaman et al. 2020) takes as input a set of candidate Datalog rules and returns a subset of them. ProSynth learns constraints that disallow certain clause combinations, e.g. to prevent clauses that entail a negative example from occurring together. Popper differs from ProSynth in several ways. ProSynth takes as input the full hypothesis space (the set of candidate rules). By contrast, Popper does not fully construct the hypothesis space. This difference is important because it is often infeasible to pre-compute the full hypothesis space.

⁴ Minor differences include the form of specification and noise handling.

For instance, the largest number of candidate rules considered in the ProSynth experiments is 1000. By contrast, in our first two experiments (Sect. 5.1), the hypothesis spaces contain approximately 10^6 and 10^{16} rules. ProSynth provides no guarantees about solution size. By contrast, Popper is guaranteed to learn an optimal (smallest) solution (Theorem 1). Moreover, whereas ProSynth synthesises Datalog programs, Popper additionally learns definite programs, and thus supports learning programs with infinite domains.

2.2 Inductive logic programming

There are various ML approaches to inductive program synthesis, including neural approaches (Balog et al. 2017; Ellis et al. 2018, 2019). We focus on inductive logic programming (ILP) (Muggleton 1991; Cropper and Dumancic 2020). As with other forms of ML, the goal of an ILP system is to learn a hypothesis that correctly generalises given training examples. However, whereas most forms of ML represent data (examples and hypotheses) as tables, ILP represents data as logic programs. Moreover, whereas most forms of ML learn *functions*, ILP learns *relations*.

Rather than refine a clause (Quinlan 1990; Muggleton 1995; De Raedt and Bruynooghe 1993; Blockeel and De Raedt 1998; Srinivasan 2001; Ahlgren and Yuen 2013), or a hypothesis (Shapiro 1983; Bratko 1999; Athakravi et al. 2013; Cropper and Muggleton 2016), our approach refines the *hypothesis space* through learned *hypothesis constraints*. In other words, in our approach continually builds a set of constraints. The more constraints we learn, the more we reduce the hypothesis space. By reasoning about the hypothesis space, our approach can drastically prune large parts of the hypothesis space by testing a single hypothesis.

Atom (Ahlgren and Yuen 2013) learns definite programs using SAT solvers and also learns constraints. However, because it builds on Progol Muggleton (1995), and thus employs inverse entailment, Atom struggles to learn recursive programs because it needs examples of both the base and step cases of a recursive program. For the same reason, Atom struggles to learn optimal solutions. By contrast, Popper can learn recursive and optimal solutions because it learns programs rather than individual clauses.

2.3 Recursion

Learning recursive programs has long been considered a difficult problem in ILP (Muggleton et al. 2012). Without recursion, it is often difficult for an ILP system to generalise from small numbers of examples (Cropper et al. 2015). Indeed, many popular ILP systems, such as FOIL (Quinlan 1990), Progol (Muggleton 1995), TILDE (Blockeel and De Raedt 1998), and Aleph (Srinivasan 2001) struggle to learn recursive programs. The reason is that they employ a set covering approach to build a hypothesis clause by clause. Each clause is usually found by searching an ordering over clauses. A common approach is to pick an uncovered example, generate the bottom clause (Muggleton 1995) for this example, the logically most specific clause that entails the example, and then to search the subsumption lattice (either top-down or bottom-up) bounded by this bottom clause. Systems that implement this approach are often efficient because the hypothesis search is example-driven. However, these systems tend to learn overly specific solutions and struggle to learn recursive programs (Bratko 1999; Cropper et al. 2020). To overcome this limitation, Popper searches over logic programs (sets of clauses), a technique used by other ILP systems (Bratko 1999; Athakravi et al. 2013; Law et al. 2014; Cropper and Muggleton 2016; Evans and Grefenstette 2018; Kaminski et al. 2018).

2.4 Optimality

There are often multiple (sometimes infinite) hypotheses that explain the data. Deciding which hypothesis to choose is a difficult problem. Many ILP systems (Muggleton 1995; Srinivasan 2001; Blockeel and De Raedt 1998; Ray 2009) are not guaranteed to learn optimal solutions, where optimal typically means the smallest program or the program with the minimal description length. The claimed advantage of learning optimal solutions is better generalisation. Recent meta-level ILP approaches often learn optimal solutions, such as programs with the fewest clauses (Muggleton et al. 2015; Cropper and Muggleton 2016; Kaminski et al. 2018) or literals (Corapi et al. 2011; Law et al. 2014). Popper also learns optimal solutions, measured as the total number of literals in the hypothesis.

2.5 Language bias

ILP approaches use a language bias (Nienhuys-Cheng and de Wolf 1997) to restrict the hypothesis space. Language bias can be categorised as *syntactic bias*, which restricts the syntax of hypotheses, such as the number of variables allowed in a clause, and *semantic bias*, which restricts hypotheses based on their semantics, such as whether they are functional, irreflexive, etc.

Mode declarations (Muggleton 1995) are a popular language bias (Blockeel and De Raedt 1998; Srinivasan 2001; Ray 2009; Corapi et al. 2010, 2011; Athakravi et al. 2013; Ahlgren and Yuen 2013; Law et al. 2014). Mode declarations state which predicate symbols may appear in a clause, how often they may appear, the types of their arguments, and whether their arguments must be ground. We do not use mode declarations. We instead use a simple language bias which we call *predicate declarations* (Sect. 3), where a user needs only state whether a predicate symbol may appear in the head or/and body of a clause. Predicate declarations are almost identical to determinations in Aleph (Srinivasan 2001). The only difference is a minor syntactic one. In addition to *predicate declarations*, a user can provide other language biases, such as type information, as *hypothesis constraints* (Sect. 2.7).

Metarules (Cropper and Tourret 2020) are another popular syntactic bias used by many ILP approaches (De Raedt and Bruynooghe 1992; Wang et al. 2014; Albarghouthi et al. 2017; Kaminski et al. 2018), including Metagol (Muggleton et al. 2015; Cropper et al. 2020; Cropper and Muggleton 2016) and, to an extent⁵, ∂ ILP (Evans and Grefenstette 2018). A metarule is a higher-order clause which defines the exact form of clauses in the hypothesis space. For instance, the *chain* metarule is of the form $P(A, B) \leftarrow Q(A, C), R(C, B)$, where P , Q , and R denote predicate variables, and allows for instantiated clauses such as $\text{last}(A, B) :- \text{reverse}(A, C), \text{head}(C, B)$. Compared with predicate (and mode) declarations, metarules are a much stronger inductive bias because they specify the exact form of clauses in the hypothesis space. However, the major problem with metarules is determining which ones to use (Cropper and Tourret 2020). A user must either (i) provide a set of metarules, or (ii) use a set of metarules restricted to a certain fragment of logic, e.g. dyadic Datalog (Cropper and Tourret 2020). This limitation means that ILP systems that use metarules are difficult to use, especially when the BK contains predicate symbols with arity greater than two. If suitable metarules are known, then, as we show in “Appendix A”, Popper can simulate metarules through hypothesis constraints.

⁵ ∂ ILP uses program templates to essentially generate sets of metarules.

2.6 Answer set programming

Much recent work in ILP uses ASP to learn Datalog (Evans et al. 2019), definite (Muggleton et al. 2014; Kaminski et al. 2018; Cropper and Dumancic 2020), normal (Ray 2009; Corapi et al. 2011; Athakravi et al. 2013), and answer set programs (Law et al. 2014). ASP is a declarative language that supports language features such as aggregates and weak and hard constraints. Most ASP solvers only work on ground programs (Gebser et al. 2014)⁶. Therefore, a major limitation of most pure ASP-based ILP systems is the intrinsic grounding problem, especially on large domains, such as reasoning about lists or numbers—most ASP implementations do not support lists nor real numbers. For instance, ILASP (Law et al. 2014) can represent real numbers as strings and delegate the reasoning to Python via Clingo’s scripting feature (Gebser et al. 2014). However, in this approach, the numeric computation is performed when grounding the inputs, so the grounding must be finite. Difficulty handling large (or infinite) domains is not specific to ASP. For instance, ∂ ILP uses a neural network to induce programs, but only works on BK formed of a finite set of ground atoms. To overcome this grounding limitation, Popper combines ASP and Prolog. Popper uses ASP to generate definite programs, which allows it to reason about large and infinite problem domains, such as reasoning about lists and real numbers.

ILASP3 (Law 2018) is a pure ASP-based ILP system that also employs a constrain loop. ILASP3 learns unstratified ASP programs, including programs with choice rules and weak and hard constraints, and can handle noise. By contrast, Popper learns Prolog programs, including programs operating over lists and real numbers, but cannot handle noise. ILASP3 pre-computes every clause in the hypothesis space defined by a set of given mode declarations. As we show in Experiment 1 (Sect. 5.1), this approach struggles to learn clauses with many body literals. By contrast, Popper does not pre-compute every clause, which allows it to learn clauses with many body literals. With each iteration, ILASP3 finds the best hypothesis it can. If the hypothesis does not cover one of the examples, ILASP3 finds a reason why and then generates constraints to guide subsequent search.⁷ The constraints are boolean formulas over the rules in the hypothesis space, an approach that requires a set of pre-computed rules. This approach can be very expensive to compute because in the worst-case ILASP3 may need to consider every hypothesis to build a constraint (although this worst-case scenario is unlikely). Another way of viewing ILASP3 is that it uses a counter-example guided (Solar-Lezama et al. 2008) approach and translates an uncovered example e into a constraint that is satisfied if and only if e is covered. By contrast, when a hypothesis fails, Popper translates the hypothesis itself into a set of *hypothesis constraints*. Popper’s constraints do not reason about specific clauses (because we do not pre-compute the hypothesis space), but instead reason about the syntax of hypotheses using theta-subsumption and are therefore quick to compute. Another subtle difference is how often the constrain loop is employed in ILASP3 and Popper. ILASP3’s constraint loop requires at most $|E|$ iterations, where $|E|$ is the number of ILASP examples, which are partial interpretations. Because ILASP3’s examples are partial interpretations (Law et al. 2014), it is possible to represent multiple atomic examples in a single partial interpretation example. In fact, each learning task in this paper can be represented as a single ILASP positive example (Law et al. 2014). If represented this way, ILASP3 will generate at most one constraint (which will be satisfied if and only

⁶ A notable exception is Alpha Solver (Weinzierl 2017).

⁷ This statement covers the noiseless ILASP3 setting. Things are slightly more complicated in noisy tasks where examples are given penalties and ILASP3 may return a hypothesis that does not cover all examples, but is optimal with respect to the penalties. Since Popper does not yet support noise, we only consider the noiseless ILASP3 setting.

if a hypothesis covers the example). For this reason, ILASP3 performs much better if the examples are split into one (partial interpretation) example per atomic example. By contrast, the constraint loop of Popper is not bound by the number of examples but by the size of the hypothesis space.

2.7 Hypothesis constraints

Constraints are fundamental to our idea. Many ILP systems allow a user to constrain the hypothesis space through clause constraints (Muggleton 1995; Srinivasan 2001; Blockeel and De Raedt 1998; Ahlgren and Yuen 2013; Law et al. 2014). For instance, Progol, Aleph, and TILDE allow for a user to provide constraints on clauses that should not be violated. Popper also allows a user to provide clause constraints. Popper additionally allows a user to provide *hypothesis constraints* (or *meta-constraints*),⁸ which are constraints over a whole hypothesis (a set of clauses), not an individual clause. As a trivial example, suppose you want to disallow two predicate symbols $p/2$ and $q/2$ from both simultaneously appearing in a program (in any body literal in any clause). Then, because Popper reasons at the meta-level, this restriction is trivial to express:

```
:- body_literal(_, p, 2, _), body_literal(_, q, 2, _).
```

This constraint prunes hypotheses where the predicate symbols $p/2$ and $q/2$ both appear in the body of a hypothesis (possibly in different clauses). The key thing to notice is the ease, uniformity, and succinctness of expressing constraints. We introduce our full meta-level encoding in Sect. 4.

Declarative hypothesis constraints have many advantages. For instance, through hypothesis constraints, Popper can enforce (optional) type, metarule, recall, and functionality restrictions. Moreover, hypothesis constraints allow us to prune recursive programs without a base case and subsumption redundant programs. Finally, and most importantly, hypothesis constraints allow us to prune generalisations and specialisations of failed hypotheses, which we discuss in the next section.

Athakravi et al. (2014) introduce *domain-dependent constraints*, which are constraints on the hypothesis space provided as input by a user. INSPIRE (Schüller and Benz 2018) also uses predefined constraints to remove redundancy from the hypothesis space (in INSPIRE's case, each hypothesis is a single clause). Popper also supports such constraints but goes further by learning constraints from failed hypotheses.

3 Problem setting

We now define our problem setting.

3.1 Logic preliminaries

We assume familiarity with logic programming notation (Lloyd 2012) but we restate some key terminology. All sets are finite unless otherwise stated. A *clause* is a set of literals. A *clausal theory* is a set of clauses. A *Horn* clause is a clause with at most one positive literal.

⁸ The term *hypothesis constraint* is also used in existing work (Srinivasan and Kothari 2005; Costa et al. 2003) as an optional set of constraints on acceptable hypotheses, but without any further explanation.

A *Horn theory* is a set of Horn clauses. A *definite* clause is a Horn clause with exactly one positive literal. A *definite theory* is a set of definite clauses. A Horn clause is a *Datalog* clause if it contains no function symbols and every variable that appears in the head of the clause also appears in the body of the clause. A *Datalog theory* is a set of Datalog clauses. Simultaneously replacing variables v_1, \dots, v_n in a clause with terms t_1, \dots, t_n is a *substitution* and is denoted as $\theta = \{v_1/t_1, \dots, v_n/t_n\}$. A substitution θ unifies atoms A and B when $A\theta = B\theta$. We will often use *program* as a synonym for *theory*, e.g. a *definite program* as a synonym for a *definite theory*.

3.2 Problem setting

Our problem setting is based on the ILP learning from entailment setting (De Raedt 2008). Our goal is to take as input positive and negative examples of a target predicate, background knowledge (BK), and to return a hypothesis (a logic program) that with the BK entails all the positive and none of the negative examples. In this paper, we focus on learning definite programs. We will generalise the approach to non-monotonic programs in future work.

ILP approaches search a *hypothesis space*, the set of learnable hypotheses. ILP approaches restrict the hypothesis space through a language bias (Sect. 2.5). Several forms of language bias exist, such as mode declarations (Muggleton 1995), grammars (Cohen 1994) and metarules (Cropper and Tourret 2020). We use a simple language bias which we call *predicate declarations*. A predicate declaration simply states which predicate symbols may appear in the head (*head declarations*) or body (*body declarations*) of a clause in a hypothesis:

Definition 1 (*Head declaration*) A *head* declaration is a ground atom of the form $head_pred(p,a)$ where p is a predicate symbol of arity a .

Definition 2 (*Body declaration*) A *body* declaration is a ground atom of the form $body_pred(p,a)$ where p is a predicate symbol of arity a .

Predicate declarations are almost identical to Aleph’s *determinations* (Srinivasan 2001) but with a minor syntactical difference because determinations are of the form:

$$determination(TargetName/Arity, BackgroundName/Arity).$$

A *declaration bias* D is a pair (D_h, D_b) of sets of head (D_h) and body (D_b) declarations. We define a *declaration consistent* clause:

Definition 3 (*Declaration consistent clause*) Let $D = (D_h, D_b)$ be a declaration bias and $C = h \leftarrow b_1, b_2, \dots, b_n$ be a definite clause. Then C is *declaration consistent* with D if and only if:

- h is an atom of the form $p(X_1, \dots, X_n)$ and $head_pred(p,n)$ is in D_h
- every b_i is a literal of the form $p(X_1, \dots, X_n)$ and $body_pred(p, n)$ is in D_b
- every X_i is a first-order variable

Example 2 (*Declaration consistency*) Let D be the declaration bias:

$$(\{head_pred(targ,2)\}, \{body_pred(head,2), body_pred(tail,2)\})$$

Then the following clauses are all consistent with D :

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

By contrast, the following clauses are inconsistent with D :

$$\begin{aligned} \text{targ}(A):- \text{head}(A, C). \\ \text{targ}(A, B):- \text{targ}(A, B). \\ \text{tail}(A, B):- \text{reverse}(A, C), \text{tail}(C, B). \end{aligned}$$

We define a *declaration consistent hypothesis*:

Definition 4 (*Declaration consistent hypothesis*) A *declaration consistent hypothesis* H is a set of definite clauses where each $C \in H$ is declaration consistent with D .

Example 3 (*Declaration consistent hypothesis*) Let D be the declaration bias:

$$(\{\text{head_pred}(\text{targ}, 2)\}, \{\text{body_pred}(\text{head}, 2), \text{body_pred}(\text{tail}, 2)\})$$

Then two declaration consistent hypotheses are:

$$\begin{aligned} h_1 : \{ \text{targ}(A, B):- \text{head}(A, B) \} \\ h_2 : \left\{ \begin{array}{l} \text{targ}(A, B):- \text{head}(A, B). \\ \text{targ}(A, B):- \text{tail}(A, C), \text{head}(C, B). \end{array} \right\} \end{aligned}$$

In addition to a declaration bias, we restrict the hypothesis space through *hypothesis constraints*. We first clarify what we mean by a *constraint*:

Definition 5 (*Constraint*) A *constraint* is a Horn clause without a head, i.e. a *denial*. We say that a constraint is *violated* if all of its body literals are true.

Rather than define hypothesis constraints for a specific encoding (e.g. the encoding we use in Sect. 4), we use a more general definition:

Definition 6 (*Hypothesis constraint*) Let \mathcal{L} be a language that defines hypotheses, i.e. a meta-language. Then a hypothesis constraint is a constraint expressed in \mathcal{L} .

