Machine learning manuscript No.
(will be inserted by the editor)

Learning programs by learning from failures

Andrew Cropper · Rolf Morel

the date of receipt and acceptance should be inserted later

Abstract We introduce *learning programs by learning from failures*. In this approach, an inductive logic programming (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 (1) the learner finds a hypothesis that entails all the positive and none of the negative examples, or (2) there are no more hypotheses to test. We implement our idea in Popper, an ILP system which combines answer set programming and Prolog. Popper supports infinite domains, reasoning about lists and numbers, learning optimal (textually minimal) programs, and learning recursive programs. Our experimental results on three diverse domains (number theory problems, robot strategies, and list transformations) show that (1) constraints drastically improve learning performance, and (2) Popper can substantially outperform state-of-the-art ILP systems, both in terms of predictive accuracies and learning times.

1 Introduction

Inductive logic programming (ILP) (Muggleton, 1991) is a form of machine learning. Given positive and negative examples of a target predicate and background knowledge

A. Cropper
University of Oxford
E-mail: andrew.cropper@cs.ox.ac.uk

R. Morel
University of Oxford
E-mail: rolf.morel@cs.ox.ac.uk

(BK), the ILP problem is to induce a hypothesis which, with the BK, entails as many positive and as few negative examples as possible. ILP represents the examples, BK, and hypotheses as logic programs (sets of logical rules).

Compared to most machine learning approaches, ILP has several advantages. 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). Moreover, because ILP systems learn logic programs, ILP is also a form of *program synthesis* (Shapiro, 1983), where the goal is to automatically generate computer programs from specifications, typically input/output examples. Finally, because of their symbolic nature, ILP systems naturally support lifelong and transfer learning (Cropper, 2019a), which is considered essential for human-like AI (Lake et al., 2016).

The fundamental problem in ILP is to efficiently search a huge (potentially infinite) hypothesis space (the set of all hypotheses). For instance, in our simplest experiment (Section 5.1), the hypothesis space contains approximately 10^{13} 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 Raedt, 1998; Srinivasan, 2001; Ahlgren and Yuen, 2013). Systems that implement this approach are often very 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 a SAT problem (Corapi et al., 2011; Law et al., 2014; Kaminski et al., 2018; Evans and Grefenstette, 2018; Evans et al., 2019). Systems that implement this approach can often learn optimal and recursive programs. Moreover, they can use efficient SAT solvers based on conflict-driven clause learning. However, the major limitation of these systems is scalability, especially in terms of the domain size.

In this paper, we introduce an ILP approach called *learning programs by learning from failures*, largely inspired by Karl Popper's idea of falsification (Popper, 2005) and Shapiro's seminal program synthesis work (Shapiro, 1983). In our 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). Importantly, in this step, the learner ignores the BK and examples, and instead focuses on finding a constraint satisfying hypothesis. 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. 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 (1) the learner finds a *solution* (a hypothesis that entails all the positive examples and none of the negative examples), or (2) 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

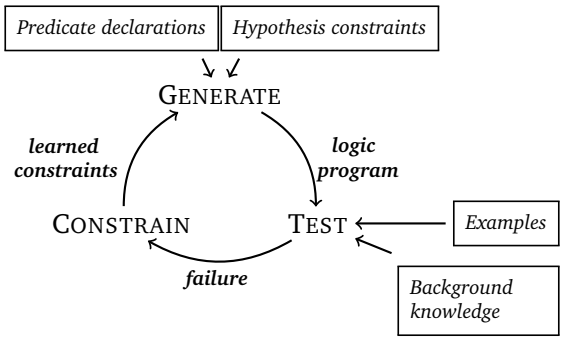


Fig. 1: The generate, test, and constrain loop.

space $\mathscr{H}_1$:

$$\mathscr{H}_1 = \left\{ \begin{array}{l} \mathtt{h}_1 = \{\, \mathtt{last(A,B):-\ head(A,B).}\ \} \\ \mathtt{h}_2 = \{\, \mathtt{last(A,B):-\ head(A,B),empty(A).}\ \} \\ \mathtt{h}_3 = \{\, \mathtt{last(A,B):-\ head(A,B),reverse(A,C),head(C,B).}\ \} \\ \mathtt{h}_4 = \{\, \mathtt{last(A,B):-\ tail(A,C),head(C,B).}\ \} \\ \mathtt{h}_5 = \{\, \mathtt{last(A,B):-\ reverse(A,C),head(C,B).}\ \} \\ \mathtt{h}_6 = \{\, \mathtt{last(A,B):-\ tail(A,C),head(C,B).} \\ \quad\ \mathtt{last(A,B):-\ reverse(A,C),head(C,B).}\ \} \\ \mathtt{h}_7 = \{\, \mathtt{last(A,B):-\ tail(A,C),head(C,B).} \\ \quad\ \mathtt{last(A,B):-\ tail(A,C),tail(C,D),head(D,B).}\ \} \\ \mathtt{h}_8 = \{\, \mathtt{last(A,B):-\ reverse(A,C),tail(C,D),head(D,B).} \\ \quad\ \mathtt{last(A,B):-\ tail(A,C),reverse(C,D),head(D,B).}\ \} \end{array} \right\}$$

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

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

In the generate stage, the learner generates a hypothesis:

$$\mathtt{h}_1 = \{\, \mathtt{last(A,B):-\ head(A,B).}\ \}$$

In the test stage, the learner tests $\mathtt{h}_1$ 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 $\mathtt{h}_1$ ($\mathtt{h}_2$ and $\mathtt{h}_3$) from the hypothesis space. The hypothesis space is now:

$$\mathscr{H}_2 = \left\{ \begin{array}{l} \mathtt{h}_4 = \{\, \mathtt{last(A,B):-\ tail(A,C),head(C,B).}\ \} \\ \mathtt{h}_5 = \{\, \mathtt{last(A,B):-\ reverse(A,C),head(C,B).}\ \} \\ \mathtt{h}_6 = \{\, \mathtt{last(A,B):-\ tail(A,C),head(C,B).} \\ \quad\ \mathtt{last(A,B):-\ reverse(A,C),head(C,B).}\ \} \\ \mathtt{h}_7 = \{\, \mathtt{last(A,B):-\ tail(A,C),head(C,B).} \\ \quad\ \mathtt{last(A,B):-\ tail(A,C),tail(C,D),head(D,B).}\ \} \\ \mathtt{h}_8 = \{\, \mathtt{last(A,B):-\ reverse(A,C),tail(C,D),head(D,B).} \\ \quad\ \mathtt{last(A,B):-\ tail(A,C),reverse(C,D),head(D,B).}\ \} \end{array} \right\}$$

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

$$h_4 = \{\texttt{last(A,B):- tail(A,C),head(C,B).}\}$$

The learner tests h_4 against the examples and finds that it fails because it entails the negative example `last([e,m,m,a],m)` and is therefore too *general*. The learner learns constraints to prune generalisations of h_4 (h_6 and h_7) from the hypothesis space. The hypothesis space is now:

$$\mathcal{H}_3 = \left\{ \begin{array}{l} h_5 = \{\texttt{last(A,B):- reverse(A,C),head(C,B).}\} \\ h_8 = \left\{ \begin{array}{l} \texttt{last(A,B):- reverse(A,C),tail(C,D),head(D,B).} \\ \texttt{last(A,B):- tail(A,C),reverse(C,D),head(D,B).} \end{array} \right\} \end{array} \right\}$$

The learner generates another hypothesis (`h5`), tests it against the examples, finds that it does not fail, and returns it.