Example 4 In Sect. 4, we introduce a meta-language for definite programs. In our encoding, the atom `head_literal(Clause, Pred, Arity, Vars)` denotes that the clause `Clause` has a head literal with the predicate symbol `Pred`, is of arity `Arity`, and has the arguments `Vars`. An example hypothesis constraint in this language is:

$$:- \text{head_literal}(_, p, 2, _).$$

This constraint states that a predicate symbol `p` of arity 2 cannot appear in the head of any clause in a hypothesis.

Example 5 In our encoding, the atom `body_literal(Clause, Pred, Arity, Vars)` denotes that the clause `Clause` has a body literal with the predicate symbol `Pred`, is of arity `Arity`, and has the arguments `Vars`. An example hypothesis constraint in this language is:

$$:- \text{head_literal}(_, p, 2, _), \text{body_literal}(_, p, 2, _).$$

This constraint states that the predicate symbol `p` cannot appear in the body of a clause if it appears in the head of a clause (not necessarily the same clause).

We define a *constraint consistent hypothesis*:

Definition 7 (*Constraint consistent hypothesis*) Let C be a set of hypothesis constraints written in a language \mathcal{L} . A set of definite clauses H is *consistent* with C if, when written in \mathcal{L} , H does not violate any constraint in C .

We now define our hypothesis space:

Definition 8 (*Hypothesis space*) Let D be a declaration bias and C be a set of hypothesis constraints. Then the hypothesis space $\mathcal{H}_{D,C}$ is the set of all declaration and constraint consistent hypotheses. We refer to any element in $\mathcal{H}_{D,C}$ as a *hypothesis*.

We define the LFF problem input:

Definition 9 (*LFF problem input*) Our problem input is a tuple (B, D, C, E^+, E^-) where

- B is a Horn program denoting background knowledge
- D is a declaration bias
- C is a set of hypothesis constraints
- E^+ is a set of ground atoms denoting positive examples
- E^- is a set of ground atoms denoting negative examples

Note that C , E^+ , and E^- can be empty sets (but E^+ and E^- cannot both be empty). We assume that no predicate symbol in the body of a clause in B appears in a head declaration of D . In other words, we assume that the BK does not depend on any hypothesis.

For convenience, we define different types of hypotheses, mostly using standard ILP terminology (Nienhuys-Cheng and de Wolf 1997):

Definition 10 (*Hypothesis types*) Let (B, D, C, E^+, E^-) be an input tuple and $H \in \mathcal{H}_{D,C}$ be a hypothesis. Then H is:

- *Complete* when $\forall e \in E^+ H \cup B \models e$
- *Consistent* when $\forall e \in E^-, H \cup B \not\models e$
- *Incomplete* when $\exists e \in E^+, H \cup B \not\models e$
- *Inconsistent* when $\exists e \in E^-, H \cup B \models e$
- *Totally incomplete* when $\forall e \in E^+, H \cup B \not\models e$

We define a LFF *solution*, i.e. our problem output:

Definition 11 (*LFF solution*) Given an input tuple (B, D, C, E^+, E^-) , a hypothesis $H \in \mathcal{H}_{D,C}$ is a *solution* when H is complete and consistent.

Conversely, we define a *failed hypothesis*:

Definition 12 (*Failed hypothesis*) Given an input tuple (B, D, C, E^+, E^-) , a hypothesis $H \in \mathcal{H}_{D,C}$ *fails* (or is a *failed hypothesis*) when H is either incomplete or inconsistent.

There may be multiple (sometimes infinite) solutions. We want to find the smallest solution:

Definition 13 (*Hypothesis size*) The function $size(H)$ returns the total number of literals in the hypothesis H .

We define an *optimal solution*:

Definition 14 (*Optimal solution*) Given an input tuple (B, D, C, E^+, E^-) , a hypothesis $H \in \mathcal{H}_{D,C}$ is an *optimal solution* when two conditions hold:

- H is a solution
- $\forall H' \in \mathcal{H}_{D,C}$, such that H' is a solution, $size(H) \leq size(H')$.

3.3 Hypothesis space

The purpose of LFF is to reduce the size of the hypothesis space through learned hypothesis constraints. The size of the unconstrained hypothesis space is a function of a declaration bias and additional bounding variables:

Proposition 1 (Hypothesis space size) *Let $D = (D_h, D_b)$ be a declaration bias with a maximum arity a , v be the maximum number of unique variables allowed in a clause, m be the maximum number of body literals allowed in a clause, and n be the maximum number of clauses allowed in a hypothesis. Then the maximum number of hypotheses in the unconstrained hypothesis space is:*

$$\sum_{j=1}^n \left(|D_h|v^a \sum_{i=1}^m \binom{|D_b|v^a}{i} \right)$$

Proof Let C be an arbitrary clause in the hypothesis space. There are $|D_h|v^a$ ways to define the head literal of C . There are $|D_b|v^a$ ways to define a body literal in C . The body of C is a set of literals. There are $\binom{|D_b|v^a}{k}$ ways to choose k body literals. We bound the number of body literals to m , so there are $\sum_{i=1}^m \binom{|D_b|v^a}{i}$ ways to choose at most m body literals. Therefore, there are $|D_h|v^a \sum_{i=1}^m \binom{|D_b|v^a}{i}$ ways to define C . A hypothesis is a set of definite clauses. Given n clauses, there are $\binom{n}{k}$ ways to choose k clauses to form a hypothesis. Therefore, there are $\sum_{j=1}^n \left(|D_h|v^a \sum_{i=1}^m \binom{|D_b|v^a}{i} \right)$ ways to define a hypothesis with at most n clauses. \square

As this result shows, the hypothesis space is huge for non-trivial inputs, which motivates using learned constraints to prune the hypothesis space.

3.4 Generalisations and specialisations

To prune the hypothesis space, we learn constraints to remove *generalisations* and *specialisations* of failed hypotheses. We reason about the generality of hypotheses syntactically through θ -subsumption (or *subsumption* for short) (Plotkin 1971):

Definition 15 (*Clausal subsumption*) A clause C_1 *subsumes* a clause C_2 if and only if there exists a substitution θ such that $C_1\theta \subseteq C_2$.

Example 6 (*Clausal subsumption*) Let C_1 and C_2 be the clauses:

$$\begin{aligned} C_1 &= f(A, B):- \text{head}(A, B) \\ C_2 &= f(X, Y):- \text{head}(X, Y), \text{odd}(Y). \end{aligned}$$

Then C_1 subsumes C_2 because $C_1\theta \subseteq C_2$ with $\theta = \{A/X, Y/B\}$.

If a clause C_1 subsumes a clause C_2 then C_1 entails C_2 (Nienhuys-Cheng and de Wolf 1997). However, if C_1 entails C_2 then it does not necessarily follow that C_1 subsumes C_2 . Subsumption is therefore weaker than entailment. However, whereas checking entailment between clauses is undecidable (Church 1936), checking subsumption between clauses is decidable, although, in general, deciding subsumption is a NP-complete problem (Nienhuys-Cheng and de Wolf 1997).

Midelfart (1999) extends subsumption to clausal theories:

Definition 16 (*Theory subsumption*) A clausal theory T_1 subsumes a clausal theory T_2 , denoted $T_1 \preceq T_2$, if and only if $\forall C_2 \in T_2, \exists C_1 \in T_1$ such that C_1 subsumes C_2 .

Example 7 (*Theory subsumption*) Let h_1, h_2 , and h_3 be the clausal theories:

$$\begin{aligned} h_1 &= \{ f(A, B):- \text{head}(A, B). \} \\ h_2 &= \{ f(A, B):- \text{head}(A, B), \text{odd}(B). \} \\ h_3 &= \left. \begin{aligned} &\{ f(A, B):- \text{head}(A, B). \\ &f(A, B):- \text{reverse}(A, C), \text{head}(C, B). \} \end{aligned} \right\} \end{aligned}$$

Then $h_1 \preceq h_2, h_3 \preceq h_1$, and $h_3 \preceq h_2$.

Theory subsumption also implies entailment:

Proposition 2 (Subsumption implies entailment) *Let T_1 and T_2 be clausal theories. If $T_1 \preceq T_2$ then $T_1 \models T_2$.*

Proof Follows trivially from the definitions of clausal subsumption (Definition 15) and theory subsumption (Definition 16). \square

We use theory subsumption to define a *generalisation*:

Definition 17 (*Generalisation*) A clausal theory T_1 is a *generalisation* of a clausal theory T_2 if and only if $T_1 \preceq T_2$.

We likewise define our notion of a *specialisation*.

Definition 18 (*Specialisation*) A clausal theory T_1 is a *specialisation* of a clausal theory T_2 if and only if $T_2 \preceq T_1$.

In the next section, we use these definitions to define constraints to prune the hypothesis space.

3.5 Learning constraints from failures

In the test stage of LFF, a learner tests a hypothesis against the examples. A hypothesis fails when it is incomplete or inconsistent. If a hypothesis fails, a learner learns hypothesis constraints from the different types of *failures*. We define two general types of constraints, *generalisation* and *specialisation*, which apply to any clausal theory, and show that they are sound in that they do not prune solutions. We also define an *elimination* constraint, which, under certain assumptions, allows us to prune programs that generalisation and specialisation constraints do not, and which we show is sound in that it does not prune optimal solutions. We describe these constraints in turn.

3.5.1 Generalisations and specialisations

To illustrate generalisations and specialisations, suppose we have positive examples E^+ , negative examples E^- , background knowledge B , and a hypothesis H . First consider the outcomes of testing H against E^- :

Outcome	Description	Formula
N_{none}	H is consistent, i.e. H entails no negative example	$\forall e \in E^-, H \cup B \not\models e$
N_{some}	H is inconsistent, i.e. H entails at least one negative example	$\exists e \in E^-, H \cup B \models e$

Suppose the outcome is N_{none} , i.e. H is consistent. Then we cannot prune the hypothesis space.

Suppose the outcome is N_{some} , i.e. H is inconsistent. Then H is too general so we can prune generalisations (Definition 17) of H . A constraint that only prunes generalisations is a *generalisation constraint*.

Definition 19 (*Generalisation constraint*) A generalisation constraint only prunes generalisations of a hypothesis from the hypothesis space.

Example 8 (*Generalisation constraint*) Suppose we have the negative examples E^- and the hypothesis h :

$$E^- = \{ \text{last}([a, n, n], a) \} \quad h = \{ \text{last}(A, B) :- \text{head}(A, B). \}$$

Because h entails a negative example, it is too general, so we can prune generalisations of it, such as h_1 and h_2 :

$$h_1 = \left\{ \begin{array}{l} \text{last}(A, B) :- \text{head}(A, B). \\ \text{last}(A, B) :- \text{tail}(A, C), \text{head}(C, B). \end{array} \right\}$$

$$h_2 = \left\{ \begin{array}{l} \text{last}(A, B) :- \text{head}(A, B). \\ \text{last}(A, B) :- \text{tail}(A, C), \text{head}(C, B), \text{head}(A, B). \end{array} \right\}$$

We show that pruning generalisations of an inconsistent hypothesis is *sound* in that it only prunes inconsistent hypotheses, i.e. does not prune consistent hypotheses:

Proposition 3 (*Generalisation soundness*) Let (B, D, C, E^+, E^-) be a problem input, $H \in \mathcal{H}_{D,C}$ be an inconsistent hypothesis, and $H' \in \mathcal{H}_{D,C}$ be a hypothesis such that $H' \preceq H$. Then H' is inconsistent.

Proof Follows from Proposition 2. □

Now consider the outcomes⁹ of testing H against E^+ :

Outcome	Description	Formula
P_{all}	H is complete, i.e. H entails all positive examples	$\forall e \in E^+, H \cup B \models e$
P_{some}	H is incomplete, i.e. H does not entail all positive examples	$\exists e \in E^+, H \cup B \not\models e$
P_{none}	H is totally incomplete, i.e. H entails no positive examples	$\forall e \in E^+, H \cup B \not\models e$

Suppose the outcome is P_{all} , i.e. H is complete. Then we cannot prune the hypothesis space.

Suppose the outcome is P_{some} , i.e. is incomplete. Then H is too specific so we can prune specialisations (Definition 18) of H . A constraint that only prunes specialisations of a hypothesis is a *specialisation constraint*:

⁹ The outcomes are not mutually exclusive.

Definition 20 (*Specialisation constraint*) A specialisation constraint only prunes specialisations of a hypothesis from the hypothesis space.

Example 9 (*Specialisation constraint*) Suppose we have the positive examples E^+ and the hypothesis h :

$$E^+ = \left\{ \begin{array}{l} \text{last}([b, o, b], b) \\ \text{last}([a, l, i, c, e], e) \end{array} \right\} \quad h = \{ \text{last}(A, B):- \text{head}(A, B). \}$$

Because h entails the first example but not the second it is too specific. We can therefore prune specialisations of h , such as h_1 and h_2 :

$$\begin{aligned} h_1 &= \{ \text{last}(A, B):- \text{head}(A, B), \text{empty}(A). \} \\ h_2 &= \{ \text{last}(A, B):- \text{head}(A, B), \text{tail}(A, C). \} \end{aligned}$$

We show that pruning specialisations of an incomplete hypothesis is *sound* because it only prunes incomplete hypotheses, i.e. does not prune complete hypotheses:

Proposition 4 (*Specialisation soundness*) *Let (B, D, C, E^+, E^-) be a problem input, $H \in \mathcal{H}_{D,C}$ be an incomplete hypothesis, and $H' \in \mathcal{H}_{D,C}$ be a hypothesis such that $H \preceq H'$. Then H' is incomplete.*

Proof Follows from Proposition 2. □

3.5.2 Eliminations

Suppose the outcome is \mathbf{P}_{none} , i.e. H is totally incomplete. Then H is too specific so, as with \mathbf{P}_{some} , we can prune specialisations of H . However, because H is totally incomplete (i.e does not entail *any* positive example), under certain assumptions, we can prune more. If H is totally incomplete then there is no need for H to appear in a complete and *separable* hypothesis:

Definition 21 (*Separable*) A *separable* hypothesis G is one where no predicate symbol in the head of a clause in G occurs in the body of clause in G .

Note that separable programs include recursive programs.

Example 10 (*Non-separable hypothesis*) The following hypothesis is non-separable because $f1/2$ appears in the head and body of the program:

$$\left\{ \begin{array}{l} f(A, B):- f1(A, C), \text{head}(C, B). \\ f1(A, B):- \text{tail}(A, C), \text{tail}(C, B). \end{array} \right\}$$

The following hypothesis is non-separable because $\text{last}/2$ appears in the head and body of the program:

$$\left\{ \begin{array}{l} \text{last}(A, B):- \text{head}(A, B), \text{tail}(A, C), \text{empty}(C). \\ \text{last}(A, B):- \text{tail}(A, C), \text{last}(C, B). \end{array} \right\}$$

In other words, if H is totally incomplete and does not entail *any* positive example, then no specialisation of H can appear in an optimal separable solution. We can therefore prune separable hypotheses that contain specialisations of H . We call such a constraint an *elimination constraint*:

Definition 22 (*Elimination constraint*) An elimination constraint only prunes separable hypotheses that contain specialisations of a hypothesis from the hypothesis space.

Example 11 (Elimination constraint) Suppose we have the positive examples E^+ and the hypothesis h :

$$E^+ = \left\{ \begin{array}{l} \text{last}([b, o, b], b) \\ \text{last}([a, l, i, c, e], e) \end{array} \right\} \quad h = \{ \text{last}(A, B):- \text{tail}(A, C), \text{head}(C, B). \}$$

Because h does not entail any positive example there is no reason for h (nor its specialisations) to appear in a separable hypothesis. We can therefore prune separable hypotheses which contain specialisations of h , such as:

$$\begin{aligned} h_1 &= \left\{ \begin{array}{l} \text{last}(A, B):- \text{head}(A, B). \\ \text{last}(A, B):- \text{tail}(A, C), \text{head}(C, B). \end{array} \right\} \\ h_2 &= \left\{ \begin{array}{l} \text{last}(A, B):- \text{head}(A, B). \\ \text{last}(A, B):- \text{tail}(A, C), \text{head}(C, B), \text{odd}(B). \end{array} \right\} \\ h_3 &= \left\{ \begin{array}{l} \text{last}(A, B):- \text{head}(A, B), \text{even}(B). \\ \text{last}(A, B):- \text{tail}(A, C), \text{head}(C, B), \text{odd}(B). \end{array} \right\} \end{aligned}$$

Elimination constraints are not sound in the same way as the generalisation and specialisation constraints because they prune solutions (Definition 11) from the hypothesis space.