There are two key ideas to our approach. Rather than refine a clause (Quinlan, 1990; Muggleton, 1995; Raedt and Bruynooghe, 1993; Blockeel and 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 first key idea is to refine the *hypothesis space* through learned *hypothesis constraints*. In other words, our key idea is to continually build a set of meta-constraints to constrain the hypothesis space. 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. Our second key idea is to decompose the ILP problem into entirely separate tasks: generate, test, and constrain. This idea allows for flexibility in how to implement our idea. Moreover, decomposing the problem allows for greater scalability with respect to the problem size (particularly the domain size and the number of training examples). In other words, decomposing the problem alleviates the combinatorial explosion problem faced by approaches that frame the ILP problem as a single SAT problem (Corapi et al., 2011; Law et al., 2014; Kaminski et al., 2018; Evans and Grefenstette, 2018; Evans et al., 2019).

We implement our idea in Popper[1], a new ILP system which combines answer set programming (ASP) (Gebser et al., 2012) and Prolog. In the generate stage, Popper uses ASP to declaratively define, constrain, and search the hypothesis space. The idea is to define an ASP problem where an answer set (a model) corresponds to a definite program. By later learning hypothesis constraints, we eliminate answer sets and thus prune the hypothesis space. Importantly, this stage ignores the examples and BK so that the search is focused on finding a constraint satisfying hypothesis. 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

[1] Popper is named after *Karl Poppper*, 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.

for using Prolog in this stage is to learn programs that use lists, numbers, and infinite 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 constraint 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 Progol (Muggleton, 1995), a classical ILP system, and Metagol (Cropper and Muggleton, 2016), ILASP[2] (Law et al., 2014), and ∂ILP (Evans and Grefenstette, 2018), three state-of-the-art ILP systems based on Prolog, ASP, and neural networks respectively. Compared to Progol, Popper can learn optimal and recursive programs. Compared to Metagol, Popper does not need metarules (Cropper and Tourret, 2019), so can learn programs with any arity predicates. Compared to ILASP and ∂ILP, Popper supports large and infinite domains. Compared to all the systems, Popper supports hypothesis constraints[3], such as disallowing the co-occurrence of predicate symbols in a program or disallowing recursive hypotheses that do not contain base cases.

	Progol	**Metagol**	**ILASP**	**∂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**
Infinite domains	**Yes**	**Yes**	No	No	**Yes**
Hypothesis constraints	No	No	No	No	**Yes**

Table 1: A simplified comparison of ILP systems. Note that Progol, Metagol, and Popper also induce Datalog programs, since Datalog is a subset of definite programs. Progol can learn recursive programs but struggles because it requires examples of both the base and inductive cases. Metagol supports *automatic* predicate invention, whereas ILASP and ∂ILP support *prescriptive* predicate invention (Cropper et al., 2019a), where the arity and argument types of an invented predicate must be specified by the given language bias.

Overall our specific contributions in this paper are:

- We define our problem setting, introduce our simple language bias called *predicate declarations*, introduce *hypothesis constraints*, calculate the size of the hypothesis space, define hypothesis *generalisations* and *specialisations*, and introduce the idea of learning from failures (Section 3).
- We introduce Popper, an ILP system that learns definite programs (Section 4). Popper uses ASP to declaratively define, constrain, and search the hypothesis space and Prolog to test hypotheses. Popper support types, learning optimal (textually minimal) solutions, learning recursive programs, reasoning about lists and infinite domains,

[2] There are many versions of ILASP. Unless otherwise stated, any reference to ILASP is to the set of ILASP systems.

[3] Law et al. (2018) also uses the term *hypothesis constraints* to describe the ILP system ILASP3. However, the authors replace that term with *hypothesis schemas* in the main reference for ILASP3 (Law, 2018). As we explain in Section 2.8, our notion of a hypothesis constraint is different to a hypothesis schema.

and the novel feature of hypothesis constraints. We show that Popper is sound and complete with respect to *optimal solutions* (Theorem 1).
– We experimentally show (Section 5) on three diverse domains (number theory problems, robot strategies, and list transformations) that (1) constraints drastically reduce the hypothesis space, (2) Popper can substantially outperform state-of-the-art ILP systems Metagol, ILASP, and FastLAS (Law et al., 2020), both in terms of predictive accuracies and learning times, (3) Popper scales well with respect to domain size, the number of training examples, and the size of the training examples, and (4) Popper is reasonably robust to its parameters.

2 Related work

2.1 Program synthesis

The goal of program synthesis is to automatically generate a computer program from a specification. Program synthesis from examples (Summers, 1977; Shapiro, 1983) interests researchers from many areas of computer science, notably machine learning (ML) and programming languages (PL). The major[4] 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 (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 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 SAT and SMT solvers. Neo inherently requires SMT specifications for domain specific background functions and predicates (i.e. background knowledge). For instance, the specification for *head*, taking an $input$ list and returning an $output$ list, is the formula $input.size \geq 1 \wedge output.size = 1 \wedge output.max \leq input.max$. Our approach does not need such definitions for the BK. We only need to evaluate hypotheses to determine their truth or falsity with respect to examples. 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. For instance, the largest number of candidate rules considered in the ProSynth experiments is 1000. By contrast, in our simplest experiment (Section 5.1), the hypothesis space contains approximately 10^{13} rules. ProSynth

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

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 program synthesis, including neural approaches (Balog et al., 2017; Ellis et al., 2018,0). We focus on inductive logic programming (ILP) (Muggleton, 1991). As with other forms of ML, given positive and negative examples, the goal of an ILP system is to learn a hypothesis which correctly explains as many positive and as few negative examples as possible. However, whereas most forms of ML represent data (examples and hypotheses) as tables (i.e. vectors), ILP represents data as logic programs. Moreover, whereas most forms of ML learn *functions*, ILP learns *relations*.

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 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 very 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. Progol, Aleph, TILDE, and XHAIL (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 ILP approaches, especially those that encode the ILP problem as a SAT problem, 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 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 Raedt, 1998; Srinivasan, 2001; Ray, 2009; Corapi et al., 2010,0; 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, what their arguments types are, 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* (Section 3), where a user needs only state whether a predicate symbol may appear in the head or/and body of a clause, similar to determinations in Aleph (Srinivasan, 2001). In our approach, a user can additionally provide other language biases, such as type information, as *hypothesis constraints* (Section 2.8).

Metarules (Cropper and Tourret, 2019) are another popular syntactic bias used by many program synthesis approaches (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., 2019b; Cropper and Muggleton, 2016) and, to an extent[5], ∂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 `last(A,B):- reverse(A,C),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, 2019). A user must either (1) provide a set of metarules, or (2) use a set of metarules restricted to a certain fragment of logic, e.g. dyadic Datalog (Cropper and Tourret, 2019). 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.

2.6 SAT approaches and infinite domains

An increasingly popular ILP approach is to encode the ILP problem as a SAT problem (Corapi et al., 2011; Athakravi et al., 2013; Law et al., 2014; Kaminski et al., 2018; Evans and Grefenstette, 2018; Evans et al., 2019).