Example 12 (Elimination solution unsoundness) Suppose we have the positive examples E^+ and the hypothesis h_1 :

$$E^+ = \left\{ \begin{array}{l} \text{last}([j, i, m], m) \\ \text{last}([a, l, i, c, e], e) \end{array} \right\} \quad h_1 = \{ \text{last}(A, B):- \text{head}(A, B). \}$$

Then an elimination constraint would prune the complete hypothesis h_2 :

$$h_2 = \left\{ \begin{array}{l} \text{last}(A, B):- \text{head}(A, B). \\ \text{last}(A, B):- \text{reverse}(A, C), \text{head}(C, B). \end{array} \right\}$$

However, for separable definite programs, elimination constraints are *sound* with respect to optimal solutions, i.e. they only prune non-optimal solutions from the hypothesis space. To show this result, we first introduce a lemma:

Lemma 1 *Let (B, D, C, E^+, E^-) be a problem input, $D = (D_h, D_b)$ be head and body declarations, $H_1 \in \mathcal{H}_{D,C}$ be a totally incomplete hypothesis, $H_2 \in \mathcal{H}_{D,C}$ be a complete and separable hypothesis such that $H_1 \subset H_2$, and $H_3 = H_2 \setminus H_1$. Then H_3 is complete.*

Proof By assumption, no predicate symbol in D_h occurs in the body of a clause in B, H_2 (since H_2 is separable), nor H_1 (since $H_1 \subset H_2$), i.e. no clause in a hypothesis depends on another, so we can reason about entailment using single clauses. Since H_1 is totally incomplete, it holds that $\forall e \in E^+, \neg \exists C \in H_1, \{C\} \cup B \models e$. Since H_2 is complete, it holds that $\forall e \in E^+, \exists C \in H_2, \{C\} \cup B \models e$. Therefore, it is clear that $\forall e \in E^+, \exists C \in H_2, C \notin H_1, \{C\} \cup B \models e$, which implies $\forall e \in E^+, H_2 \setminus H_1 \cup B \models e$, and thus H_3 is complete. \square

We use this result to show that elimination constraints are *sound* with respect to optimal solutions:

Proposition 5 (Elimination optimal soundness) *Let (B, D, C, E^+, E^-) be a problem input, $D = (D_h, D_b)$ be head and body declarations, $H_1 \in \mathcal{H}_{D,C}$ be a totally incomplete hypothesis, $H_2 \in \mathcal{H}_{D,C}$ be a hypothesis such that $H_1 \preceq H_2$, and $H_3 \in \mathcal{H}_{D,C}$ be a separable hypothesis such that $H_2 \subset H_3$. Then H_3 is not an optimal solution.*

Table 2 The constraints we can learn from testing a hypothesis

Outcome	N_{none}	N_{some}
P_{all}	n/a	Generalisation
P_{some}	Specialisation	Specialisation, Generalisation
P_{none}	Specialisation, elimination	Specialisation, elimination, generalisation

The **P_{all}** and **N_{none}** outcomes denote that we have found a solution

Proof Assume that H_3 is an optimal solution. This assumption implies that (i) H_3 is a solution, and (ii) there is no hypothesis $H_4 \in \mathcal{H}_{D,C}$ such that H_4 is a solution and $size(H_4) < size(H_3)$. Let $H_4 = H_3 \setminus H_2$. Since H_1 is totally incomplete and $H_1 \preceq H_2$ then, by Proposition 2, H_2 is totally incomplete. By assumption, H_3 is complete and since $H_4 = H_3 \setminus H_2$ and H_2 is totally incomplete then, by Lemma 1, H_4 is complete. Because H_3 is consistent, then, by the monotonicity of definite programs, H_4 is consistent (i.e removing clauses can only make a definite program more specific). Therefore, H_4 is complete and consistent and is a solution. Since $H_4 = H_3 \setminus H_2$ and $H_2 \subset H_3$, then $size(H_4) < size(H_3)$. Therefore, condition (ii) cannot hold, which contradicts the assumption and completes the proof. \square

This proof relies on a hypothesis H being (i) a definite program and (ii) separable. Condition (i) is clear because the proof relies on the monotonicity of definite programs. To illustrate condition (ii), we give a counter-example to show why we cannot use elimination constraints to prune non-separable hypotheses:

Example 13 (*Non-elimination for non-separable hypotheses*) Suppose we have the positive examples E^+ and the hypothesis h :

$$E^+ = \left\{ \begin{array}{l} \text{last}([a, l, a, n], n) \\ \text{last}([t, u, r, i, n, g], g) \end{array} \right\}$$

$$h = \{ \text{last}(A, B):- \text{head}(A, B), \text{tail}(A, C), \text{empty}(C). \}$$

Then h is totally incomplete so there is no reason for h to appear in a separable hypothesis. However, h can still appear in a recursive hypothesis, where the clauses depend on each other, such as h_2 :

$$h_2 = \left\{ \begin{array}{l} \text{last}(A, B):- \text{head}(A, B), \text{tail}(A, C), \text{empty}(C). \\ \text{last}(A, B):- \text{tail}(A, C), \text{last}(C, B). \end{array} \right\}$$

3.5.3 Constraints summary

To summarise, combinations of these different outcomes imply different combinations of constraints, shown in Table 2. In the next section we introduce Popper, which uses these constraints to learn definite programs.

4 Popper

Popper implements the LFF approach and works in three separate stages: generate, test, and constrain. Algorithm 1 sketches the Popper algorithm which combines the three stages. To

learn optimal solutions (Definition 14), Popper searches for programs of increasing size. We describe the generate, test, and constrain stages in detail, how we use ASP's multi-shot solving (Gebser et al. 2019) to maintain state between the three stages, and then prove the soundness and completeness of Popper.

Algorithm 1 Popper

```

1  def popper(e+, e-, bk, declarations, constraints, max_vars,
2     max_literals, max_clauses): num_literals = 1
3     while num_literals ≤ max_literals:
4         program = generate(declarations, constraints, max_vars,
5             num_literals, max_clauses)
6         if program == 'space_exhausted':
7             num_literals += 1
8             continue
9         outcome = test(e+, e-, bk, program)
10        if outcome == ('all_positive', 'none_negative')
11            return program
12        constraints += learn_constraints(program, outcome)
13    return {}

```

The generate step of Popper takes as input (i) predicate declarations, (ii) hypothesis constraints, and (iii) bounds on the maximum number of variables, literals, and clauses in a hypothesis, and returns an answer set which represents a definite program, if one exists. The idea is to define an ASP problem where an answer set (a model) corresponds to a definite program, an approach also employed by other recent ILP approaches (Corapi et al. 2011; Law et al. 2014; Kaminski et al. 2018; Schüller and Benz 2018). In other words, we define a meta-language in ASP to represent definite programs. Popper uses ASP constraints to ensure that a definite program is declaration consistent and obeys hypothesis constraints, such as enforcing type restrictions or disallowing mutual recursion. By later adding learned hypothesis constraints, we eliminate answer sets, and thus reduce the hypothesis space. In other words, the more constraints we learn, the more we reduce the hypothesis space.

Figure 2 shows the base ASP program to generate programs. The idea is to find an answer set with suitable head and body literals, which both have the arguments (Clause, Pred, Arity, Vars) to denote that there is a literal in the clause Clause, with the predicate symbol Pred, arity Arity, and variables Vars. For instance, `head_literal(0, p, 2, (0, 1))` denotes that clause 0 has a head literal with the predicate symbol `p`, arity 2, and variables (0, 1), which we interpret as (A, B). Likewise, `body_literal(1, q, 3, (0, 0, 2))` denotes that clause 1 has a body literal with the predicate symbol `q`, arity 3, and variables (0, 0, 2), which we interpret as (A, A, C). Head and body literals are restricted by `head_pred` and `body_pred` declarations respectively. Table 3 shows examples of the correspondence between an answer set and a definite program, which we represent as a Prolog program.

4.1 Validity, redundancy, and efficiency constraints

Popper uses hypothesis constraints (in the form of ASP constraints) to eliminate answer sets, i.e. to prune the hypothesis space. Popper uses constraints to prune invalid programs. For instance, Fig. 3 shows constraints specifically for recursive programs, such as preventing

```

% possible clauses
allowed_clause(0..N-1):- max_clauses(N).

% variables
var(0..N-1):- max_vars(N).

% clauses with a head literal
clause(Clause):- head_literal(Clause,_,_,_).

%% head literals
0 {head_literal(Clause,P,A,Vars): head_pred(P,A), vars(A,Vars)} 1:-
  allowed_clause(Clause).

%% body literals
1 {body_literal(Clause,P,A,Vars): body_pred(P,A), vars(A,Vars)} N:-
  clause(Clause), max_body(N).

% variable combinations
vars(1,(Var1,))- var(Var1).
vars(2,(Var1,Var2))- var(Var1),var(Var2).
vars(3,(Var1,Var2,Var3))- var(Var1),var(Var2),var(Var3).
    
```

Fig. 2 Popper base ASP program. The `head_literal` literals are bounded from 0 to 1, i.e for each possible clause there can be at most 1 head literal. The `body_literal` literals are bounded from 1 to N , where N is the maximum number of literals allowed in a clause, i.e. for each clause with a head literal, there has to be at least 1 but at most N body literals

Table 3 The correspondence between an answer set and a definite program represented as a Prolog program

Answer set	Prolog program
{ <code>head_literal(0,f,2,(0,1)),body_literal(0,empty,(1,))</code> }	<code>f(A, B):- empty(B).</code>
{ <code>head_literal(0,f,2,(0,1)),body_literal(0,head,2,(1,0))</code> }	<code>f(A, B):- head(B, A).</code>
{ <code>head_literal(0,f,2,(0,1)),body_literal(0,tail,2,(0,1)), body_literal(0,tail,2,(0,2))</code> }	<code>f(A, B):- tail(A, B), tail(A, C).</code>
{ <code>head_literal(0,connected,2,(0,1)),body_literal(0,edge,2,(0,1)), head_literal(1,connected,2,(0,1)),body_literal(1,edge,2,(0,2)), body_literal(1,connected,(2,1))</code> }	<code>connected(A, B):- edge(A, B). connected(A, B):- edge(A, C), connected(C, B).</code>
{ <code>head_literal(0,last,2,(0,1)),body_literal(0,tail,2,(0,2)), body_literal(0,empty,1,(2,)),body_literal(0,head,2,(0,1)), head_literal(1,last,2,(0,1)),body_literal(1,tail,2,(0,2)), body_literal(1,last,2,(2,1))</code> }	<code>last(A, B):- tail(A, C), empty(C), head(A, B). last(A, B):- tail(A, C), last(C, B).</code>

recursion without a base case. Popper also uses constraints to reduce redundancy. For instance, Popper prunes subsumption redundant programs, such as pruning the following program because the first clause subsumes the second:

$$h = \left\{ \begin{array}{l} p(A):- q(A). \\ p(A):- q(A), r(A). \end{array} \right\}$$

Finally, Popper uses constraints to improve efficiency (mostly by removing redundancy). For instance, Popper uses constraints to use variables in order, which prunes the program `p(B):- q(B)` because we could generate `p(A):- q(A)`.

```

recursive:- recursive(Clause).

recursive(Clause):- head_literal(Clause,P,A,_), body_literal(Clause,P,A,_).

has_base:- clause(Clause), not recursive(Clause).

% need multiple clauses for recursion
:- recursive(_), not clause(1).

% prevent recursion without a basecase
:- recursive, not has_base.

```

Fig. 3 Constraints used by Popper to prune invalid recursive programs

```

head_var(Clause,Var):- head_literal(Clause,_,_,Vars), var_member(Var,Vars).

body_var(Clause,Var):- body_literal(Clause,_,_,Vars), var_member(Var,Vars).

% prevent singleton variables
:- clause_var(Clause,Var), #count{P,Vars: var_in_literal(Clause,P,Vars,Var)} == 1.

% head vars must appear in the body
:- head_var(Clause,Var), not body_var(Clause,Var).

% type matching
var_type(Clause,Var,Type):-
  var_in_literal(Clause,P,Vars,Var),
  var_pos(Var,Vars,Pos),
  type(P,Pos,Type).

:- clause_var(Clause,Var), #count{Type : var_type(Clause,Var,Type)} > 1.

```

Fig. 4 Optional language bias constraints used by Popper

4.2 Language bias constraints

Popper supports optional hypothesis constraints to prune the hypothesis space. Figure 4 shows example language bias constraints, such as to prevent singleton variables and to enforce Datalog restrictions (where head variables must appear in the body). Declarative constraints have many benefits, notably the ease to define them. For instance, to add simple types to Popper requires the single constraint shown in Fig. 4. Through constraints, Popper also supports the standard notions of *recall* and *input/output*¹⁰ arguments of mode declarations (Muggleton 1995). Popper also supports *functional* and *irreflexive* constraints, and constraints on recursive programs, such as disallowing left recursion or mutual recursion. Finally, as we show in “Appendix A”, Popper can also use constraints to impose *metarules*, clause templates used by many ILP systems (Cropper and Tourret 2020), which ensures that each clause in a program is an instance of a metarule.

4.3 Hypothesis constraints

As with many ILP systems (Muggleton 1995; Srinivasan 2001; Athakravi et al. 2014; Law et al. 2014; Schüller and Benz 2018), Popper supports *clause* constraints, which allow a user to prune specific clauses from the hypothesis space. Popper additionally supports the more general concept of *hypothesis constraints* (Definition 6), which are defined over a

¹⁰ An input argument specifies that, at the time of calling a predicate, the corresponding argument must be instantiated, which is useful when inducing Prolog programs where literal order matters.

whole program (a set of clauses) rather than a single clause (also employed in previous work (Athakravi et al. 2014)). For instance, hypothesis constraints allow us to prune recursive programs that do not contain a base case clause (Fig. 3), to prune left recursive or mutually recursive programs, or to prune programs which contain subsumption redundancy between clauses.

As a toy example, suppose you want to disallow two predicate symbols $p/2$ and $q/2$ from both appearing in a program. Then this hypothesis constraint is trivial to express with Popper:

```
:- body_literal(_,p,2,_), body_literal(_,q,2,_).
```

As we show in “Appendix A”, Popper can simulate metarules through hypothesis constraints. We are unaware of any other ILP system that supports hypothesis constraints, at least with the same ease and flexibility as Popper.

4.4 Test

In the test stage, Popper converts an answer set to a definite program and tests it against the training examples. As Table 3 shows, this conversion is straightforward, except if input/output argument directions are given, in which case Popper orders the body literals of a clause. To evaluate a hypothesis, we use a Prolog interpreter. For each example, Popper checks whether the example is entailed by the hypothesis and background knowledge. We enforce a timeout to halt non-terminating programs. If a hypothesis fails, then Popper identifies what type of failure has occurred and what constraints to generate (using the failures and constraints from Sect. 3.5).

4.5 Constrain

If a hypothesis fails, then, in the constrain stage, Popper derives ASP constraints which prune hypotheses, thus constraining subsequent hypothesis generation. Specifically, we describe how we transform a failed hypothesis (a definite program) to a hypothesis constraint (an ASP constraint written in the encoding from Sect. 4). We describe the generalisation, specialisation, and elimination constraints that Popper uses, based on the definitions in Sect. 3.5. As our experiments consider a version of Popper without constraint pruning, we also describe the *banish* constraint, which prunes one specific hypothesis. To distinguish between Prolog and ASP code, we represent the code of definite programs in `typewriter` font and ASP code in **bold typewriter** font.

4.5.1 Encoding atoms

In our encoding, the atom $f(A, B)$ is represented as either **head_literal(Clause, f, 2, (V0,V1))** or **body_literal(Clause, f, 2, (V0,V1))**. The constant **2** is the predicate’s arity and the variable **Clause** indicates that the clause index is undetermined. Two functions encode atoms into ASP literals. The function *encodeHead* encodes a head atom and *encodeBody* encodes a body atom. The first argument specifies the clause the atom

belongs to. The second argument is the atom. Variables of the atom are converted to variables in our ASP encoding by the *encodeVar* function.

```

encodeHead(Clause, Pred(Var0, ..., Vark))
  := head_literal(Clause, Pred, k + 1, (encodeVar(Var0), ..., encodeVar(Vark)))

encodeBody(Clause, Pred(Var0, ..., Vark)) :=
  body_literal(Clause, Pred, k + 1, (encodeVar(Var0), ..., encodeVar(Vark)))

```

For instance, using the term **C1** as a clause variable, calling *encodeHead*(**C1**, *f*(A, B)) returns the ASP literal **head_literal**(**C1**, *f*, 2, (**V0**, **V1**)). Similarly, calling *encodeBody*(**C1**, *f*(A, B)) returns **body_literal**(**C1**, *f*, 2, (**V0**, **V1**)).

4.5.2 Encoding clauses

We encode clauses by building on the encoding of atoms. Let **C1** be a clause index variable. Consider the clause *last*(A, B):- *reverse*(A, C), *head*(C, B). The following ASP literals encode where these atoms occur in a single clause:

```

head_literal(C1, last, 2, (V0, V1))
body_literal(C1, reverse, 2, (V0, V2))
body_literal(C1, head, 2, (V2, V1))

```

An ASP solver will instantiate the variables **V0**, **V1**, and **V2** with indices representing variables of hypotheses, e.g. **0** for A, **1** for B, etc. Note that the above encoding allows for **V0** = **V1** = **V2** = **0**, where all the variables are A. To ensure that variables are distinct we need to impose the inequality **V0** ≠ **V1** and **V0** ≠ **V2** and **V1** ≠ **V2**. The function *assertDistinct* generates such inequalities, one between each pair of variables it is given. The function *encodeClause* implements both the straightforward translation and the variable distinctness assertion:

```

encodeClause(Clause, (head:- body1, ..., bodym))
  := encodeHead(Clause, head), encodeBody(Clause, body1), ...,
  encodeBody(Clause, bodym),
  assertDistinct(vars(head) ∪ vars(body1) ∪ ... ∪ vars(bodym))

```

As clauses can occur in multiple hypotheses, it is convenient to refer to clauses by identifiers. The function *clauseIdent* maps clauses to unique ASP constants.¹¹ We use the ASP literal **included_clause**(*cl*, *id*) to represent that a clause with index *cl* includes all literals of a clause identified by *id*. The *inclusionRule* function generates an *inclusion rule*, an ASP rule whose head is true when the literals of the provided clause occur together in a clause:

```

inclusionRule(head:- body1, ..., bodym)
  := included_clause(C1, clauseIdent(head:- body1, ..., bodym)) :-
  encodeClause(C1, (head:- body1, ..., bodym)).

```

¹¹ Even though the examples use increasing numbers in the identifiers, *clauseIdent* can be any injective function, i.e. always mapping a clause to the same unique identifier.

$h = \{ \text{last}(A, B) :- \text{reverse}(A, C), \text{head}(C, B). \}$

```

included_clause(Cl, id1) :-
  head_literal(Cl, last, 2, (V0, V1)),
  body_literal(Cl, reverse, 2, (V0, V2)),
  body_literal(Cl, head, 2, (V2, V1)),
  V0!=V1, V0!=V2, V1!=V2.
:-
  included_clause(Cl0, id1),
  clause_size(Cl0, 2).