Datalog is the target language of many ILP systems (Muggleton et al., 2014,0; Kaminski et al., 2018; Evans and Grefenstette, 2018; Evans et al., 2019). One motivation for learning Datalog, rather than Prolog, programs is to allow the ILP problem to be encoded as a SAT problem, particularly to leverage recent developments in SAT and SMT. This encoding is possible because a Datalog query is guaranteed to terminate – although this termination guarantee comes at the expense of not being a Turing-complete language.

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

A major limitation with these approaches is that they mostly encode the ILP problem as a single (often very large) SAT problem and thus struggle to scale to large problems.

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). Like Datalog, ASP is a truly declarative language. However, compared to Datalog, ASP is more expressive, allowing, for instance, aggregates, a form of disjunction in the head of a clause, and hard and weak constraints. Most ASP solvers only work on ground programs (Gebser et al., 2014)[6]. Therefore, a major limitation of 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. This grounding problem also implies that such systems do not support infinite domains. Difficulty handling large (or infinite) domains is not specific to ASP, and applies to other pure SAT-based approaches, even those based on neural networks, such as ∂ILP, which only works on BK formed of a finite set of ground atoms. To overcome this limitation, Popper combines ASP and Prolog. Popper uses ASP to generate definite programs, which allows it to reason about large and infinite domains, such as reasoning about lists and numbers.

2.7 Generate, test, and constrain

A key idea of our approach is to reason about the *hypothesis space*. Rather than refine a clause (Quinlan, 1990; Muggleton, 1995; Raedt and Bruynooghe, 1993; Blockeel and Raedt, 1998; Srinivasan, 2001; Ahlgren and Yuen, 2013), or a hypothesis (Shapiro, 1983; Bratko, 1999; Athakravi et al., 2013; Cropper and Muggleton, 2016), we refine the *hypothesis space* through learned *hypothesis constraints*. In other words, our key idea is to continually build a set of meta-constraints to constrain the hypothesis space. 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) also learns definite programs using SAT solvers and 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 case (in that order) of a recursive program. Moreover, for the same reason, Atom struggles to learn optimal solutions. By contrast, Popper imposes no such conditions because it learns programs rather than individual clauses.

The ILASP systems (Law et al., 2014,0,0), notably ILASP3 (Law, 2018), also follow a generate, test, and constrain loop. We focus on ILASP3, ILASP3 is a pure ASP-based ILP system. ILASP3 takes as input the full hypothesis space of ground clauses defined by given mode declarations. Each clause is given a unique id. The ILASP3 task is to find a subset of the clauses which covers as many positive and as few negative examples as possible. ILASP3 also tests hypotheses to generate constraints. If a hypothesis is not an optimal solution, ILASP3 translates an example into a set of *coverage constraints* over the

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

hypothesis space. We refer the reader to the work of Law (2018) for a detailed description, but, at a very high-level, a coverage constraint states that specific clauses must or must not be in a hypothesis (remember that ILASP3 precomputes the hypothesis space and assigns each clause a unique identifier).

Popper is similar to ILASP3 in that it follows a generate, test, and constrain loop. However, Popper differs from ILASP3 in several ways. ILASP3 learns unstratified ASP programs, including programs with normal rules, choice rules, and both hard and weak constraints. By contrast, Popper learns definite programs, typically described as Prolog programs, including programs with functions symbols, real numbers, and infinite domains. ILASP3 requires the full hypothesis space of pre-generated clauses as input. By contrast, Popper never fully constructs the hypothesis space, which allows it to scale better to larger programs (Section 5). If a hypothesis is non-optimal, ILASP3 finds a *relevant* example which it translates into a set of *coverage constraints* over the hypothesis space. By contrast, in our approach, when a hypothesis fails, we translate the *hypothesis* into a set of *hypothesis constraints*. Our hypothesis constraints are different because they do not reason about specific clauses (because we do not precompute the hypothesis space), but instead reason about the structure of hypotheses, i.e. are meta-constraints. Finally, ILASP3 is based entirely on ASP and the generate, test, and constrain stages are closely aligned. By contrast, Popper completely separates the generate, test, and constrain stages, where the generate stage ignores the examples and BK to alleviate the inherent grounding problem faced by ILASP3, which limits it to small domains (which we experimentally show in Section 5).

FastLAS (Law et al., 2020) builds on ILASP. The key difference is that FastLAS does not take the full hypothesis space as input. Instead it uses something similar to bottom clause construction (Muggleton, 1995) to find a subset of the hypothesis space. FastLAS does not, however, support recursion.

The general generate, test, and constrain approach can be traced back to Shapiro's seminal program synthesis work on the model inference system (MIS) (Shapiro, 1983), which, like our approach, was heavily inspired by Karl Popper's idea of falsification (Popper, 2005). MIS is a top-down, incremental, and interactive ILP approach which specialises and generalises a theory until it covers all of the positive and one of the negative examples. However, whereas MIS refines a hypothesis, by either deleting incorrect clauses or specialising clauses, our approach works at the meta-level, and refines the hypothesis space through learned hypothesis constraints.

2.8 Hypothesis constraints

Constraints are fundamental to our idea. Many ILP systems allow a user to constrain the hypothesis space though clause constraints (Muggleton, 1995; Srinivasan, 2001; Blockeel and 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*)[7], 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

[7] The term *hypothesis constraint* is also used by Srinivasan and Kothari (2005) and Costa et al. (2003) as an optional set of constraints on acceptable hypotheses, but without any further explanation.

reasons at the meta-level, this restriction is trivial to express:

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

We introduce this meta-level encoding in Section 4, but the 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 argue that 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.

3 Problem setting

We now define our problem setting, introduce our simple language bias called *predicate declarations*, introduce *hypothesis constraints*, calculate the size of the hypothesis space, define hypothesis *generalisations* and *specialisations*, and introduce our idea of learning from failures.

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. 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 (1) it contains no function symbols, and (2) 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 $v_1, \ldots, v_n$ in a clause with terms $t_1, \ldots, t_n$ is a *substitution* and is denoted as $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$. 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 (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 (Section 2.5). Several forms of language bias exist, such as mode declarations (Muggleton, 1995), grammars (Cohen, 1994) and metarules (Cropper and Tourret, 2019). We use a simple language

bias which we call *predicate declarations*, which are similar to Aleph's *determinations* (Srinivasan, 2001). 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.

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 $C = h \leftarrow b_1, b_2, \ldots, b_n$ be a definite clause. Then C is *declaration consistent* with D if and only if:

- h is an atom of the form $p(X_1, \ldots, X_n)$ and *head_pred(p,n)* is in D_h
- every b_i is a literal of the form $p(X_1, \ldots, X_n)$ 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:

$$(\{\textit{head\_pred(targ,2)}\}, \{\textit{body\_pred(head,2)}, \textit{body\_pred(tail,2)}\})$$

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:

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

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:

$$(\{\textit{head\_pred(targ,2)}\}, \{\textit{body\_pred(head,2)}, \textit{body\_pred(tail,2)}\})$$

Then two declaration consistent hypotheses are:

h_1 :
```
targ(A,B):- head(A,B)
```

h_2 :
```
targ(A,B):- head(A,B).
targ(A,B):- tail(A,C),head(C,B).
```

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 Section 4), we use a more general definition:

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

Example 4 In Section 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:

```
:- 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:

```
:- head_literal(_,p,2,_), 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 $\mathscr{L}$. A set of definite clauses H is *consistent* with C if, when written in $\mathscr{L}$, 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 $\mathscr{H}_{D,C}$ is the (possibly infinite) set of all declaration and constraint consistent hypotheses. We refer to any element in $\mathscr{H}_{D,C}$ as a *hypothesis*.

We define our problem input:

Definition 9 (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. In other words, our approach does not need both positive and negative examples, and can work with only positive or only negative examples. 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 Wolf, 1997):

Definition 10 (Hypothesis types) Let (B, D, C, E^+, E^-) be an input tuple and $H \in \mathscr{H}_{D,C}$ 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 *solution*, i.e. our problem output:

Definition 11 (Solution) Given an input tuple (B, D, C, E^+, E^-), a hypothesis $H \in \mathscr{H}_{D,C}$ 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 $H \in \mathscr{H}_{D,C}$ *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 $H \in \mathscr{H}_{D,C}$ is an *optimal solution* when two conditions hold:

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

In Section 4, we introduce Popper, which, given the problem input, is guaranteed to return an optimal solution (Theorem 1).

3.3 Hypothesis space

One of the main ideas of our learning from failures approach 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:*

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

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 chose 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 chose 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 chose k clauses to form a hypothesis. Therefore, there are $\sum_{j=1}^{n}\binom{|D_h|v^a \ \sum_{i=1}^{m}\binom{|D_b|v^a}{i}}{j}$ ways to define a hypothesis with at most n clauses.

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 $\texttt{C}_1$ and $\texttt{C}_2$ be the clauses:

```
C1 = f(A,B):- head(A,B)
C2 = f(X,Y):- head(X,Y),odd(Y).
```

Then $\texttt{C}_1$ subsumes $\texttt{C}_2$ because $\texttt{C}_1\theta \subseteq \texttt{C}_2$ 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 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 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:

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

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).

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 our learning from failures approach, 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 not prune solutions. We also define an *elimination* constraint, specific to learning non-recursive definite programs, 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
$\mathbf{N_{none}}$	H is consistent, i.e. H entails no negative example	$\forall e \in E^-,\ H \cup B \not\models e$
$\mathbf{N_{some}}$	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 $\mathbf{N_{none}}$, i.e. H is consistent. Then we cannot prune the hypothesis space.

Suppose the outcome is $\mathbf{N_{some}}$, 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:

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

Because h entails a negative example, it is too general, so we can prune generalisations of it, such as $\texttt{h}_1$ and $\texttt{h}_2$:

$$\texttt{h}_1 = \left\{\begin{array}{l}\texttt{last(A,B):-head(A,B).}\\ \texttt{last(A,B):-tail(A,C),head(C,B).}\end{array}\right\}$$
$$\texttt{h}_2 = \left\{\begin{array}{l}\texttt{last(A,B):-head(A,B).}\\ \texttt{last(A,B):-tail(A,C),head(C,B),head(A,B).}\end{array}\right\}$$

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 outcomes[8] of testing H against E^+:

Outcome	Description	Formula
$\mathbf{P_{all}}$	H is complete, i.e. H entails all positive examples	$\forall e \in E^+, H \cup B \models e$
$\mathbf{P_{some}}$	H is incomplete, i.e. H does entail all positive examples	$\exists e \in E^+, H \cup B \not\models e$
$\mathbf{P_{none}}$	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 $\mathbf{P_{all}}$, i.e. H is complete. Then we cannot prune the hypothesis space.

Suppose the outcome is $\mathbf{P_{some}}$, 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*:

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

[8] The outcomes are not mutually exclusive.

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

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

Because h entails the first example but not the second it is too specific. We can therefore prune specialisations of h, such as $\texttt{h}_1$ and $\texttt{h}_2$:

$$\begin{array}{l} \texttt{h}_1 = \left\{ \texttt{last(A,B):- head(A,B),empty(A).} \right\} \\ \texttt{h}_2 = \left\{ \texttt{last(A,B):- head(A,B),tail(A,C).} \right\} \end{array}$$

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, $H \in \mathscr{H}_{D,C}$ be an incomplete hypothesis, and $H' \in \mathscr{H}_{D,C}$ 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 $\mathbf{P}_{\mathbf{none}}$, i.e. H is totally incomplete. Then H is too specific so, as with $\mathbf{P}_{\mathbf{some}}$, 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 non-recursive hypothesis (we illustrate why recursion matters in a moment). In other words, if H does not entail *any* positive example, then no specialisation of H can appear in an optimal non-recursive solution. We can therefore prune non-recursive hypotheses that contain specialisations of H. We call such a constraint an *elimination constraint*:

Definition 21 (Elimination constraint) An elimination constraint only prunes non-recursive hypotheses that contain specialisations of a hypothesis from the hypothesis space.

Example 10 (Elimination constraint) Suppose we have the positive examples E^+ and the hypothesis h:

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

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

$$\begin{array}{l} \texttt{h}_1 = \left\{ \begin{array}{l} \texttt{last(A,B):-head(A,B).} \\ \texttt{last(A,B):-tail(A,C),head(C,B).} \end{array} \right\} \\ \texttt{h}_2 = \left\{ \begin{array}{l} \texttt{last(A,B):-head(A,B).} \\ \texttt{last(A,B):-tail(A,C),head(C,B),odd(B).} \end{array} \right\} \\ \texttt{h}_3 = \left\{ \begin{array}{l} \texttt{last(A,B):-head(A,B),even(B).} \\ \texttt{last(A,B):-tail(A,C),head(C,B),odd(B).} \end{array} \right\} \end{array}$$

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 11 (Elimination solution unsoundness) Suppose we have the positive examples E^+ and the hypothesis $\mathtt{h}_1$:

$$E^+ = \left\{ \begin{array}{l} \texttt{last([j,i,m],m)} \\ \texttt{last([a,l,i,c,e],e)} \end{array} \right\} \quad \mathtt{h}_1 = \left\{ \texttt{last(A,B):- head(A,B).} \right\}$$

Then an elimination constraint would prune the complete hypothesis $\mathtt{h}_2$:

$$\mathtt{h}_2 = \left\{ \begin{array}{l} \texttt{last(A,B):- head(A,B).} \\ \texttt{last(A,B):- reverse(A,C),head(C,B).} \end{array} \right\}$$

However, for non-recursive 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, $H_1 \in \mathscr{H}_{D,C}$ be a totally incomplete hypothesis, $H_2 \in \mathscr{H}_{D,C}$ be a complete hypothesis such that (i) $H_1 \subset H_2$ and (ii) no predicate of D_h occurs in the body of a clause in H_2, and $H_3 = H_2 \setminus H_1$. Then H_3 is complete.*

Proof By assumption, no predicate in D_h occurs in the body of a clause in B, H_2, 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.

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, $H_1 \in \mathscr{H}_{D,C}$ be a totally incomplete hypothesis, $H_2 \in \mathscr{H}_{D,C}$ be a hypothesis such that $H_1 \preceq H_2$, and $H_3 \in \mathscr{H}_{D,C}$ be a hypothesis such that $H_2 \subset H_3$ and no predicate in D_h is in the body of a clause of H_3. Then H_3 is not an optimal solution.*

Proof Assume that H_3 is an optimal solution. This assumption implies that (1) H_3 is a solution, and (2) there is no hypothesis $H_4 \in \mathscr{H}_{D,C}$ 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 (2) cannot hold, which contradicts the assumption and completes the proof.