```

Fig. 5 ASP encoded inclusion rule and generalisation constraint for the hypothesis h

Suppose that $\text{clauseIdent}(\text{last}(A, B) :- \text{reverse}(A, C), \text{head}(C, B)) = \mathbf{id}_1$. Then the rule obtained by $\text{inclusionRule}(\text{last}(A, B) :- \text{reverse}(A, C), \text{head}(C, B))$ is:

```

included_clause(Cl, id1)
  :- head_literal(Cl, last, 2, (V0, V1)),
     body_literal(Cl, reverse, 2, (V0, V2)),
     body_literal(Cl, head, 2, (V2, V1)),
     V0!=V1, V0!=V2, V1!=V2.

```

Note that $\text{included_clause}(cl, id)$ being true does not mean that other literals do *not* occur in the clause. For example, if a clause with index $\mathbf{0}$ encoded the clause $\text{last}(A, B) :- \text{reverse}(A, C), \text{head}(C, B), \text{tail}(C, A)$, then $\text{included_clause}(\mathbf{0}, \mathbf{id}_1)$ would also hold.

In our encoding, $\text{clause_size}(cl, m)$ is only true when clause cl has exactly m body literals. Hence when literals $\text{included_clause}(\mathbf{0}, \mathbf{id}_1)$ and $\text{clause_size}(\mathbf{0}, \mathbf{2})$ are both true, the clause with index $\mathbf{0}$ exactly encodes $\text{last}(A, B) :- \text{reverse}(A, C), \text{head}(C, B)$. The function exactClause derives a pair of ASP literals checking that a clause occurs exactly:

```

exactClause(Clause, (head:- body1, ..., bodym))
  := included_clause(Clause, clauseIdent(head:- body1, ..., bodym)),
     clause_size(Clause, m)

```

4.5.3 Generalisation constraints

Given a hypothesis H , by Definition 17, any hypothesis that includes all of H 's clauses exactly is a generalisation of H . We use this fact to define function $\text{generalisationConstraint}$, which converts a set of clauses into ASP encoded clause inclusion checking rules as well as a generalisation constraint (Definition 19). We use exactClause to impose that a clause is not specialised. Each clause is given its own ASP variable, meaning that the clauses can occur in any order.

```

generalisationConstraint({Clause0, ..., Clausen-1}) :=
  inclusionRule(Clause0)
  ...
  inclusionRule(Clausen-1)
  :- exactClause(Cl0, Clause0), ..., exactClause(Cln-1, Clausen-1).

```

Figure 5 illustrates $\text{generalisationConstraint}$ deriving both an inclusion rule and a generalisation constraint.

$$h = \left\{ \begin{array}{l} \text{rev}(A,B) :- \text{head}(A,B). \\ \text{rev}(A,B) :- \text{tail}(A,C), \text{head}(C,B). \end{array} \right\}$$

```

included_clause(Cl, id2) :-
  head_literal(Cl, rev, 2, (V0, V1)),
  body_literal(Cl, head, 2, (V0, V1)),
  V0 != V1.
included_clause(Cl, id3) :-
  head_literal(Cl, rev, 2, (V0, V1)),
  body_literal(Cl, tail, 2, (V0, V2)),
  body_literal(Cl, head, 2, (V2, V1)),
  V0 != V1, V0 != V2, V1 != V2.
:-
  included_clause(Cl0, id2),
  included_clause(Cl1, id3),
  Cl0 != Cl1, not clause(2).

```

Fig. 6 ASP encoded inclusion rules and specialisation constraint for the hypothesis h

4.5.4 Specialisation constraints

Given a hypothesis H , by Definition 18, any hypothesis which has every clause of H occur, where each of these clauses may be specialised, and includes no other clauses, is a specialisation of H . The function *specialisationConstraint* uses this fact to derive an ASP encoded specialisation constraint (Definition 20) alongside inclusion rules. When **included_clause**(cl, id) is true, additional atoms can occur in the clause cl . The literal **not clause**(n) ensures that no additional clause is added to the n distinct clauses of the provided hypothesis.

```

specialisationConstraint({Clause0, ..., Clausen-1}) :=
  inclusionRule(Clause0)
  ...
  inclusionRule(Clausen-1)
  :- included_clause(Cl0, clauseIdent(Clause0)), ...,
     included_clause(Cln-1, clauseIdent(Clausen-1)),
     assertDistinct({Cl0, ..., Cln-1}), not clause(n).

```

We illustrate why asserting that specialised clauses are distinct is necessary. Consider the hypotheses h_1 and h_2 :

$$h_1 = \left\{ \begin{array}{l} \text{last}(A, B) :- \text{head}(A, B). \\ \text{last}(A, B) :- \text{sumlist}(A, B). \end{array} \right\} \quad h_2 = \left\{ \begin{array}{l} \text{last}(A, B) :- \text{head}(A, B), \text{sumlist}(A, B). \\ \text{last}(A, B) :- \text{member}(A, B). \end{array} \right\}$$

The first clause of h_2 specialises both clauses in h_1 , yet h_2 is not a specialisation of h_1 . According to Definition 18, each clause needs to be subsumed by a provided clause. Note that *specialisationConstraint* only considers hypotheses with at most n clauses. It is not possible for one of these clauses to be non-specialising, as each of the original n clauses is required to be specialised by a distinct clause.

Figure 6 illustrates a specialisation constraint derived by *specialisationConstraint*.

4.5.5 Elimination constraints

By Proposition 5, given a totally incomplete hypothesis H , any separable hypothesis which includes all of H 's clauses, where each clause may be specialised, cannot be an optimal solution. We add the following code to the Popper encoding to detect separable hypotheses:

$$h = \{ \text{last}(A,B) :- \text{tail}(A,C), \text{head}(C,B). \}$$

```

included_clause(Cl, id4) :-
  head_literal(Cl, last, 2, (V0, V1)),
  body_literal(Cl, tail, 2, (V0, V2)),
  body_literal(Cl, head, 2, (V2, V1)),
  V0!=V1, V0!=V2, V1!=V2.
:-
  included_clause(Cl0, id4),
  separable.

```

Fig. 7 ASP encoded inclusion rule and elimination constraint for the hypothesis h

```

non_separable :-
  head_literal(_, P, A, _),
  body_literal(_, P, A, _).

separable :-
  not non_separable.

```

The function *eliminationConstraint* uses this fact to derive an ASP encoded elimination constraint (Definition 22). As in *specialisationConstraint*, **included_clause**(cl, id) is used to allow additional literals in clauses, ensuring that provided clauses are specialised. However, *eliminationConstraint* does not require that every clause is a specialisation of a provided clause. Instead, all that is required is that the hypothesis is separable.

```

eliminationConstraint({Clause0, ..., Clausen-1}) :=
  inclusionRule(Clause0)
  ...
  inclusionRule(Clausen-1)
  :- included_clause(Cl0, clauseIdent(Clause0)), ...,
     included_clause(Cln-1, clauseIdent(Clausen-1)),
     separable.

```

Figure 7 illustrates an elimination constraint derived by *eliminationConstraint*.

4.5.6 Banish constraints

In the experiments we compare Popper against a version of itself without constraint pruning. To do so we need to remove single hypotheses from the hypothesis space. We introduce the *banish constraint* for this purpose. To prune a specific hypothesis, hypotheses with different variables should not be pruned. We accomplish this condition by changing the behaviour of the *encodeVar* function. Normally *encodeVar* returns ASP variables which are then grounded to indices that correspond to the variables of hypotheses. Instead, by the following definition, *encodeVar* directly assigns the corresponding index for a hypothesis variable:

$$\text{encodeVar} = \{ A \mapsto \mathbf{0}; B \mapsto \mathbf{1}; C \mapsto \mathbf{2}; \dots \}$$

For a banish constraint no additional literals in clauses are allowed, nor are additional clauses. The below function *banishConstraint* ensures both conditions when converting a hypothesis to an ASP encoded banish constraint. That provided clauses occur non-specialised is ensured by *exactClause*. The literal **not clause**(n) asserts that there are no more clauses than the original number.

$$h = \left\{ \begin{array}{l} f(A) :- \text{head}(A,B), \text{one}(B). \\ f(A) :- \text{tail}(A,B), \text{empty}(B). \end{array} \right\}$$

```

included_clause(Cl, id5) :-
  head_literal(Cl, f, 1, (0,)),
  body_literal(Cl, head, 2, (0,1)),
  body_literal(Cl, one, 1, (1,)).
included_clause(Cl, id6) :-
  head_literal(Cl, f, 1, (0,)),
  body_literal(Cl, tail, 2, (0,1)),
  body_literal(Cl, empty, 1, (1,)).
:-
  included_clause(Cl0, id5),
  clause_size(Cl0, 2),
  included_clause(Cl1, id6),
  clause_size(Cl1, 2),
  not clause(2).

```

Fig. 8 ASP encoded inclusion rules and banish constraint for the hypothesis h

```

banishConstraint({Clause0, ..., Clausen-1}) :=
  inclusionRule(Clause0)
  ...
  inclusionRule(Clausen-1)
  :- exactClause(Cl0, Clause0), ..., exactClause(Cln-1, Clausen-1),
     not clause(n).

```

Figure 8 illustrates a banish constraint derived by *banishConstraint*.

4.6 Popper loop and multi-shot solving

A naive implementation of Algorithm 1, such as performing iterative deepening on the program size, would duplicate grounding and solving during the generate step. To improve efficiency, we use Clingo’s multi-shot solving (Gebser et al. 2019) to maintain state between the three stages. The idea of multi-shot solving is that state of the solving process for an ASP program can be saved to help solve modifications of that program. The essence of the multi-shot cycle is that a ground program is given to an ASP solver, yielding an answer set, who’s processing leads to a (first-order) extension of the program. Only this extension then needs grounding and adding to the running ASP instance, which means that the running solver may, for example, maintain learned conflicts.

Popper uses multi-shot solving as follows. The initial ASP program is the encoding described in Sect. 4. Popper asks Clingo to ground the initial program and prepare for its solving. In the generate stage, the solver is asked to return an answer set, i.e. a model, of the current program. Popper converts such an answer set to a definite program and tests it against the examples. If a hypothesis fails, Popper generates ASP constraints using the functions in Sect. 4.5 and adds them to the running Clingo instance, which grounds the constraints and adds the new (propositional) rules to the running solver. We employ a hard constraint on the program size that reasons about an *external atom* (Gebser et al. 2019) *size(N)*. Initially, programs need to consist of just one literal. When there are no more answer sets, we increment the program size. Every time we increment the program size, e.g. from N to $N+1$, we add a new atom *size(N+1)* and a new constraint enforcing this program size. Only the new constraint is ground at this point. We disable the previous constraint by setting the external atom *size(N)* to false. The solver knows which parts of the search space (i.e. hypothesis

space) have already been considered and will not revisit them. This loop repeats until either (i) Popper finds an optimal solution, or (ii) there are no more hypotheses to test.

4.7 Worked example

To illustrate Popper, reconsider the example from the introduction of learning a *last/2* hypothesis to find the last element of a list. For simplicity, assume an initial hypothesis space \mathcal{H}_1 :

$$\mathcal{H}_1 = \left\{ \begin{array}{l} h_1 = \{ \text{last}(A, B):- \text{head}(A, B). \} \\ h_2 = \{ \text{last}(A, B):- \text{head}(A, B), \text{empty}(A). \} \\ h_3 = \{ \text{last}(A, B):- \text{tail}(A, C), \text{head}(C, B). \} \\ h_4 = \{ \text{last}(A, B):- \text{reverse}(A, C), \text{head}(C, B). \} \\ h_5 = \{ \text{last}(A, B):- \text{head}(A, B), \text{reverse}(A, C), \text{head}(C, B). \} \\ h_6 = \left\{ \begin{array}{l} \text{last}(A, B):- \text{tail}(A, C), \text{head}(C, B). \\ \text{last}(A, B):- \text{reverse}(A, C), \text{head}(C, B). \end{array} \right\} \\ h_7 = \left\{ \begin{array}{l} \text{last}(A, B):- \text{tail}(A, C), \text{head}(C, B). \\ \text{last}(A, B):- \text{tail}(A, C), \text{tail}(C, D), \text{head}(D, B). \end{array} \right\} \\ h_8 = \left\{ \begin{array}{l} \text{last}(A, B):- \text{reverse}(A, C), \text{tail}(C, D), \text{head}(D, B). \\ \text{last}(A, B):- \text{tail}(A, C), \text{reverse}(C, D), \text{head}(D, B). \end{array} \right\} \\ h_9 = \left\{ \begin{array}{l} \text{last}(A, B):- \text{head}(A, B). \\ \text{last}(A, B):- \text{reverse}(A, C), \text{tail}(C, D), \text{head}(D, B). \\ \text{last}(A, B):- \text{tail}(A, C), \text{reverse}(C, D), \text{head}(D, B). \end{array} \right\} \end{array} \right.$$

Also assume we have the positive (E^+) and negative (E^-) examples:

$$E^+ = \left\{ \begin{array}{l} \text{last}([1, a, u, r, a], a). \\ \text{last}([p, e, n, e, l, o, p, e], e). \end{array} \right\} \quad E^- = \left\{ \begin{array}{l} \text{last}([e, m, m, a], m). \\ \text{last}([j, a, m, e, s], e). \end{array} \right\}$$

To start, Popper generates the simplest hypothesis:

$$h_1 = \{ \text{last}(A, B):- \text{head}(A, B). \}$$

Popper then tests h_1 against the examples and finds that it *fails* because it does not entail any positive example and is therefore too *specific*. Popper then generates a specialisation constraint to prune specialisations of h_1 :

```

included_clause(C1, id1) :-
    head_literal(C1, last, 2, (V0, V1)),
    body_literal(C1, head, 2, (V0, V1)),
    V0 != V1.
:-
    included_clause(C10, id1),
    not clause(1).
```

Popper adds this constraint to the meta-level ASP program which prunes h_2 and h_5 from the hypothesis space. In addition, because h_1 does not entail any positive example (is *totally* incomplete), Popper also generates an elimination constraint:

```

:-
    included_clause(C10, id1),
    separable.
```

Popper adds this constraint to the meta-level ASP program which prunes h_9 from the hypothesis space. The hypothesis space is now:

$$\mathcal{H}_2 = \left\{ \begin{array}{l} h_3 = \{ \text{last}(A, B):- \text{tail}(A, C), \text{head}(C, B). \} \\ h_4 = \{ \text{last}(A, B):- \text{reverse}(A, C), \text{head}(C, B). \} \\ h_6 = \left\{ \begin{array}{l} \text{last}(A, B):- \text{tail}(A, C), \text{head}(C, B). \\ \text{last}(A, B):- \text{reverse}(A, C), \text{head}(C, B). \end{array} \right\} \\ h_7 = \left\{ \begin{array}{l} \text{last}(A, B):- \text{tail}(A, C), \text{head}(C, B). \\ \text{last}(A, B):- \text{tail}(A, C), \text{tail}(C, D), \text{head}(D, B). \end{array} \right\} \\ h_8 = \left\{ \begin{array}{l} \text{last}(A, B):- \text{reverse}(A, C), \text{tail}(C, D), \text{head}(D, B). \\ \text{last}(A, B):- \text{tail}(A, C), \text{reverse}(C, D), \text{head}(D, B). \end{array} \right\} \end{array} \right\}$$

Popper generates another hypothesis (h_3) and tests against the examples and finds that it fails because it entails the negative example $\text{last}([e, m, m, a], m)$ and is therefore too *general*. Popper then generates a generalisation constraint to prune generalisations of h_3 :

```

included_clause(C1, id2) :-
    head_literal(C1, last, 2, (V0, V1)),
    body_literal(C1, tail, 2, (V0, V2)),
    body_literal(C1, head, 2, (V2, V1)),
    V0 != V1, V0 != V2, V1 != V2.

:-
    included_clause(C10, id2),
    clause_size(C10, 2).
```

Popper adds this constraint to the meta-level ASP program which prunes h_6 and h_7 from the hypothesis space. The hypothesis space is now:

$$\mathcal{H}_3 = \left\{ \begin{array}{l} h_4 = \{ \text{last}(A, B):- \text{reverse}(A, C), \text{head}(C, B). \} \\ h_8 = \left\{ \begin{array}{l} \text{last}(A, B):- \text{reverse}(A, C), \text{tail}(C, D), \text{head}(D, B). \\ \text{last}(A, B):- \text{tail}(A, C), \text{reverse}(C, D), \text{head}(D, B). \end{array} \right\} \end{array} \right\}$$

Finally, Popper generates another hypothesis (h_4), tests it against the examples, finds that it does not fail, and returns it.

4.8 Correctness

We now show the correctness of Popper. However, we can only show this result for when the hypothesis space only contains decidable programs, e.g. Datalog programs. When the hypothesis space contains arbitrary definite programs, then the results do not hold because checking for entailment of an arbitrary definite program is only semi-decidable (Tärnlund 1977). In other words, the results in this section only hold when every hypothesis in the hypothesis space is guaranteed to terminate.¹²

We first show that Popper’s base encoding (Fig. 2) can generate every declaration consistent hypothesis (Definition 4).

Proposition 6 *The base encoding of Popper has a model for every declaration consistent hypothesis.*

¹² In practice, such as in our experiments on learning list transformation programs, we enforce a timeout when testing programs, i.e. we assume that every solution terminates before the timeout.