This proof relies on a hypothesis H being (1) a definite program, and (2) non-recursive (i.e. no predicate in the body of a clause in H appears in the head of a clause in H). Condition (1) is clear because the proof relies on the monotonicity of definite programs. To illustrate condition (2), we give a counter-example to show why we can only use elimination constraints to prune non-recursive hypotheses.

Example 12 (Non-elimination for recursive hypotheses) Suppose we have the positive examples E^+ and the hypothesis h:

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

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

Then h is totally incomplete so there is no reason for h to appear in a non-recursive hypothesis. However, h can still appear in a recursive hypothesis, where the clauses depend on each other, such as $\texttt{h}_2$:

$$\texttt{h}_2 = \left\{ \begin{array}{l} \texttt{last(A,B):- head(A,B),tail(A,C),empty(C).} \\ \texttt{last(A,B):- tail(A,C),last(C,B).} \end{array} \right\}$$

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.

Outcome	$\mathbf{N_{none}}$	$\mathbf{N_{some}}$
$\mathbf{P_{all}}$	n/a	Generalisation
$\mathbf{P_{some}}$	Specialisation	Specialisation, Generalisation
$\mathbf{P_{none}}$	Specialisation, Elimination	Specialisation, Elimination, Generalisation

Table 2: The constraints we can learn from testing a hypothesis. The $\mathbf{P_{all}}$ and $\mathbf{N_{none}}$ outcomes denote that we have found a solution.

4 Popper

Popper is an implementation of our learning from failures idea[9]. Popper works in three separate stages: generate, test, and constrain, as described in Section 1. 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.

[9] Popper is only one implementation of our idea. The flexibility of our three staged approach allows for a variety of algorithms, which we intend to explore in future work.

Algorithm 1 Popper

```
def popper(e+, e−, bk, declarations, constraints, max_literals):
  num_literals = 1
  while num_literals ≤ max_literals:
    program = generate(declarations, constraints, num_literals)
    if program == 'space_exhausted':
      num_literals += 1
      continue
    outcome = test(e+, e−, bk, program)
    if outcome == ('all_positive', 'none_negative')
      return program
    constraints += learn_constraints(program, outcome)
  return {}
```

Answer set programming. Popper uses ASP to generate logic programs. We briefly introduce some ASP specific syntax and refer the reader to the excellent book by Gebser et al. (2012) for more information about ASP. A literal is either an atom `a` or its default negation `not a`. An ASP rule is of the form `h:- b1, ..., bn.` where `h` is an atom, each b_i is a literal, `h` is the *head*, and `b1, ..., bn`, is the *body*. A *constraint* is a rule without a head. A *choice rule* is of the form `l{h1, ..., hm}u :- b1, ..., bn.` where `l` and `u` are integers denoting lower and upper bounds. For instance, `2{ a,b,c }3` asserts that at least two of `a,b,c` need to be true. A *conditional literal* is of the form `l:l1,...,ln` and is replaced by the conjunction of all of `l` such that *condition* `l1,...,ln` is true. A *range* is shorthand syntax of the form *(start..end)* where *start* and *end* are integers. For instance, the range *p(1..3)* is syntactic sugar for *p(1). p(2). p(3)*. The *aggregate* `#count` calculates the number of elements of a set. For example, the expression `#count{X : knows(X,alice)} == N` counts how many unique values `X` hold for `knows(X,alice)` and checks that it is equal to `N`.

4.1 Generate

The generate step of Popper takes as input (1) predicate declarations, (2) hypothesis constraints, and (3) a bound on the total number of literals in a hypothesis and returns an answer set which represents a definite program, if one exists. There are also implicit input parameters that bound the number of unique variables, literals, and clauses allowed in a hypothesis. The idea is to define an ASP problem where an answer set (a model) corresponds to a definite program. 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 key 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 p, 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.

```
% 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.

4.1.1 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, Figure 3 shows constraints specifically for recursive programs, such as preventing 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:

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

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).

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

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

```
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.

4.1.2 Language bias constraints

A key feature of Popper is that it supports optional[10] 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 Figure 4. Through constraints, Popper also supports the standard notions of *recall* and *input/output*[11] 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, 2019), which ensures that each clause in a program is an instance of a metarule.

[10] In contrast to most ILP systems, the only bias that Popper needs is predicate declarations. Other biases, such as types or recall, are all optional.

[11] 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.

```
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_in_literal(Clause,P,Vars1,Var),var_in_literal(Clause,Q,Vars2,Var),
    var_pos(Var,Vars1,Pos1),var_pos(Var,Vars2,Pos2),
    type(P,Pos1,Type1),type(Q,Pos2,Type2),
    Type1 != Type2.
```

Fig. 4: Optional language bias constraints used by Popper.

4.1.3 Hypothesis constraints

As with many ILP systems (Muggleton, 1995; Srinivasan, 2001; Law et al., 2014), 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 whole program (a set of clauses) rather than a single clause. For instance, hypothesis constraints allow us to prune recursive programs that do not contain a base case clause (Figure 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 two 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.

4.2 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. In addition to evaluating a whole hypothesis, Popper also individually evaluates each non-recursive clause in a hypothesis. This extra check allows us to identify additional elimination constraints. If a hypothesis fails, then Popper identifies what type of failure has occurred and what constraints to generate (using the failures and constraints from Section 3.5).

4.3 Constrain

If a hypothesis fails, then, in the constrain stage, Popper generates ASP constraints to prune the hypothesis space, and thus constrain 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 Section 4.1). We describe the generalisation, specialisation, and elimination constraints that Popper uses, based on the definitions in Section 3.5. As a version of Popper without these constraints is considered in the experiments, 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.

4.3.1 Encoding atoms

Consider encoding the atom `f(A,B)`. An atom is either in the head or body of a clause. In our encoding, the atom is either represented as **`head_literal(Clause,f,2,(V0,V1))`** or as **`body_literal(Clause,f,2,(V0,V1))`**. The relevant clause is indicated by **`Clause`** and the **`2`** indicates the predicate's arity. Two functions below encode atoms into ASP literals. The function *encodeHead* encodes a head atom and *encodeBody* encodes a body atom. The first argument specifies the clause an atom belongs to. The second argument is the atom. A hypothesis variable is converted to a variable in our ASP encoding by the *encodeVar* function[12].

encodeHead($Clause$,`Pred(`$\texttt{Var}_0,\ldots,\texttt{Var}_k$`)`) :=
  `head_literal(`$Clause$**`,`**`Pred`**`,k+1,(`***encodeVar***`(`**$\texttt{Var}_0$**`),...,`***encodeVar***`(`**$\texttt{Var}_k$**`)))`**

encodeBody($Clause$,`Pred(`$\texttt{Var}_0,\ldots,\texttt{Var}_k$`)`) :=
  `body_literal(`$Clause$**`,`**`Pred`**`,k+1,(`***encodeVar***`(`**$\texttt{Var}_0$**`),...,`***encodeVar***`(`**$\texttt{Var}_k$**`)))`**

For instance calling *encodeHead*(**`Clause`**, `f(A,B)`) generates the ASP literal **`head_literal(Clause,f,2,(V0,V1))`** and calling *encodeBody*(**`Clause`**, `f(A,B)`) generates the ASP literal **`body_literal(Clause,f,2,(V0,V1))`**.

4.3.2 Encoding clauses

Using the encoding of atoms to ASP literals, we can encode clauses. Consider a clause `last(A,B):- reverse(A,C),head(C,B)`. Supposing **`C`**$_\texttt{i}$ identifies the clause, the following ASP literals capture where the atoms occur:

`head_literal(C`$_\texttt{i}$**`,last,2,(V0,V1)),`**
`body_literal(C`$_\texttt{i}$**`,reverse,2,(V0,V2)),body_literal(C`**$_\texttt{i}$**`,head,2,(V2,V1))`**

Note that ASP variables **`V0`**, **`V1`**, **`V2`** will be instantiated by 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`**, which represents the clause with all variables as `A`. To ensure that these variables remain distinct we need to impose **`V0!=V1`** and **`V0!=V2`** and **`V1!=V2`**. The

[12] While not reflected in the examples, *encodeVar* and *vars* automatically ensure that variables from distinct clauses get distinct names.

function *encodeClause* implements both the straightforward translation and the variable distinctness assertion:

$$\begin{aligned}&\mathit{encodeClause}(\mathit{Clause}, (\texttt{head:-body}_1,\ldots,\texttt{body}_m)) :=\\ &\quad \mathit{encodeHead}(\mathit{Clause},\texttt{head}),\mathit{encodeBody}(\mathit{Clause},\texttt{body}_1),\ldots,\\ &\quad \mathit{encodeBody}(\mathit{Clause},\texttt{body}_m),\\ &\quad \mathit{assertDistinct}(\mathit{vars}(\texttt{head}) \cup \mathit{vars}(\texttt{body}_1) \cup \ldots \cup \mathit{vars}(\texttt{body}_m))\end{aligned}$$

An encoding by *encodeClause* only asserts what occurs in the clause. It does not state that other literals do not occur in the clause. For example, the above ASP literals would also be true of the clause `last(A,B):- reverse(A,C),head(C,B),tail(C,A).`

In our encoding, the ASP literal **`clause_size(C_i,m)`** is only true when clause **`C_i`** has exactly **`m`** body literals. The function *encodeSizedClause* uses this literal to assert that, beyond the **`m`** body literals already asserted, there can be no other body literals:

$$\begin{aligned}&\mathit{encodeSizedClause}(\mathit{Clause}, (\texttt{head:-body}_1,\ldots,\texttt{body}_m)) :=\\ &\quad \mathit{encodeClause}(\mathit{Clause}, (\texttt{head:-body}_1,\ldots,\texttt{body}_m)),\textbf{clause\_size(}\mathit{Clause}\textbf{,m)}\end{aligned}$$

For instance, *encodeSizedClause*(**`C_i`**,`(last(A,B):- reverse(A,C),head(C,B)))` imposes that clause **`C_i`** must correspond *exactly* to the given clause:

```
head_literal(C_i,last,2,(V0,V1)),
body_literal(C_i,reverse,2,(V0,V2)),body_literal(C_i,head,2,(V2,V1))
V0!=V1,V0!=V2,V1!=V2,clause_size(C_i,2)
```

With the clause encoding functions defined, we can now use them to define our constraints.

4.3.3 Generalisation constraints

Given a hypothesis H, by Definition 17, any hypothesis that includes all of H's clauses exactly, i.e. not specialised, is a generalisation of H. We use this fact to define function *generalisationConstraint*, which converts a set of clauses into an ASP encoded generalisation constraint (Definition 19). We use *encodeSizedClause* to impose that a clause is not specialised. Each clause gets its own ASP variable **`C_i`**, meaning the clauses can occur in any order.

$$\begin{aligned}&\mathit{generalisationConstraint}(\{\texttt{Clause}_0,\ldots,\texttt{Clause}_{n-1}\}) :=\\ &\quad \texttt{:-}\ \mathit{encodeSizedClause}(\textbf{C}_\textbf{0},\texttt{Clause}_0),\ldots,\mathit{encodeSizedClause}(\textbf{C}_\textbf{n-1},\texttt{Clause}_{n-1}).\end{aligned}$$

Figure 5 illustrates a generalisation constraint derived by *generalisationConstraint*.

`h = { last(A,B):- head(A,B). }`

```
:-
  head_literal(C0,last,2,(C0V0,C0V1)),
  body_literal(C0,head,2,(C0V0,C0V1)),
  C0V0 != C0V1,clause_size(C0,1).
```

Fig. 5: The ASP encoded generalisation constraint for the hypothesis `h`.

4.3.4 Specialisation constraints

Given a hypothesis H, by Definition 18, any hypothesis which has every clause of H occur, where each clause 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). We use that *encodeClause* allows additional literals to be added to a provided clause. The literal **`not clause(n)`** ensures no additional clause is added to the **`n`** distinct clauses of the provided hypothesis.

$$\begin{aligned}&\textit{specialisationConstraint}(\{\texttt{Clause}_0,\ldots,\texttt{Clause}_{n-1}\}) :=\\ &\quad\texttt{:-}\ \textit{encodeClause}(\texttt{Cl}_\texttt{0},\texttt{Clause}_0),\ldots,\textit{encodeClause}(\texttt{Cl}_{\texttt{n-1}},\texttt{Clause}_{n-1}),\\ &\qquad\textit{assertDistinct}(\{\texttt{Cl}_\texttt{0},\ldots,\texttt{Cl}_{\texttt{n-1}}\}),\texttt{not clause(n).}\end{aligned}$$

We illustrate why asserting that specialised clauses are distinct is necessary. Consider the hypotheses $\texttt{h}_1$ and $\texttt{h}_2$:

$$\texttt{h}_1 = \left\{\begin{array}{l}\texttt{last(A,B):- head(A,B).}\\ \texttt{last(A,B):- sumlist(A,B).}\end{array}\right\}\ \texttt{h}_2 = \left\{\begin{array}{l}\texttt{last(A,B):- head(A,B),sumlist(A,B).}\\ \texttt{last(A,B):- member(A,B).}\end{array}\right\}$$

The first clause of $\texttt{h}_2$ specialises both clauses in $\texttt{h}_1$, yet $\texttt{h}_2$ is not a specialisation of $\texttt{h}_1$. 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*.

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

```
:-
  head_literal(C0,rev,2,(C0V0,C0V1)),
  body_literal(C0,head,2,(C0V0,C0V1)),
  head_literal(C1,rev,2,(C1V0,C1V1)),
  body_literal(C1,tail,2,(C1V0,C1V2)),
  body_literal(C1,head,2,(C1V2,C1V1)),
  C0V0 != C0V1,C1V0 != C1V1,
  C1V0 != C1V2,C1V1 != C1V2,
  C0 != C1,not clause(2).
```

Fig. 6: The ASP specialisation constraint for the hypothesis h.