Proof Let $D = (D_h, D_b)$ be a declaration bias, N_{var} be the maximum number of unique variables, N_{body} be the maximum number of body literals, N_{clause} be the maximum number of clauses, H be any hypothesis declaration consistent with D and these parameters, and C be any clause in H . Our encoding represents the head literal $p_h(H_1, \dots, H_n)$ of C as a choice literal $\text{head_literal}(i, p_h, n, (H_1, \dots, H_n))$ guarded by the condition $\text{head_pred}(p_h, n) \in D_h$, which clearly holds. Our encoding represents a body literal $p_b(B_1, \dots, B_m)$ of C as a choice literal $\text{body_literal}(i, p_b, m, (B_1, \dots, B_m))$ guarded by the condition $\text{body_pred}(p_b, m) \in D_b$, which clearly holds. The base encoding only constrains the above guesses by three conditions: (i) at most N_{var} unique variables per clause, (ii) at least 1 and at most N_{body} body literals per clause, and (iii) at most N_{clause} clauses. As both the hypothesis and the guessed literals satisfy the same conditions, we conclude there exists a model representing H . \square

We show that any hypothesis returned by Popper is a solution (Definition 11).

Proposition 7 (Soundness) *Any hypothesis returned by Popper is a solution.*

Proof Any returned hypothesis has been tested against the training examples and confirmed as a solution. \square

To make the next two results shorter, we introduce a lemma to show that Popper never prunes optimal solutions (Definition 14).

Lemma 2 *Popper never prunes optimal solutions.*

Proof Popper only learns constraints from a failed hypothesis, i.e. a hypothesis that is incomplete or inconsistent. Let H be a failed hypothesis. If H is incomplete, then, as described in Sect. 4.5, Popper prunes specialisations of H . Proposition 4 shows that a specialisation constraint never prunes complete hypotheses, and thus never prunes optimal solutions. If H is inconsistent, then, as described in Sect. 4.5, Popper prunes generalisations of H . Proposition 3 shows that a generalisation constraint never prunes consistent hypotheses, and thus never prunes optimal solutions. Finally, if H is totally incomplete, then, as described in Sect. 4.5, Popper uses an elimination constraint to prune all separable hypotheses that contain H . Proposition 5 shows that an elimination constraint never prunes optimal solutions. Since Popper only uses these three constraints, it never prunes optimal solutions. \square

We now show that Popper returns a solution if one exists.

Proposition 8 (Completeness) *Popper returns a solution if one exists.*

Proof Assume, for contradiction, that Popper does not return a solution, which implies that (1) Popper returned a hypothesis that is not a solution, or (2) Popper did not return a solution. Case (1) cannot hold because Proposition 7 shows that every hypothesis returned by Popper is a solution. For case (2), by Proposition 6, Popper can generate every hypothesis so it must be the case that (i) Popper did not terminate, (ii) a solution did not pass the test stage, or (iii) that every solution was incorrectly pruned. Case (i) cannot hold because Proposition 1 shows that the hypothesis space is finite so there are finitely many hypotheses to generate and test. Case (ii) cannot hold because a solution is by definition a hypothesis that passes the test stage. Case (iii) cannot hold because Lemma 2 shows that Popper never prunes optimal solutions. These cases are exhaustive, so the assumption cannot hold, and thus Popper returns a solution if one exists. \square

We show that Popper returns an optimal solution if one exists:

Theorem 1 (Optimality) *Popper returns an optimal solution if one exists.*

Proof By Proposition 8, Popper returns a solution if one exists. Let H be the solution returned by Popper. Assume, for contradiction, that H is not an optimal solution. By Definition 14, this assumption implies that either (1) H is not a solution, or (2) H is a non-optimal solution. Case (1) cannot hold because H is a solution. Therefore, case (2) must hold, i.e. there must be at least one smaller solution than H . Let H' be an optimal solution, for which we know $size(H') < size(H)$. By Proposition 6, Popper generates every hypothesis, and Popper generates hypotheses of increasing size (Algorithm 1), therefore the smaller solution H' must have been considered before H , which implies that H' must have been pruned by a constraint. However, Lemma 2 shows that H' could not have been pruned and so cannot exist, which contradicts the assumption and completes the proof. \square

5 Experiments

We now evaluate Popper. Popper learns constraints from failed hypotheses to prune the hypothesis space to improve learning performance. We therefore claim that, compared to unconstrained learning, constraints can improve learning performance. One may think that this improvement is obvious, i.e. constraints will definitely improve performance. However, it is unclear whether in practice, and if so by how much, constraints will improve learning performance because Popper needs to (i) analyse failed hypotheses, (ii) generate constraints from them, and (iii) pass the constraints to the ASP system, which then needs to ground and solve them, which may all have non-trivial computational overheads. Our experiments therefore aim to answer the question:

Q1 Can constraints improve learning performance compared to unconstrained learning?

To answer this question, we compare Popper with and without the constrain stage. In other words, we compare Popper against a brute-force generate and test approach. To do so, we use a version of Popper with only banish constraints enabled to prevent repeated generation of a failed hypothesis. We call this system *Enumerate*.

Proposition 1 shows that the size of the learning from failures hypothesis space is a function of many parameters, including the number of predicate declarations, the number of unique variables in a clause, and the number of clauses in a hypothesis. To explore this result, our experiments aim to answer the question.

Q2 How well does Popper scale?

To answer this question, we evaluate Popper when varying (i) the size of the optimal solution, (ii) the number of predicate declarations, (iii) the number of constants in the problem, (iv) the number of unique variables in a clause, (v) the maximum number of literals in a clause, and (vi) the maximum number of clauses allowed in a hypothesis.

We also compare Popper against existing ILP systems. Our experiments therefore aim to answer the question.

Q3 How well does Popper perform compared to other ILP systems?

To answer this question, we compare Popper against Aleph (Srinivasan 2001), Metagol, ILASP2i (Law et al. 2016), and ILASP3 (Law 2018). It is, however, important to note that

a direct comparison of ILP systems is difficult because different systems excel at different problems and often employ different biases. For instance, directly comparing the Prolog-based Metagol against the ASP-based ILASP is difficult because Metagol is often used to learn recursive list manipulation programs, such as string transformations and sorting algorithms, whereas ILASP does not support explicit lists because the ASP system Clingo (Gebser et al. 2014), on which ILASP is built, disallows explicit lists. Likewise, Aleph and ILASP3 support noise, whereas Metagol and Popper do not. Moreover, because ILP systems have many learning parameters, it is often possible to show that there exist some parameter settings for which system X can perform better than system Y on a particular problem. Overall, a direct comparison between ILP systems is difficult, so a reader should not interpret the results as system X is better than system Y.

5.1 Buttons

The purpose of this first experiment is to evaluate how well Popper scales when varying the optimal solution size.¹³ We therefore need a problem where we can control the optimal solution size. We consider a problem loosely based on the IGGP game *buttons and lights* (Cropper et al. 2020). The problem is purposely simple: given p buttons, learn which n buttons need to be pressed to win. For instance, for $n = 3$, a solution could be:

$$\text{win}(A):- \text{button6}(A), \text{button4}(A), \text{button7}(A)$$

The variable A denotes the player and button_p denotes that player A pressed button_p .

In this experiment, we fix p , the number of buttons, and vary n , the number of buttons that need to be pressed, which directly corresponds to the optimal solution size.

5.1.1 Materials

We consider two variations of the problem where $p = 20$ and $p = 200$, which we name *small* and *big* respectively. We compare Popper, Enumerate, Metagol, ILASP2i, ILASP3, and Aleph. To compare the systems, we try to use settings so that each system searches approximately the same hypothesis space. However, ensuring that the systems search identical hypothesis spaces is near impossible. For instance, Metagol performs automatic predicate invention and so considers a different hypothesis space to the other systems. The exact language biases used are in “Appendix B”.

ILASP settings We asked Mark Law, the ILASP author, for advice on how best to solve this problem with ILASP2i and ILASP3.¹⁴ We run both ILASP2i and ILASP3 with the same settings so we simply refer to both as ILASP. We run ILASP with the ‘no constraints’, ‘no aggregates’, ‘disable implication’, ‘disable propagation’, and ‘simple contexts’ flags. We tell ILASP that each BK relation is positive, which prevents it from generating body literals using negation. We also make the problem propositional and use context-dependent examples (Law et al. 2016) where the context-dependent BK for each example contains the buttons pressed in that example. We initially tried to run ILASP with at most ten body literals (‘-ml = 10’

¹³ Note that, in this experiment, increasing the optimal solution size almost always also increases the size of the hypothesis space for the considered ILP systems.

¹⁴ Mark suggested an alternative representation that corresponds to learning the negation of the concept, which would have been much more suitable for ILASP. However, this alternative different representation requires NAF which not all of the other systems support.

and ‘ $\text{-max-rule-length} = 11$ ’) but when given this parameter ILASP would not terminate in the time limit because it pre-computes every rule in the hypothesis space. Therefore, for each number of buttons n , we set the maximum number of body literals to n (‘ $\text{-ml} = n$ ’ and ‘ $\text{-max-rule-length} = n + 1$ ’), to ensure that ILASP terminates on some of the problems.

Metagol settings Metagol needs metarules (Sect. 2.5) to guide the proof search. We provide Metagol with the following two metarules:

$$\begin{aligned} P(A):- Q(A). \\ P(A):- Q(A), R(A). \end{aligned}$$

Popper and Enumerate settings We set Popper and Enumerate to use at most 1 unique variable, at most 1 clause, and at most n body literals. These settings match those imposed by Metagol’s metarules and somewhat ILASP’s propositional representation. We restrict the clause to have at most n body literals to match ILASP’s settings. When allowed up to ten body literals, Popper performs almost identically.

Aleph settings We also set the maximum number of nodes to be search to be 5000. As with Popper, Enumerate, and ILASP, we increase the maximum clause length for Aleph for each value n .

5.1.2 Methods

For each n in $\{1, 2, \dots, 10\}$, we generate 200 positive and 200 negative examples. A positive example is a player that has pressed the correct n buttons. To generate a positive example we sample without replacement n integers from the set $\{1, \dots, p\}$ which correspond to the n buttons that *must* be pressed. We additionally sample extra buttons that are also pressed, but which are not necessarily pressed in all the positive examples. A negative example is a player that has not pressed the correct n buttons. To generate a negative example we sample without replacement at most $n - 1$ buttons from the set that must be pressed. We then sample other buttons that should not be pressed. By including all n negative examples with $n - 1$ correct buttons we guarantee that there is only one correct solution. We measure learning time as the time to learn a solution. We enforce a timeout of one minute per task. We repeat each experiment ten times and plot the standard error.

5.1.3 Results

Figure 9 shows that Popper clearly outperforms Enumerate on both datasets. On the small dataset ($p = 20$), Enumerate only learns a program for when three buttons must be pressed ($n = 3$). On the large dataset ($p = 200$), Enumerate only learns a program for when one button must be pressed ($n = 1$). By contrast, on both datasets, Popper learns a program for when ten buttons must be pressed ($n = 10$), i.e. a program with ten body literals. Moreover, Popper always learns a solution comfortably within the time limit. This result strongly suggests that the answer to **Q1** is yes, constraints can drastically improve learning performance.

Popper outperforms Metagol on both datasets. For the small dataset, the largest program that Metagol learns is for when $n = 4$, which takes 50 seconds to learn, compared to one second for Popper. For the big dataset, the largest program that Metagol learns is for when $n = 3$, which takes 57 seconds to learn, compared to eight seconds for Popper. Metagol struggles because of its inefficient search. Metagol performs iterative deepening over the number of clauses allowed in a solution (Muggleton et al. 2015). However, if a clause or

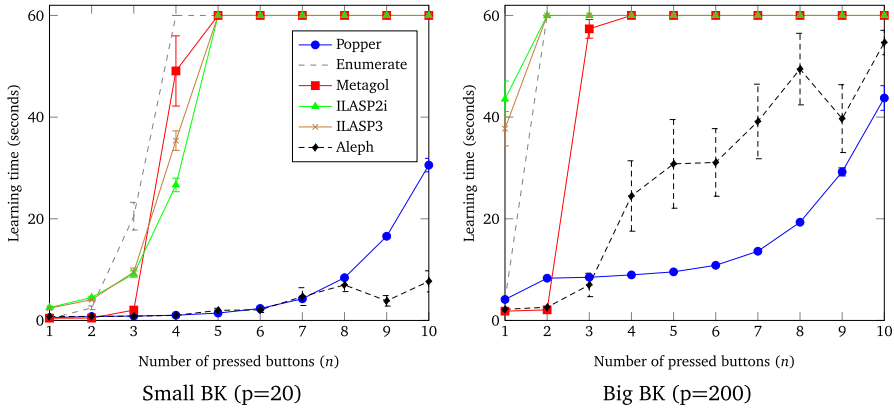


Fig. 9 Buttons experiment

literal fails during the search, Metagol does not remember this failure, and will retry already failed clauses and literals at each depth (and even multiple times as the same depth). By contrast, if a clause fails, Popper learns constraints from the failure so it never tries that clause (or its specialisations) again.

Popper outperforms ILASP2i and ILASP3 on both datasets. ILASP2i only learns programs with four (small dataset) and one (big dataset) body literals. ILASP3 only learns programs with four (small dataset) and one (big dataset) body literals. ILASP2i and ILASP3 both struggle on this problem because they pre-compute every clause in the hypothesis space, which means that they struggle to learn clauses with many body literals. By contrast, Popper can learn programs with ten body literals on both datasets.

Aleph outperforms Popper on the small dataset when $n > 8$. However, on the big dataset, Popper outperforms Aleph when $n > 3$.

Overall, the results from this experiment suggest that (i) the answer to question **Q1** is certainly yes, constraints improve learning performance, (ii) the answer to **Q2** is that Popper scales well in terms of the number of body literals in a solution and the number of background relations, and (iii) the answer to **Q3** is that Popper can outperform other ILP systems when varying the optimal solution size and the number of background relations.

5.2 Robots

The purpose of this second experiment is to evaluate how well Popper scales with respect to the domain size (i.e. the constant signature). We therefore need a problem where we can control the domain size. We consider a robot strategy learning problem (Cropper and Muggleton 2015). There is a robot in a $n \times n$ grid world. Given an arbitrary start position, the goal is to learn a general strategy to move the robot to the topmost row in the grid. For instance, for a 10×10 world and the start position (2, 2), the goal is to move to position (2, 10). The domain contains all possible robot positions. We therefore vary the domain size by varying n , the size of the world. The optimal solution is a recursive strategy for *keep moving upwards until you are at the top row*. To reiterate, we purposely fix the optimal solution so that the only variable in the experiment is the domain size (i.e. the grid world size), which we progressively increase to evaluate how well the systems scale.

$P(A) : \neg Q(A).$ $P(A) : \neg Q(A), R(A).$ $P(A) : \neg Q(A, B), R(B).$ $P(A) : \neg Q(A, B), P(B).$ $P(A) : \neg Q(A, B), R(A, B).$	$P(A, B) : \neg Q(B, A).$ $P(A, B) : \neg Q(A, B), R(A, B).$ $P(A, B) : \neg Q(A), R(A, B).$ $P(A, B) : \neg Q(A, B), R(B).$ $P(A, B) : \neg Q(A, C), R(C, B).$ $P(A, B) : \neg Q(A, C), P(C, B).$
---	---

Fig. 10 The metarules used by Metagol in the robot and list transformation experiments

5.2.1 Materials

We consider two representations: a representation for Popper, Enumerate, Metagol, and Aleph, and then a representation designed to help ILASP solve the problem. When given the Prolog representation, neither ILASP2i nor ILASP3 could solve any of the problems because of the grounding problem. In both representations, we provide as BK four dyadic relations, *move_right*, *move_left*, *move_up*, and *move_down*, that change the state, e.g. *move_right*((2,2),(3,2)), and four monadic relations, *at_top*, *at_bottom*, *at_left*, and *at_right*, that check the state. The exact language biases used can be found in “Appendix C”.

Prolog representation In the Prolog representation, an example is an atom of the form $f(s_1, s_2)$, where s_1 and s_2 represent start and end states. A state is a pair of discrete coordinates (x, y) denoting the column (x) and row (y) position of the robot.

ILASP representation When given the Prolog representation, neither ILASP2i nor ILASP3 could solve any of the problems in this experiment because of the grounding problem. We therefore asked Mark Law to help us design a more suitable representation. In this representation, an example is an atom of the form $f(s_2)$ where s_2 represents the end state. Each example is a distinct ILASP example (a partial interpretation) with its own *context*, where the start state is given in the context as *start_state*(s_1). This representation alleviates the grounding problem of the Prolog representation.

ILASP2i and ILASP3 settings We run both ILASP2i and ILASP3 with the same settings, so we again refer to both as ILASP. We run ILASP with the ‘no constraints’, ‘no aggregates’, ‘disable implication’, ‘disable propagation’, and ‘simple contexts’ flags. We tell ILASP that each BK relation is *positive*, *anti_reflexive*, and *symmetric*. We also employ a set of ‘bias constraints’ to reduce the hypothesis space. We also restrict some of the recall values for the BK relations. We set ILASP to use at most four unique variables and at most three body literals (‘-ml=3’ and ‘-max-rule-length=4’). The full language bias restrictions can be found in the “Appendix C”.

Metagol settings We provide Metagol with the metarules in Fig. 10. These metarules constitute an almost¹⁵ complete set of metarules for a singleton-free fragment of monadic and dyadic Datalog (Cropper and Tourret 2020).

Popper settings We allow Popper and Enumerate to use at most four unique variables per clause and at most three body literals (which match the ILASP settings), and at most three clauses.

Aleph settings We set the maximum variable depth and clause length to six and set the maximum number of search nodes to 30,000.

¹⁵ It is impossible to generate a finite and complete set of metarules for a singleton-free fragment of monadic and dyadic Datalog (Cropper and Tourret 2020).

5.2.2 Methods

We run the experiment with an $n \times n$ grid world for each n in $\{10, 20, \dots, 100\}$. To generate examples, for start states, we uniformly sample positions that are not at the top of the world. For the positive examples, the end state is the topmost position, e.g. (x, n) where n is the grid size. For negative examples, we randomly sample start and end states and reject the example if it is a positive example. To ensure that there are some negative examples with the topmost position, in 25% of the examples we set the end position to be the topmost row of column y , but ensure that the start position is not y . We sample with replacement 20 positive and 20 negative training examples, and 1000 positive and 1000 negative testing examples. The default predictive accuracy is therefore 50%. We measure predictive accuracies and learning times. We enforce a timeout of one minute per task. If a system fails to learn a solution in the given time then it only achieves default predictive accuracy (50%). We repeat each experiment ten times and plot the standard error.

5.2.3 Results

Figure 11 shows the results. Popper achieves the best predictive accuracy out of all the systems. Enumerate is the second best performing system, although it does not always learn the optimal solution. Popper is substantially quicker than Enumerate (on average about 40 times quicker) and is the fastest of all the systems. The learning time of Popper slightly decreases as the grid size grows. The reason for this is twofold. First, when the grid world is small, there are often many small programs that cover some of the positive examples but none of the negative examples, such as:

$$f(S1, S2):- \text{move_up}(S1, S3), \text{move_up}(S3, S2).$$

Because they cover some of the examples, Popper cannot completely rule them out. However, as the grid size grows, these smaller programs are less likely to cover the examples because the examples are more spread out over the grid. Second, solutions have either five or six literals, with smaller solutions becoming more likely with increasing world size. These reasons explain why the predictive accuracy of Enumerate improves as the grid size grows. The reason that the learning time of Popper does not increase is that the domain size has no influence on the size of the learning from failures hypothesis space (Proposition 1). The only influence the grid size has is any overhead in executing the induced Prolog program on larger grids. This result suggests that Popper can scale well with respect to the domain size.

Popper outperforms Metagol in all cases. For a small 10×10 grid world, Metagol learns the optimal solution and does so quicker than Popper (Metagol takes 1 second compared to Popper which takes 9 seconds). However, as the grid size grows, Metagol's performance quickly degrades. For a grid size greater than 20, Metagol almost always times out before finding a solution. The reason is that Metagol searches for a hypothesis by inducing and executing partial programs over the examples. In other words, Metagol uses the examples to guide the hypothesis search. As the grid size grows, there are more partial programs to construct, so its performance suffers. Note that when Metagol learns a solution, it is always an accurate solution.

Popper outperforms ILASP2i and ILASP3 both in terms of predictive accuracies and learning times. ILASP3 cannot learn any solutions in the given time, even for the 10×10 world. ILASP2i initially learns solutions in the given time limit, but struggles as the grid size grows. Note that when ILASP2i learns a solution, it is always an accurate solution.

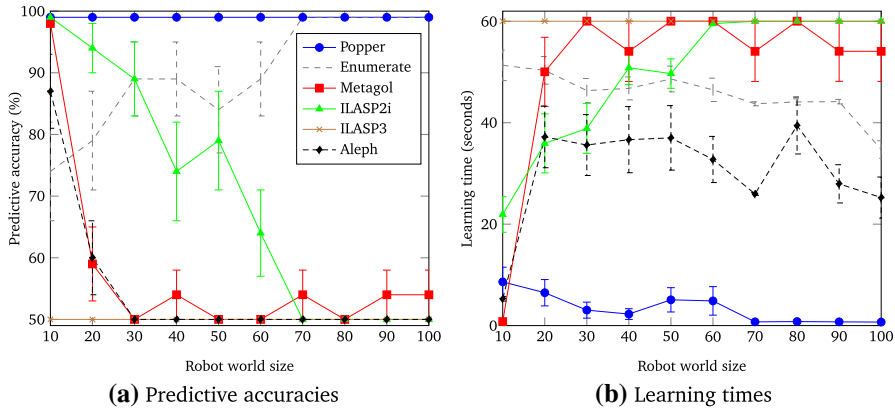


Fig. 11 Robots experimental results when varying the world size, which corresponds to the domain size

ILASP2i outperforms ILASP3 because once ILASP2i finds a solution it terminates. By contrast, ILASP3 finds one hypothesis schema that guarantees coverage of the example (which, in this special case, also implies finding a solution), then carries on to find alternative hypothesis schemas. The extra work done by ILASP3 is needed when learning general ASP programs, but in this special case (where there no ILASP negative examples) it is unnecessary and computationally expensive. We refer the reader to Law’s thesis (Law 2018) for a detailed comparison of ILASP2i and ILASP3.¹⁶

Popper outperforms Aleph. For small grid worlds, Aleph sometimes learns programs that generalise to the training set (such as move up three times). But as the grid size grows, Aleph struggles because it struggles to learn recursive programs.

Overall, the results from this experiment suggest that (i) the answer to question **Q1** is certainly yes, constraints improve learning performance, (ii) the answer to **Q2** is that Popper scales well in terms of the domain size, and (iii) the answer to **Q3** is that Popper can outperform other ILP systems when varying the domain size.

5.3 List transformation problem

The purpose of this third experiment is to evaluate how well Popper performs on difficult (mostly recursive) list transformation problems. Learning recursive programs has long been considered a difficult problem in ILP (Muggleton et al. 2012) and most ILP and program synthesis systems cannot learn recursive programs. Because ILASP2i and ILASP3 do not support lists, we only compare Popper, Enumerate, Metagol, and Aleph.

5.3.1 Materials

We evaluate the systems on the ten list transformation tasks shown in Table 4. These tasks include a mix of monadic (e.g. *evens* and *sorted*), dyadic (e.g. *droplast* and *finddup*), and triadic (*dropk*) target predicates. The tasks also contain a mix of functional (e.g. *last* and *len*) and relational problems (e.g. *finddup* and *member*). These tasks are extremely difficult for ILP systems. To learn solutions that generalise, an ILP system needs

¹⁶ We thank Mark Law for this explanation.

to support recursion and large domains. As far as we are aware, no existing ILP system can learn optimal solutions for all of these tasks without being provided with a strong inductive bias.¹⁷

We give each system the following dyadic relations *head*, *tail*, *decrement*, *geq* and the monadic relations *empty*, *zero*, *one*, *even*, and *odd*. We also include the dyadic relation *increment* in the `len` experiment. We had to remove this relation from the BK for the other experiments because when given this relation Metagol runs into infinite recursion¹⁸ on almost every problem and could not find any solutions. We also include *member/2* in the `find duplicate` problem. We also include *cons/3* in the `addhead`, `dropk`, and `droplast` experiments. We exclude this relation from the other experiments because Metagol does not easily support triadic relations. The exact language biases used can be found in Appendix D.

Metagol settings For Metagol, we use almost the same metarules as in the previous robot experiment (Fig. 10). However, when given the *inverse* metarule $P(A, B) \leftarrow Q(B, A)$, Metagol could not learn any solution, again because of infinite recursion. To aid Metagol, we therefore replace the *inverse* metarule with the *identity* metarule, i.e. $P(A, B) \leftarrow Q(A, B)$. In addition, when we first ran the experiment with randomly ordered examples, we found that Metagol struggled to find solutions for all the problems (except `member`). The reason is that Metagol is sensitive to the order of examples because it uses the examples in the order they are given to induce a hypothesis. Therefore, to aid Metagol, we provide the examples in increasing size (i.e. the length of the input lists).

Popper and Enumerate settings We set Popper and Enumerate to use at most five unique variables, at most five body literals, and at most two clauses. In Sect. 5.5, we evaluate how sensitive Popper is to these parameters. For each BK relation, we also provide both systems with simple types and argument directions (whether input or output). Because Popper and Enumerate can generate non-terminating Prolog programs, we set both systems to use a testing timeout of 0.1 seconds per example. If a program times out, we view it as a failure.

Aleph settings We give Aleph identical mode declarations and determinations to Popper and Enumerate. We set the maximum variable depth and clause length to six and set the maximum number of search nodes to 30,000.

5.3.2 Methods

For each problem, we generate 10 positive and 10 negative training examples, and 1000 positive and 1000 negative testing examples. The default predictive accuracy is therefore 50%. Each list is randomly generated and has a maximum length of 50. We sample the list elements uniformly at random from the set $\{1, 2, \dots, 100\}$. We measure the predictive accuracy and learning times. We enforce a timeout of five minutes per task. We repeat each experiment 10 times and plot the standard error.

¹⁷ As mentioned in Sect. 2.3, some inverse entailment methods (Muggleton 1995) might sometimes learn solutions for them. However, these approaches need an example to learn the base case of a recursive program and then an example to learn the inductive case. Moreover, these approaches would not be guaranteed to learn the optimal solution. Metagol could possibly learn solutions for them if given the exact metarules needed, but that requires that you know the solution before you try to learn it.

¹⁸ Because Metagol induces hypotheses by partially constructing and evaluating hypotheses, it is very difficult to impose a timeout on a particular hypothesis, which we can easily do with Popper.

Table 4 Example solutions for the list transformation problems

Name	Description	Example solution
addhead	Prepend the head three times	addhead(A,B) :- head(A,C), cons(C,A,D), cons(C,D,E), cons(C,E,B)
dropk	Drop the first k elements	dropk(A,B,C) :- one(B), tail(A,C) dropk(A,B,C) :- tail(A,D), decrement(B,E), dropk(D,E,C)
droplast	Drop the last element	droplast(A,B) :- tail(A,B), empty(B) droplast(A,B) :- tail(A,C), droplast(C,D), head(A,E), cons(E,D,B)
evens	Check all elements are even	evens(A) :- empty(A) evens(A) :- head(A,B), even(B), tail(A,C), evens(C)
finddup	Find duplicate elements	finddup(A,B) :- head(A,B), tail(A,C), member(B,C) finddup(A,B) :- tail(A,C), finddup(C,B)
last	Last element	last(A,B) :- tail(A,C), empty(C), head(A,B) last(A,B) :- tail(A,C), last(C,B)
len	Calculate list length	len(A,B) :- empty(A), zero(B) len(A,B) :- tail(A,C), len(C,D), succ(D,B)
member	Member of a list	member(A,B) :- head(A,B) member(A,B) :- tail(A,C), member(C,B)
sorted	Check list is sorted	sorted(A) :- tail(A,B), empty(B) sorted(A) :- head(A,B), tail(A,C), head(C,D), geq(D,B), sorted(C)
threesame	First three elements are identical	threesame(A) :- head(A,B), tail(A,C), head(C,B), tail(C,D), head(D,B)

Table 5 List transformation predictive accuracies

Name	Popper	Enumerate	Metagol	Aleph
addhead	100 ± 0	100 ± 0	n/a	90 ± 10
dropk	100 ± 0	50 ± 0	n/a	50 ± 0
droplast	100 ± 0	50 ± 0	n/a	50 ± 0
evens	100 ± 0	100 ± 0	50 ± 0	50 ± 0
finddup	98 ± 0	50 ± 0	100 ± 0	50 ± 0
last	100 ± 0	50 ± 0	100 ± 0	50 ± 0
len	100 ± 0	50 ± 0	50 ± 0	50 ± 0
member	100 ± 0	100 ± 0	100 ± 0	50 ± 0
sorted	100 ± 0	50 ± 0	50 ± 0	68 ± 2
threesame	99 ± 0	99 ± 0	99 ± 0	99 ± 0

We round accuracies to integer values. The error is standard error

Table 6 List transformation learning times

Name	Popper	Enumerate	Metagol	Aleph
addhead	0.5 ± 0	2 ± 0	n/a	103 ± 49
dropk	0.8 ± 0	300 ± 0	n/a	3 ± 0.2
droplast	3 ± 0.1	300 ± 0	n/a	300 ± 0
evens	4 ± 0.1	159 ± 0.1	300 ± 0	1 ± 0
finddup	36 ± 2	300 ± 0	2 ± 0.5	1 ± 0.1
last	2 ± 0.1	300 ± 0	0.7 ± 0.2	1 ± 0.1
len	12 ± 0.3	300 ± 0	300 ± 0	1 ± 0
member	0.4 ± 0.1	7 ± 0	0.3 ± 0	0.9 ± 0.1
sorted	23 ± 1	300 ± 0	300 ± 0	0.8 ± 0
threesame	0.2 ± 0.1	0.4 ± 0.2	0.9 ± 0.3	0.5 ± 0

We round times over 1 second to the nearest second. The error is standard error. Note that although Aleph is sometimes faster than Popper, it only learns accurate solutions for `addhead` and `threesame`

5.3.3 Results

Table 5 shows that Popper equals or outperforms Enumerate on all the tasks in terms of predictive accuracies. When a system has 50% accuracy, it means that the system has failed to learn a program in the given amount of time, and so achieves the default accuracy. Table 6 shows that Popper substantially outperforms Enumerate in terms of learning times. For instance, whereas it takes Enumerate 159 seconds to find an `evens` program, it takes Popper only four seconds. Table 7 decomposes the learning times of Popper.

Table 5 shows that Popper equals or outperforms Metagol on all the tasks in terms of predictive accuracies, except the `finddup` problem, where Metagol has a 2% higher predictive accuracy. Table 5 also shows that Aleph struggles to learn solutions to these problems. The exceptions are `addhead` and `threesame`, which do not need recursion.

Overall, the results from this experiment suggest that (i) the answer to question **Q1** is again yes, constraints improve learning performance, and (ii) Popper can outperform other ILP systems when learning complex and recursive list transformation programs.

Table 7 Decomposition of Popper learning times

Name	Time	Grounding	Solving
addhead	0.5 ± 0	0.1 ± 0	0.2 ± 0
dropk	0.8 ± 0	0.3 ± 0	0.1 ± 0
droplast	3 ± 0.1	0.4 ± 0.1	1 ± 0
evens	4 ± 0.1	1 ± 0	1 ± 0.1
finddup	36 ± 2	25 ± 1	7 ± 0.5
last	2 ± 0.1	1 ± 0	0.5 ± 0
len	12 ± 0.3	7 ± 0.2	2 ± 0.1
member	0.4 ± 0.1	0.1 ± 0	0.1 ± 0
sorted	23 ± 1	12 ± 0.9	8 ± 0.6
threesame	0.2 ± 0.1	0 ± 0	0 ± 0

The unaccounted time (time not grounding or solving) is mostly the overhead of testing the induced Prolog programs

5.4 Scalability

Our buttons experiment (Experiment 5.1) showed that Popper scales well in the size of the optimal solution size and the number of background relations. Our robot experiment (Experiment 5.2) showed that Popper scales well in the size of the domain. The purpose of this experiment is to evaluate how well Popper scales in terms of the (i) number of examples and (ii) the size of examples. To do so, we repeat the `last` experiment from Sect. 5.3, where Popper and Metagol achieved similar performance.

5.4.1 Materials

We use the same materials as Sect. 5.3.

5.4.2 Settings

We run two experiments. In the first experiment we vary the number of examples. In the second experiment we vary the size of the examples (the size of the input list). For each experiment, we measure the predictive accuracy and learning times averaged over 10 repetitions.

Number of examples For each n in $\{1000, 2000, \dots, 10,000\}$, we generate n positive and n negative training examples, and 1000 positive and 1000 negative testing examples and each element is a random integer from the range 1–1000.

Example size For each s in $\{50, 100, 150, \dots, 500\}$, we generate 10 positive and 10 negative training examples, and 1000 positive and 1000 negative testing examples, where each list is of length s and each element is a random integer from the range 1–1000.

5.4.3 Results

Figure 12 shows the results when varying the number of training examples. The predictive accuracies of Popper and Metagol are almost identical until around 10,000 examples. Given this many examples, Metagol struggles to find a solution in one minute and eventually converges on the default predictive accuracy (50%). By contrast, Popper does not struggle to find

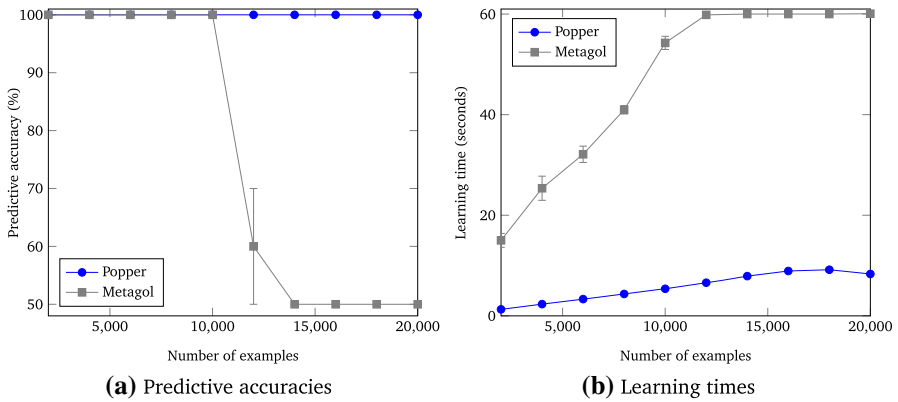


Fig. 12 The experimental results for the last task when varying the number of training examples

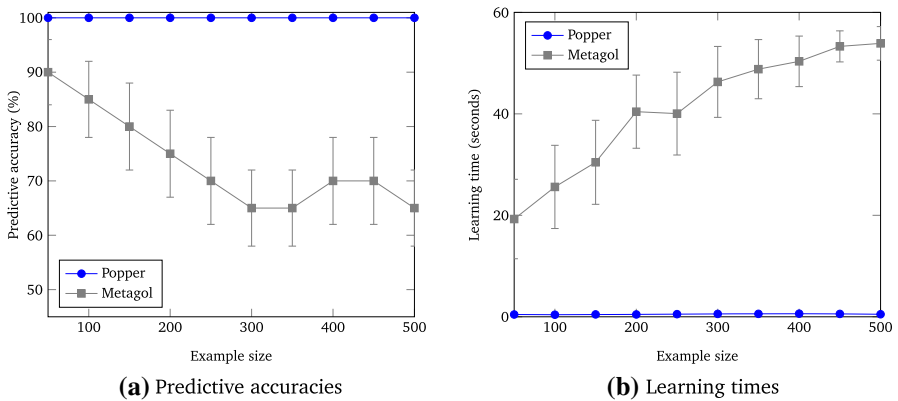


Fig. 13 The experimental results for the last task when varying the size (list length) of training examples

a solution, even given 20,000 examples. Figure 12 shows the learning times of both systems. The learning time of Popper increases linearly simply because of the overhead of testing hypotheses on more examples. The results from this experiment suggest that the answer to **Q2** is that Popper scales well with respect the number of examples.