4.3.5 Elimination constraints

By Proposition 5, given a totally incomplete hypothesis H, any non-recursive hypothesis which includes all of H's clauses, where each clause may be specialised, cannot be an optimal solution. The function *eliminationConstraint* uses this fact to derive an ASP encoded elimination constraint (Definition 21). As in *specialisationConstraint*, *encodeClause* is used to allow additional literals in clauses, ensuring that provided clauses are included or 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 non-recursive.

eliminationConstraint({Clause$_0$,...,Clause$_{n-1}$}) :=
    :- *encodeClause*(**Cl$_0$**,Clause$_0$),...,*encodeClause*(**Cl$_{n-1}$**,Clause$_{n-1}$),
       not recursive.

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

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

```
:-
  head_literal(C0,last,2,(C0V0,C0V1)),
  body_literal(C0,tail,2,(C0V0,C0V2)),
  body_literal(C0,head,2,(C0V2,C1V1)),
  C0V0 != C0V1,C0V0 != C0V2,C0V1 != C0V2,
  not recursive.
```

Fig. 7: The ASP elimination constraint for the hypothesis h.

4.3.6 Banish constraints

In the experiments section, we compare Popper against 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:

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

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 *encodeSizedClause*. The literal **not clause(n)** asserts that there are no more clauses than the original number.

banishConstraint({Clause$_0$,...,Clause$_{n-1}$}) :=
    :- *encodeSizedClause*(**C$_0$**,Clause$_0$),...,*encodeSizedClause*(**C$_{n-1}$**,Clause$_{n-1}$),
       not clause(n).

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

4.4 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 steps. The idea of multi-shot solving is that state about the search

h = { f(A):- head(A,B),one(B).
      f(A):- tail(A,B),tail(B,C),empty(C). }

```
:-
  head_literal(C0,f,1,(0,)),
  body_literal(C0,head,2,(0,1)),
  body_literal(C0,one,1,(1,)),
  head_literal(C1,f,1,(0,)),
  body_literal(C1,tail,2,(0,1)),
  body_literal(C1,tail,2,(1,2)),
  body_literal(C1,empty,1,(2,)),
  clause_size(C0,2),clause_size(C1,3),
  not clause(2).
```

Fig. 8: The ASP banish constraint for the hypothesis h.

space for an ASP program can be saved to help the search for any modifications of that program. The essence of the multi-shot cycle is that a ground program is given to a ASP solver, yielding an answer set, which 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 Section 4.1. Popper starts a Clingo instance and asks it to solve this program, which grounds it and then calls the ASP solver, which returns an answer set (if the problem is satisfiable). Popper converts the answer set to a definite program and tests it against the examples. If a hypothesis fails, Popper generates ASP constraints using the functions in Section 4.3 and adds them to the running Clingo instance, which grounds the constraints and adds the new (propositional) rules to the running solver. 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 (1) Popper finds an optimal solution, or (2) there are no more hypotheses to test.

4.5 Correctness

We now show the correctness of Popper. We first show that Popper's base encoding (Figure 2) can generate every declaration consistent hypothesis (Definition 4):

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

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 $p_h(H_1,\ldots,H_n)$ of C as a choice literal `head_literal(i,`p_h`,n,(`H_1`,...,`H_n`))` guarded by the condition `head_pred(`p_h`,n)` $\in D_h$, which clearly holds. Our encoding represents a body literal $p_b(B_1,\ldots,B_m)$ of C as a choice literal `body_literal(`i`,`p_b`,`m`,(`B_1`,...,`B_m`))` guarded by the condition `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.

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.

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 Section 4.3, 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 Section 4.3, 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 Section 4.3, Popper uses an elimination constraint to prune all non-recursive 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.

We 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.

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.

5 Experiments

We now evaluate our learning from failures idea. A key idea of our approach is to learn 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 in practice whether, and if so by how much, constraints will improve learning performance because Popper needs to (1) analyse failed hypotheses, (2) generate constraints from them, and (3) 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*.

As mentioned in Section 2, a major limitation of existing pure ASP-based ILP approaches is that they struggle to handle large domains and cannot support infinite domains (Corapi et al., 2011; Athakravi et al., 2013; Law et al., 2014; Kaminski et al., 2018; Evans et al., 2019). To address this limitation, our approach decomposes the ILP problem into separate hypothesis generation and testing stages. In our implementation, Popper uses ASP to generate programs and then uses Prolog to test programs against the examples. We therefore claim that Popper can outperform pure ASP-based ILP systems on large domains (we do not consider infinite domains because pure ASP-based ILP systems need a finite grounding). In addition, because we learn constraints to avoid repeated search, we claim that Popper can outperform existing pure Prolog-based ILP systems. Our experiments therefore aim to answer the question:

Q2 Can Popper outperform state-of-the-art ILP systems?

To answer this question, we compare Popper against Metagol (version 2.3.0) (Cropper and Muggleton, 2016), ILASP2 (Law et al., 2016), ILASP3 (Law, 2018), and FastLAS (Law et al., 2020).

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:

Q3 How well does Popper scale?

To answer this question, we evaluate Popper on several problems where we vary (1) the size of the target program, (2) the number of predicate declarations, (3) the number of constants in the problem, (4) the number of unique variables in a clause, (5) the maximum number of literals in a clause, and (6) the maximum number of clauses allowed in a hypothesis.

5.1 Primorials

The purpose of this first experiment is to evaluate how well Popper scales with respect to the optimal solution size (i.e. the total number of literals in the optimal solution). We therefore need a problem where we can control the optimal solution size. We consider a number theory problem. Let p_k denote the kth prime number. Then the *primorial* $p_n\#$ is defined as the product of the first n primes:

$$p_n\# \equiv \prod_{k=1}^{n} p_k$$

For instance, $p_5\#$ is the product of the first 5 primes:

$$p_5\# = 2 \times 3 \times 5 \times 7 \times 11 = 2310$$

The goal of this experiment is to classify primorial numbers. We vary the solution size by varying the primorial number $p_n\#$. The primorial $p_n\#$ requires n body literals. For instance, for $p_2\#$, the solution is:

```
primorial2(A):- div2(A),div3(A).
```

For $p_5\#$, the solution is:

```
primorial5(A):- div2(A),div3(A),div5(A),div7(A),div11(A).
```

5.1.1 Materials

To evaluate how well Popper scales given more predicate declarations, we compare two sets of BK (*small* and *big*). In the first set (*small*), we provide as BK a monadic predicate $divisible_i$ for each prime number i in $\{1, 2, \ldots, 100\}$, which holds when a number is evenly divisible by i. In the second set (*big*), we augment the small dataset with dummy monadic predicates which always evaluate to false. For simplicity, we use the predicate $dummy_i$ for each non-prime number i in $\{1, 2, \ldots, 100\}$. Note that this problem representation is not necessarily the most compact. Indeed, we purposely designed the representation so we can vary the optimal solution size to evaluate how well the systems scale. To be clear, the only variable in the experiment (besides the ILP system) is the optimal solution size, which we progressively increase to evaluate how well the systems scale.

We compare Popper, Enumerate, Metagol, ILASP, and FastLAS. To compare the systems, we try to use settings so that each system considers approximately the same hypothesis space.

Popper and Enumerate settings We set Popper and Enumerate to us at most 1 unique variable, at most 10 body literals, and at most 1 clause.

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

```
P(A):- Q(A).
P(A):- Q(A),R(A).
```

These metarules match the Popper settings in that only one variable is used.