Figure 13 shows the results when varying the size of the input (i.e. the size of the input list). Popper outperforms Metagol in all cases. The mean learning times of Popper for examples of length 50 and 500 are both less than a second. The reason is that Popper only uses the examples to test a hypothesis, so any increase in running time simply comes from executing the hypotheses using Prolog. By contrast, Metagol’s performance drastically degrades as the size of the examples grows. The mean learning times for Metagol for examples of length 50 and 500 are 20 and 54 seconds respectively. The reason is that Metagol uses the examples to search for a hypothesis by inducing and executing partial programs over the examples. These results suggest that the answer to **Q3** is yes and the answer to **Q2** is that Popper scales well with respect to the size of examples.

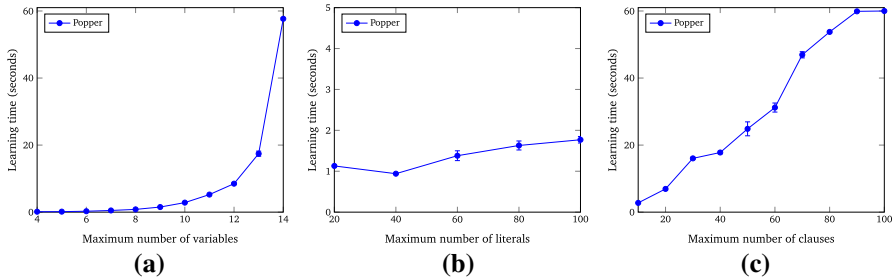


Fig. 14 The experimental results for the `len` task when varying the maximum number of unique variables (a), maximum body literals in a clause (b), and maximum number of clauses (c)

5.5 Sensitivity

The learning from failures hypothesis space (Proposition 1) is a function of the number of predicate declarations and three other variables:

- the maximum number of unique variables in a clause
- the maximum number of body literals allowed in a clause
- the maximum number of clauses allowed in a hypothesis

The purpose of this experiment is to evaluate how sensitive Popper is to these parameters. To do so, we repeat the `len` experiment from Sect. 5.3 with the same BK, settings, and method, except we run three separate experiments where we vary the three aforementioned parameters.

5.5.1 Results

Figure 14 shows the experimental results. The results show that Popper is sensitive to the maximum number of unique variables, which has a strong influence on learning times. This result follows from Proposition 1 because more variables implies more ways to form literals in a clause. Somewhat surprisingly, doubling the number of variables from 4 to 8 has little difference on performance, which suggests that Popper is robust to imperfect parameters. The results show that Popper is mostly insensitive to the maximum number of body literals in a clause. The main reason is that Popper does not pre-compute every possible clause in the hypothesis space, which is, for instance, the case with ILASP2i and ILASP3. The results show that Popper scales linearly with the maximum number of clauses. Overall these results suggest that Popper scales well with the maximum number of body literals, but can struggle with very large values for the maximum number of unique variables and clauses.

6 Conclusions and limitations

We have described an ILP approach called *learning from failures* which decomposes the ILP problem into three separate stages: *generate*, *test*, and *constrain*. In the generate stage, the learner generates a hypothesis that satisfies a set of *hypothesis constraints* (Definition 6). In the test stage, the learner tests a hypothesis against training examples. If a hypothesis fails, then, in the constrain stage, the learner learns hypothesis constraints from the failed hypothesis to prune the hypothesis space, i.e. to constrain subsequent hypothesis generation.

In Sect. 3.5, we introduced three types of constraints based on theta-subsumption: *generalisation*, *specialisation*, and *elimination* and proved their soundness in that they do not prune optimal solutions (Definition 14). This loop repeats until either (i) the learner finds an optimal solution, or (ii) there are no more hypotheses to test. We implemented this approach in Popper, an ILP system that learns definite programs. Popper combines ASP and Prolog to support types, learning optimal solutions, learning recursive programs, reasoning about lists and infinite domains, and hypothesis constraints. We evaluated our approach on three diverse domains (toy game problems, robot strategies, and list transformations). Our experimental results show that (i) constraints drastically reduce the hypothesis space, (ii) Popper scales well with respect to the optimal solution size, the number of background relations, the domain size, the number of training examples, and the size of the training examples, and (iii) Popper can substantially outperform existing ILP systems both in terms of predictive accuracies and learning times.

6.1 Limitations and future work

Popper, as implemented in this paper, has several limitations that future work should address.

6.1.1 Features

Non-observational predicate learning Unlike some ILP systems (Muggleton 1995; Katzouris et al. 2016), Popper does not support non-observational predicate learning (non-OPL) (Muggleton 1995), where examples of the target predicates are not directly given. Future work should address this limitation.

Predicate invention Predicate invention has been shown to help reduce the size of target programs, which in turns reduces sample complexity and improves predictive accuracy (Cropper 2019; Dumancic et al. 2019). Popper does not currently support predicate invention.

There are two straightforward ways to support predicate invention. Popper could mimic Metagol by imposing metarules to restrict the form of clauses in a hypothesis and to guide the invention of new predicate symbols—which is easy to do because, as we show in “Appendix A”, Popper can simulate metarules through hypothesis constraints. Alternatively Popper could mimic ILASP by supporting *prescriptive* predicate invention (Law 2018), where the arity and (in ILASP’s case, argument types) are pre-specified by the user. Most of the results in this paper should extend to both approaches.

Negation Popper learns definite programs and tests them using Prolog. Popper can also trivially learn Datalog programs and test them using ASP. In future work, we want to consider learning other types of programs. For instance, most of our pruning techniques (except the elimination constraint) should extend to learning non-monotonic programs, such as Datalog with stratified negation.

Noise Most ILP systems handle noisy (misclassified) examples (Table 1). Popper does not currently support noisy examples. We can address this issue by relaxing when to apply learned hypothesis constraints and by maintaining the best hypotheses tested during the learning, i.e. the hypothesis which entails the most positive and the fewest negative examples. However, noise handling will likely increase learning times and future work should explore this trade-off.

6.1.2 Better search

An advantage of decomposing the learning problem is that it allows for a variety of algorithms and implementations, where each stage can be improved independently of the others. For instance, any improvement to the Popper ASP encoding that generates programs would have a major influence on learning times because it would reduce the number of programs to test. Likewise, we can also optimise the testing step. Future work should consider better search techniques.

6.1.3 Better constraints

Hypothesis constraints are central to our idea. Popper uses both predefined and learned constraints to improve performance. Popper uses predefined constraints to prune redundant programs from the hypothesis space (Sect. 4), such as recursive programs without a base case and subsumption redundant program. Popper also learns constraints from failures. We think the most promising direction for future work is to improve both types of constraints (predefined and learned).

Types Like many ILP systems (Muggleton 1995; Blockeel and De Raedt 1998; Srinivasan 2001; Law et al. 2014; Evans and Grefenstette 2018), Popper supports simple types to prune the hypothesis space. However, more complex types, such as polymorphic types, can achieve better pruning for programs over structured data (Morel et al. 2019). For instance, polymorphic types would allow us to distinguish between using a predicate on a list of integers and on a list of characters. Refinement types (Polikarpova et al. 2016), i.e. types annotated with restricting predicates, could allow a user to specify stronger program properties (other than examples), such as requiring that a reverse program provably has the property that the lengths of the input and output are the same. In future work we want to explore whether we can express such complex types as hypothesis constraints.

Learned constraints The constraints described in Sect. 3.5 prune specialisations and generalisations of a failed hypothesis. However, we have only briefly analysed the properties of these constraints. We showed that these constraints are *sound* (Propositions 3 and 4) in that they do not prune optimal solutions. We have not, however, considered their *completeness*, in that they prune all non-optimal solutions. Indeed, our *elimination constraint*, for the special case of separable definite programs, prunes hypotheses that the generalisation and specialisation constraints miss. In other words, the theory regarding which constraints to use is yet to be developed, and there may be many more constraints to be learned from failed hypotheses, all of which should drastically improve learning performance. By contrast, refinement operators for clauses (Shapiro 1983; De Raedt and Bruynooghe 1993; Nienhuys-Cheng and de Wolf 1997) and theories (Nienhuys-Cheng and de Wolf 1997; Midelfart 1999; Badea 2001) have been studied in detail in ILP. Therefore, we think that this paper opens a new direction of research into identifying and analysing different constraints that we can learn from failed hypotheses.

Acknowledgements We foremost thank Mark Law for all of his help in writing this paper, including finding suitable ILASP representations for the experiments, for answering our many questions on the ILASP systems, and for suggesting much of the text on ILASP3. We thank Tobias Kaminski, Sebastijan Dumančić, and Richard Evans for extremely valuable feedback on the paper. We also thank one of the anonymous reviewers for suggesting a much more efficient constraint encoding that reduced Popper’s learning times in the experiments by almost a half.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

A Popper metarule theory constraints

A.1 Metarules

Let M be an arbitrary metarule, i.e. a second-order Horn clause which quantifies over predicate symbols. For example, $P(A, B):- Q(A, C), R(C, B)$ is known as the chain metarule. All letters are quantified variables, with $P, Q,$ and R being second-order, i.e. needing to be substituted for by predicate symbols.

A.2 From a metarule to literals

Let $M = \text{head}:- \text{body}_1, \dots, \text{body}_m$ be a metarule. We use the clause encoding function *encodeSizedClause* from Sect. 4.5.2 to derive an encoding of a metarule.

Example 14 Consider $M = P(A, B):- Q(A, C), R(C, B)$. Its encoding, *encodeSizedClause* (**Clause**, M), is

```
head_literal(Clause,P,2,(V0,V1)),
body_literal(Clause,Q,2,(V0,V2)),body_literal(Clause,R,2,(V2,V1)),
V0!=V1,V0!=V2,V1!=V2,clause_size(Clause,2).
```

A.3 Asserting metarule conformance

Let M_s be a set of metarules. For each clause of a metarule conformant program, the clause must be an *instance* of one of the metarules in M_s . A clause C is an instance of metarule $M \in M_s$ if there exists substitution θ such that $M\theta = C$.

We introduce two rules to ensure every clause of a generated program is an instance of at least one metarule. The first rule identifies when there exists some metarule for which the clause is an instance. The second rule is a constraint expressing that every clause of a program must be identified as being an instance of at least one metarule.

For each $M \in M_s$, generate the following rule of the first kind:

```
meta_clause(Clause):- encodeSizedClause(Clause,M).
```

The second rule is:

```
:- clause(Clause),not meta_clause(Clause).
```

B Language biases in buttons experiment

B.1 ILASP2i and ILASP3

```
#modeh(1,f, (positive)).
#modeb(1,button1, (positive)).
...
#modeb(1,button20, (positive)).
```

B.2 Popper and Enumerate

```
max_vars(1).
max_clauses(1).
head_pred(f,1).
body_pred(button1,1).
...
body_pred(button20,1).
```

B.3 Aleph

```
:- aleph_set(i,6).
:- aleph_set(clauselength,2).
:- aleph_set(nodes,5000).
:- modeh(f(+var)).
:- modeb(*,button1(+var)).
:- determination(f/1,button1/1).
:- modeb(*,button2(+var)).
...
:- determination(f/1,button20/1).
```

B.4 Metagol

```
metarule(unary1, [P,Q], [P,A], [[Q,A]]).
metarule(unary2, [P,Q,R], [P,A], [[Q,A],[R,A]]).
body_pred(button1/1).
...
body_pred(button20/1).
```

C Language biases in robots experiment

C.1 ILASP2i and ILASP3

```
#modeh(f(var(state)), (positive)).
#modeh(start_state(var(state)), (positive)).
#modeb(3,move_up(var(state),var(state)),
(anti_reflexive,symmetric,positive)).
#modeb(3,move_down(var(state),var(state)),
(anti_reflexive,symmetric,positive)).
#modeb(3,move_left(var(state),var(state)),
(anti_reflexive,symmetric,positive)).
#modeb(3,move_right(var(state),var(state)),
(anti_reflexive,symmetric,positive)).
#modeb(3,at_top(var(state)), (positive)).
#modeb(3,at_bottom(var(state)), (positive)).
#modeb(3,at_left(var(state)), (positive)).
```



```

#modeb(3,at_right(var(state)), (positive)).
#modeb(1,start_state(var(state)), (positive)).

#bias(":- occurs(V, X), #false : occurs(V, Y), Y != X.").
#bias("occurs(X, f(X)) :- head(f(X)).").
#bias("occurs(X, start_state(X)) :- head(start_state(X)).").
#bias("occurs(X, start_state(X)) :- body(start_state(X)).").
#bias("occurs(X, at_top(X)) :- body(at_top(X)).").
#bias("occurs(X, at_bottom(X)) :- body(at_bottom(X)).").
#bias("occurs(X, at_left(X)) :- body(at_left(X)).").
#bias("occurs(X, at_right(X)) :- body(at_right(X)).").
#bias("occurs(X, move_up(X, Y))
      :- body(move_up(X, Y)).").
#bias("occurs(X, move_left(X, Y))
      :- body(move_left(X, Y)).").
#bias("occurs(X, move_right(X, Y))
      :- body(move_right(X, Y)).").
#bias("occurs(X, move_down(X, Y))
      :- body(move_down(X, Y)).").
#bias("occurs(X, move_up(Y, X))
      :- body(move_up(Y, X)).").
#bias("occurs(X, move_left(Y, X))
      :- body(move_left(Y, X)).").
#bias("occurs(X, move_right(Y, X))
      :- body(move_right(Y, X)).").
#bias("occurs(X, move_down(Y, X))
      :- body(move_down(Y, X)).").

```

C.2 Popper and Enumerate

```

max_vars(4).
max_body(3).
max_clauses(3).
head_pred(f,2).
body_pred(f,2).
body_pred(at_top,1).
body_pred(at_bottom,1).
body_pred(at_left,1).
body_pred(at_right,1).
body_pred(move_left,2).
body_pred(move_right,2).
body_pred(move_up,2).
body_pred(move_down,2).
direction(f,0,in).
direction(f,1,out).
direction(move_left,0,in).
direction(move_right,0,in).
direction(move_up,0,in).
direction(move_down,0,in).

```

```

direction(move_left,1,out).
direction(move_right,1,out).
direction(move_up,1,out).
direction(move_down,1,out).
direction(at_top,0,in).
direction(at_bottom,0,in).
direction(at_left,0,in).
direction(at_right,0,in).

```

C.3 Aleph

```

:- aleph_set(i,6).
:- aleph_set(clauselength,6).
:- aleph_set(nodes,50000).
:- modeh,f(+state,-state).
:- modeb(*,move_up(+state,-state)).
:- modeb(*,move_down(+state,-state)).
:- modeb(*,move_left(+state,-state)).
:- modeb(*,move_right(+state,-state)).
:- modeb(*,at_top(+state)).
:- modeb(*,at_bottom(+state)).
:- modeb(*,at_left(+state)).
:- modeb(*,at_right(+state)).
:- determination(f/2,move_up/2).
:- determination(f/2,move_down/2).
:- determination(f/2,move_left/2).
:- determination(f/2,move_right/2).
:- determination(f/2,at_top/1).
:- determination(f/2,at_bottom/1).
:- determination(f/2,at_left/1).
:- determination(f/2,at_right/1).

```

C.4 Metagol

```

body_pred(move_right/2).
body_pred(move_left/2).
body_pred(move_up/2).
body_pred(move_down/2).
body_pred(at_top/1).
body_pred(at_bottom/1).
body_pred(at_left/1).
body_pred(at_right/1).
metarule([P,Q], [P,A], [[Q,A]]).
metarule([P,Q], [P,A], [[Q,A]]).
metarule([P,Q,R], [P,A], [[Q,A,B],[R,B]]).
metarule([P,Q], [P,A], [[Q,A,B],[P,B]]).
metarule([P,Q,R], [P,A], [[Q,A,B],[R,A,B]]).
metarule([P,Q], [P,A,B], [[Q,A,B]]).
metarule([P,Q,R], [P,A,B], [[Q,A,B],[R,A,B]]).
metarule([P,Q,R], [P,A,B], [[Q,A],[R,A,B]]).

```

```

metarule([P,Q,R], [P,A,B], [[Q,A,B],[R,B]]).
metarule([P,Q,R], [P,A,B], [[Q,A,C],[R,C,B]]).
metarule([P,Q], [P,A,B], [[Q,A,C],[P,C,B]]).
metarule([P,Q], [P,A,B], [[Q,B,A]]).

```

D Language biases in lists experiment

D.1 Popper and Enumerate

For each list transformation problem, we have a specific bias to specify the target relations, such as the following bias for the `finddup` problem:

```

head_pred(f,2).
type(f,0,list).
type(f,1,element).
direction(f,0,in).
direction(f,1,out).
body_pred(f,2).

```

For all the problems we include the following biases:

```

max_vars(5).
max_body(5).
max_clauses(2).
body_pred(head,2).
body_pred(tail,2).
body_pred(geq,2).
body_pred(empty,1).
body_pred(even,1).
body_pred(odd,1).
body_pred(one,1).
body_pred(zero,1).
body_pred(decrement,2).
body_pred(increment,2).
body_pred(element,2).
body_pred(cons,2).
type(cons,0,element).
type(cons,1,list).
type(cons,2,list).
direction(cons,0,in).
direction(cons,1,in).
direction(cons,2,out).
type(head,0,list).
type(head,1,element).
direction(head,0,in).
direction(head,1,out).
type(tail,0,list).
type(tail,1,list).
direction(tail,0,in).
direction(tail,1,out).

```

```

type(empty,0,list).
direction(empty,0,in).
type(element,0,list).
type(element,1,element).
direction(element,0,in).
direction(element,1,out).
type(increment,0,element).
type(increment,1,element).
direction(increment,0,in).
direction(increment,1,out).
type(decrement,0,element).
type(decrement,1,element).
direction(decrement,0,in).
direction(decrement,1,out).
type(geq,0,element).
type(geq,1,element).
direction(geq,0,in).
direction(geq,1,in).
type(even,0,element).
direction(even,0,in).
type(odd,0,element).
direction(odd,0,in).
type(one,0,element).
direction(one,0,in).
type(zero,0,element).
direction(zero,0,out).

```

D.2 Aleph

For each list transformation problem, we have a specific bias to specify the target relations, such as the following bias for the `finddup` problem:

```

:- modeh(f(+list,-element)).
:- modeb(*,f(+list,-element)).

```

For all the problems we include the following biases (we we replace `f/2` in the determinations with the correct arity of the target predicate):

Note that `increment` is only given in the `sortedexperiment`, `element` is only given in the `finddupli` experiment, and `cons` is only given in the `addhead`, `dropk`, and `droplast` experiments.

```

:- aleph_set(i,6).\
:- aleph_set(clauselength,6).\
:- aleph_set(nodes,30000).\
:- modeb(head(+list,-element)).
:- modeb(*,tail(+list,-list)).
:- modeb(*,geq(+element,+element)).
:- modeb(*,empty(+list)).
:- modeb(*,even(+element)).

```

```

:- modeb(*, odd(+element)).
:- modeb(*, one(+element)).
:- modeb(*, zero(-element)).
:- modeb(*, decrement(+element, -element)).
:- modeb(*, increment(+element, -element)).
:- modeb(*, element(+list, -element)).
:- modeb(*, cons(+element, +list, -list)).

```

D.3 Metagol

```

body_pred(head/2).
body_pred(tail/2).
body_pred(geq/2).
body_pred(empty/1).
body_pred(even/1).
body_pred(odd/1).
body_pred(one/1).
body_pred(zero/1).
body_pred(decrement/2).
body_pred(increment/2). % ONLY FOR SORTED
body_pred(member/2). % ONLY FOR FIND DUPLICATE

metarule([P,Q], [P,A], [[Q,A]]).
metarule([P,Q], [P,A], [[Q,A]]).
metarule([P,Q,R], [P,A], [[Q,A,B], [R,B]]).
metarule([P,Q], [P,A], [[Q,A,B], [P,B]]).
metarule([P,Q,R], [P,A], [[Q,A,B], [R,A,B]]).
metarule([P,Q], [P,A,B], [[Q,A,B]]).
metarule([P,Q,R], [P,A,B], [[Q,A,B], [R,A,B]]).
metarule([P,Q,R], [P,A,B], [[Q,A], [R,A,B]]).
metarule([P,Q,R], [P,A,B], [[Q,A,B], [R,B]]).
metarule([P,Q,R], [P,A,B], [[Q,A,C], [R,C,B]]).
metarule([P,Q], [P,A,B], [[Q,A,C], [P,C,B]]).
metarule([P,Q], [P,A,B], [[Q,A,B]]).

```

References

- Ahlgren, J., & Yuen, S. Y. (2013). Efficient program synthesis using constraint satisfaction in inductive logic programming. *The Journal of Machine Learning Research*, 14(1), 3649–3682.
- Albarghouthi, A., Koutris, P., Naik, M., & Smith, C. (2017). Constraint-based synthesis of datalog programs. In J. Christopher Beck (Ed.), *Principles and practice of constraint programming—23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28–September 1, 2017, Proceedings, volume 10416 of Lecture Notes in Computer Science* (pp. 689–706). Springer.
- Athakravi, D., Alrajeh, D., Broda, K., Russo, A., & Satoh, K. (2014). Inductive learning using constraint-driven bias. In J. Davis & J. Ramon (Eds.), *Inductive Logic Programming—24th International Conference, ILP 2014, Nancy, France, September 14–16, 2014, Revised Selected Papers, volume 9046 of Lecture Notes in Computer Science* (pp. 16–32). Springer.
- Athakravi, D., Corapi, D., Broda, K., & Russo, A. (2013). Learning through hypothesis refinement using answer set programming. In G. Zaverucha, V. S. Costa, & A. Paes (Eds.), *Inductive Logic Programming—23rd International Conference, ILP 2013, Rio de Janeiro, Brazil, August 28–30, 2013, Revised Selected Papers, volume 8812 of Lecture Notes in Computer Science* (pp 31–46). Springer.

- Badea, L. (2001). A refinement operator for theories. In C. Rouveirol & M. Sebag (Eds.), *Inductive Logic Programming, 11th International Conference, ILP 2001, Strasbourg, France, September 9–11, 2001, Proceedings, volume 2157 of Lecture Notes in Computer Science* (pp. 1–14). Springer.
- Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., & Tarlow, D. (2017). Deepcoder: Learning to write programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24–26, 2017, Conference Track Proceedings*. OpenReview.net.
- Blockeel, H., & De Raedt, L. (1998). Top-down induction of first-order logical decision trees. *Artificial Intelligence, 101*(1–2), 285–297.
- Bratko, I. (1999). Refining complete hypotheses in ILP. In S. Dzeroski & P. A. Flach (Eds.), *Inductive Logic Programming, 9th International Workshop, ILP-99, Bled, Slovenia, June 24–27, 1999, Proceedings, volume 1634 of Lecture Notes in Computer Science* (pp. 44–55). Springer.
- Church, A. (1936). A note on the entscheidungsproblem. *The Journal of Symbolic Logic, 1*(1), 40–41.
- Cohen, W. W. (1994). Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence, 68*(2), 303–366.
- Corapi, D., Russo, A., & Lupu, E. (2010). Inductive logic programming as abductive search. In M. V. Hermenegildo & T. Schaub (Eds.), *Technical Communications of the 26th International Conference on Logic Programming, ICLP 2010, July 16–19, 2010, Edinburgh, Scotland, UK, volume 7 of LIPIcs* (pp. 54–63). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.
- Corapi, D., Russo, A., & Lupu, E. (2011). Inductive logic programming in answer set programming. In S. Muggleton, A. Tamaddoni-Nezhad, & F. A. Lisi (Eds.), *Inductive Logic Programming—21st International Conference, ILP 2011, Windsor Great Park, UK, July 31–August 3, 2011, Revised Selected Papers, volume 7207 of Lecture Notes in Computer Science* (pp. 91–97). Springer.
- Costa, V. S., Srinivasan, A., Camacho, R., Blockeel, H., Demoen, B., Janssens, G., et al. (2003). Query transformations for improving the efficiency of ILP systems. *Journal of Machine Learning Research, 4*, 465–491.
- Cropper, A. (2019). Playgol: Learning programs through play. In S. Kraus (Ed.), *Proceedings of the twenty-eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10–*16, 2019* (pp. 6074–6080). ijcai.org.
- Cropper, A. (2020). Forgetting to learn logic programs. In *The thirty-fourth AAAI Conference on Artificial Intelligence, AAAI 2020, New York, NY, USA, February 7–12, 2020* (pp. 3676–3683). AAAI Press.
- Cropper, A., & Dumancic, S. (2020). Inductive logic programming at 30: A new introduction. *CoRR, arXiv:2008.07912*.
- Cropper, A., & Dumancic, S. (2020). Learning large logic programs by going beyond entailment. In C. Bessiere (Ed.), *Proceedings of the twenty-ninth International Joint Conference on Artificial Intelligence, IJCAI 2020* (pp. 2073–2079). ijcai.org.
- Cropper, A., Dumancic, S., & Muggleton, S. H. (2020). Turning 30: New ideas in inductive logic programming. In C. Bessiere (Ed.) *Proceedings of the twenty-ninth International Joint Conference on Artificial Intelligence, IJCAI 2020* (pp. 4833–4839). ijcai.org.
- Cropper, A., Evans, R., & Law, M. (2020). Inductive general game playing. *Machine Learning, 109*(7), 1393–1434.
- Cropper, A., Morel, R., & Muggleton, S. (2020). Learning higher-order logic programs. *Machine Learning, 109*(7), 1289–1322.
- Cropper, A., & Muggleton, S. H. (2015). Learning efficient logical robot strategies involving composable objects. In Q. Yang & M. J. Wooldridge (Eds.), *Proceedings of the twenty-fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25–31, 2015* (pp. 3423–3429). AAAI Press.
- Cropper, A., & Muggleton, S. H. (2016). *Metagol system*. <https://github.com/metagol/metagol>
- Cropper, A., Tamaddoni-Nezhad, A., & Muggleton, S. H. (2015). Meta-interpretive learning of data transformation programs. In K. Inoue, H. Ohwada, & A. Yamamoto (Eds.), *Inductive Logic Programming—25th International Conference, ILP 2015, Kyoto, Japan, August 20–22, 2015, Revised Selected Papers, volume 9575 of Lecture Notes in Computer Science* (pp. 46–59). Springer.
- Cropper, A., & Tourret, S. (2020). Logical reduction of metarules. *Machine Learning, 109*(7), 1323–1369.
- De Raedt, L. (2008). *Logical and relational learning. Cognitive technologies*. Berlin: Springer.
- De Raedt, L., & Bruynooghe, M. (1992). Interactive concept-learning and constructive induction by analogy. *Machine Learning, 8*, 107–150.
- De Raedt, L., & Bruynooghe, M. (1993). A theory of clausal discovery. In R. Bajcsy (Ed.), *Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28–September 3, 1993* (pp. 1058–1063). Morgan Kaufmann.
- Dumancic, S., Guns, T., Meert, W., & Blockeel, H. (2019). Learning relational representations with auto-encoding logic programs. In S. Kraus (Ed.), *Proceedings of the twenty-eighth International Joint*

- Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10–16, 2019* (pp. 6081–6087). ijcai.org.
- Ellis, K., Morales, L., Sablé-Meyer, M., Solar-Lezama, A., & Tenenbaum, J. (2018). Learning libraries of subroutines for neurally-guided bayesian program induction. *NeurIPS, 2018*, 7816–7826.
- Ellis, K., Nye, M. I., Yewen, P., Sosa, F., Tenenbaum, J., & Solar-Lezama, A. (2019). Write, execute, assess: Program synthesis with a REPL. In H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, & R. Garnett (Eds.), *Advances in neural information processing systems 32: Annual conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8–14 December 2019, Vancouver, BC, Canada* (pp. 9165–9174).
- Evans, R., & Grefenstette, E. (2018). Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research, 61*, 1–64.
- Evans, R., Hernández-Orallo, J., Welbl, J., Kohli, P., & Serfaty, M. J. (2019). Making sense of sensory input. *CoRR, arXiv:1910.02227*.
- Feng, Y., Martins, R., Bastani, O., & Dillig, I. (2018). Program synthesis using conflict-driven learning. In J. S. Foster & D. Grossman (Eds.), *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18–22, 2018* (pp. 420–435). ACM.
- Feser, J. K., Chaudhuri, S., & Dillig, I. (2015). Synthesizing data structure transformations from input–output examples. In D. Grove & S. Blackburn (Eds.), *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15–17, 2015* (pp. 229–239). ACM.
- Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2014). Clingo = ASP + control: Preliminary report. *CoRR, arXiv:1405.3694*.
- Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2019). Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming, 19*(1), 27–82.
- Kaminski, T., Eiter, T., & Inoue, K. (2018). Exploiting answer set programming with external sources for meta-interpretive learning. *Theory and Practice of Logic Programming, 18*(3–4), 571–588.
- Katpouris, N., Artikis, A., & Paliouras, G. (2016). Online learning of event definitions. *TPLP, 16*(5–6), 817–833.
- Lake, B. M., Ullman, T. D., Tenenbaum, J. B., & Gershman, S. J. (2016). Building machines that learn and think like people. *CoRR, arXiv:1604.00289*.
- Law, M. (2018). *Inductive learning of answer set programs*. Ph.D. thesis, Imperial College London, UK.
- Law, M., Russo, A., & Broda, K. (2014). Inductive learning of answer set programs. In E. Fermé & J. Leite (Eds.), *Logics in Artificial Intelligence—14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24–26, 2014. Proceedings, volume 8761 of Lecture Notes in Computer Science* (pp. 311–325). Springer.
- Law, M., Russo, A., & Broda, K. (2016). Iterative learning of answer set programs from context dependent examples. *Theory and Practice of Logic Programming, 16*(5–6), 834–848.
- Lin, D., Dechter, E., Ellis, K., Tenenbaum, J. B., & Muggleton, S. (2014). Bias reformulation for one-shot function induction. In T. Schaub, G. Friedrich, & B. O'Sullivan (Eds.), *ECAI 2014—21st European Conference on Artificial Intelligence, 18–22 August 2014, Prague, Czech Republic—Including Prestigious Applications of Intelligent Systems (PAIS 2014), volume 263 of frontiers in artificial intelligence and applications* (pp. 525–530). IOS Press.
- Lloyd, J. W. (2012). *Foundations of logic programming*. Berlin: Springer.
- Michie, D. (1988). Machine learning in the next five years. In D. H. Sleeman (Ed.), *Proceedings of the third European Working Session on Learning, EWSL 1988, Turing Institute, Glasgow, UK, October 3–5, 1988* (pp. 107–122). Pitman Publishing.
- Midelfart, H. (1999). A bounded search space of clausal theories. In S. Dzeroski, & P. A. Flach (Eds.), *Inductive Logic Programming, 9th International Workshop, ILP-99, Bled, Slovenia, June 24–27, 1999, Proceedings, volume 1634 of Lecture Notes in Computer Science* (pp. 210–221). Springer.
- Morel, R., Cropper, A., & Luke Ong, C.-H. (2019). Typed meta-interpretive learning of logic programs. In F. Calimeri, N. Leone, & M. Manna (Eds.), *Logics in Artificial Intelligence—16th European Conference, JELIA 2019, Rende, Italy, May 7–11, 2019, Proceedings, volume 11468 of Lecture Notes in Computer Science* (pp. 198–213). Springer.
- Muggleton, S. (1991). Inductive logic programming. *New Generation Computing, 8*(4), 295–318.
- Muggleton, S. (1995). Inverse entailment and progol. *New Generation Computing, 13*(3&4), 245–286.
- Muggleton, S., De Raedt, L., Poole, D., Bratko, I., Flach, P. A., Inoue, K., et al. (2012). ILP turns 20—Biography and future challenges. *Machine Learning, 86*(1), 3–23.
- Muggleton, S. H., Lin, D., Pahlavi, N., & Tamaddoni-Nezhad, A. (2014). Meta-interpretive learning: Application to grammatical inference. *Machine Learning, 94*(1), 25–49.

- Muggleton, S. H., Lin, D., & Tamaddoni-Nezhad, A. (2015). Meta-interpretive learning of higher-order dyadic Datalog: Predicate invention revisited. *Machine Learning*, 100(1), 49–73.
- Nienhuys-Cheng, S.-H., & de Wolf, R. (1997). *Foundations of inductive logic programming*. Secaucus, NJ: Springer-Verlag New York, Inc.
- Plotkin, G. D. (1971). *Automatic methods of inductive inference*. Ph.D. thesis, Edinburgh University.
- Polikarpova, N., Kuraj, I., & Solar-Lezama, A. (2016). Program synthesis from polymorphic refinement types. In C. Krinztz, & E. Berger (Eds.), *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13–17, 2016* (pp. 522–538). ACM.
- Popper, K. (2005). *The logic of scientific discovery*. London: Routledge.
- Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, 5, 239–266.
- Raghothaman, M., Mendelson, J., Zhao, D., Naik, M., & Scholz, B. (2020). Provenance-guided synthesis of datalog programs. *PACMPL*, 4(POPL), 62:1–62:27.
- Ray, O. (2009). Nonmonotonic abductive inductive learning. *Journal of Applied Logic*, 7(3), 329–340.
- Schüller, P., & Benz, M. (2018). Best-effort inductive logic programming via fine-grained cost-based hypothesis generation—The inspire system at the inductive logic programming competition. *Machine Learning*, 107(7), 1141–1169.
- Shapiro, E. Y. (1983). *Algorithmic program debugging*. Cambridge, MA: MIT Press.
- Solar-Lezama, A., Jones, C. G., & Bodík, R. (2008). Sketching concurrent data structures. In R. Gupta & S. P. Amarasinghe (Eds.), *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7–13, 2008* (pp. 136–148). ACM.
- Srinivasan, A. (2001). *The ALEPH manual*. Machine Learning at the Computing Laboratory: Oxford University.
- Srinivasan, A., & Kothari, R. (2005). A study of applying dimensionality reduction to restrict the size of a hypothesis space. In S. Kramer & B. Pfahringer (Eds.), *Inductive Logic Programming, 15th International Conference, ILP 2005, Bonn, Germany, August 10–13, 2005, Proceedings, volume 3625 of Lecture Notes in Computer Science* (pp. 348–365). Springer.
- Tärnlund, S. Å. (1977). Horn clause computability. *BIT*, 17(2), 215–226.
- Wang, W. Y., Mazaitis, K., & Cohen, W. W. (2014). Structure learning via parameter learning. In J. Li, X. S. Wang, M. N. Garofalakis, I. Soboroff, T. Suel, & M. Wang (Eds.), *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3–7, 2014* (pp. 1199–1208). ACM.
- Weinzierl, A. (2017). Blending lazy-grounding and CDNL search for answer-set solving. In M. Balduccini & T. Janhunen (Eds.), *Logic Programming and Nonmonotonic Reasoning—14th International Conference, LPNMR 2017, Espoo, Finland, July 3–6, 2017, Proceedings, volume 10377 of Lecture Notes in Computer Science* (pp. 191–204). Springer.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.