ILASP2 and ILASP3 settings We run both ILASP2 and ILASP3 with the same settings[13], so we simply refer to both as ILASP. We run ILASP with the '-no-constraints' and '–no-aggregates' flags. We additionally ran ILASP3 with the 'disable implication' and 'disable propagation' flags. We tell ILASP that each BK relation is positive, which prevents it from generating body literals using negation. We set ILASP to use at most 1 unique variable and at most 2 body literals ('-ml=2' and '–max-rule-length=3'). When we tried to use at most 3 and 4 body literals it took ILASP 42 seconds (3 body literals) and 41 minutes (4 body literals) to generate the hypothesis space, i.e. to generate the SAT problem. This bound implies that the largest primorial number learnable by ILASP is $p_2\#$. ILASP does not support infinite domains so requires a bound on the number of integers. We found that it took Clingo 2 seconds, 48 seconds, and 8 minutes to ground the BK for the bounds for $p_7\#$, $p_8\#$, and $p_9\#$ respectively. We therefore set the maximum integer bound to $p_7\#+1$. This bound implies that largest primorial number learnable by ILASP is $p_7\#$ (ignoring the maximum literal bound).

FastLAS settings We set FastLAS to run identically to ILASP, except we do not enforce a maximum body literal size because FastLAS does not need such a bound. Note that when we set the maximum integer bound to $p_8\#+1$, FastLAS could not find any solutions in the allocated time.

5.1.2 Methods

For each n in $\{1,2,\ldots,10\}$, we generate the single positive example corresponding to $p_n\#$. We uniformly sample 20 negative examples from the set $\{2,\ldots,p_n\}$. We measure learning time as the time to learn a solution. We enforce a timeout of 2 minutes per task. We repeat each experiment 10 times and plot the standard error.

5.1.3 Results

Figure 9 shows the results. Popper clearly outperforms Enumerate (the unconstrained approach) on both datasets. On the small dataset, Enumerate can only learn a program for the second primorial number, i.e. a program with two body literals. On the big dataset Enumerate can only learn a program for the first primorial number, i.e. a program with one body literal. By contrast, on both datasets, Popper can learn a program for the 10th primorial number, i.e. a program with 10 body literals. This result strongly suggests that the answer to **Q1** is yes, constraints can drastically improve learning performance.

Why does Popper perform much better than Enumerate? Enumerate tests every hypothesis, i.e. every combination of literals. By contrast, Popper learns constraints from failed hypotheses to prune the hypothesis space, i.e. to remove certain combinations of literals. For instance, consider learning a program for $p_3\# = 2\times3\times5 = 30$. Below shows a tiny subset ($\mathscr{H}$) of the hypothesis space for this problem (the full hypothesis for the big BK problem contains approximately 10^{13} hypotheses). When Popper tests $\mathtt{h}_1$, it fails because it is too specific, i.e. `div53(30)` fails. Popper therefore generates a constraint to remove specialisations of $\mathtt{h}_1$ ($\mathtt{h}_1$ - $\mathtt{h}_8$) from the hypothesis space. From testing this single

[13] We consulted the ILASP authors for suggestions on which settings to run ILASP2 and ILASP3 with.

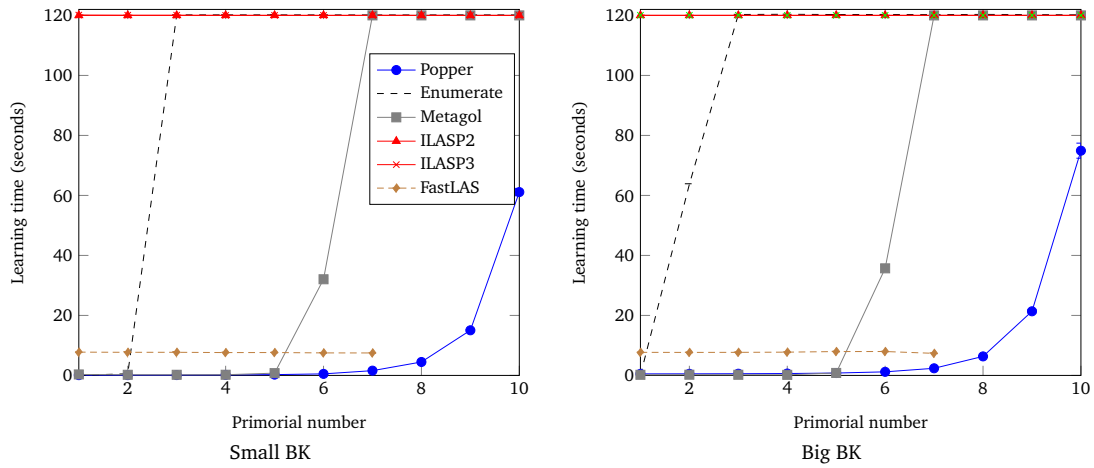


Fig. 9: Primorials experimental results when varying the primorial number, which corresponds to the size of the optimal solution. Note that FastLAS cannot solve any problems for $p_8\#$, $p_9\#$, and $p_{10}\#$ because of a maximum integer bound.

hypothesis, Popper drastically reduces the size of the hypothesis space.

$$\mathcal{H} = \left\{ \begin{array}{l} \texttt{h}_1 \texttt{ = primorial3(A):- div53(A).} \\ \texttt{h}_2 \texttt{ = primorial3(A):- div53(A),div2(A).} \\ \texttt{h}_3 \texttt{ = primorial3(A):- div53(A),div3(A).} \\ \texttt{h}_4 \texttt{ = primorial3(A):- div53(A),div5(A).} \\ \texttt{h}_5 \texttt{ = primorial3(A):- div53(A),div2(A),div3(A).} \\ \texttt{h}_6 \texttt{ = primorial3(A):- div53(A),div2(A),div5(A).} \\ \texttt{h}_7 \texttt{ = primorial3(A):- div53(A),div3(A),div5(A).} \\ \texttt{h}_8 \texttt{ = primorial3(A):- div53(A),div2(A),div3(A),div5(A)} \\ \texttt{h}_9 \texttt{ = primorial3(A):- div2(A),div3(A),div5(A)} \end{array} \right\}$$

Popper outperforms Metagol. The highest primorial number for which Metagol can learn a solution is $p_6\#$, which takes 35 seconds to learn. By contrast it takes Popper 2 seconds to learn the solution for $p_6\#$. We think the performance difference is because of Metagol's inefficient search. Metagol performs iterative deepening over the number of clauses allowed in a solution (Muggleton et al., 2015). However, if a clause or 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 ILASP2, ILASP3, and FastLAS. ILASP2 and ILASP3 cannot solve any problem, even for $p_1\#$, because they both pre-compute the hypothesis space. FastLAS performs much better than both. For $p_7\#$ it takes FastLAS 8 seconds to learn a solution. By contrast it takes Popper 2 seconds. FastLAS cannot learn solutions for $p_8\#$, $p_9\#$, or $p_{10}\#$ because of the maximum integer bound. Note that when given a larger bound, FastLAS could not learn a solution for any primorial number.

Overall, the results from this experiment suggest that the answers to questions **Q1** and **Q2** are both yes, and that the answer to **Q3** is that Popper scales better than state-of-the-art ILP systems with respect to the optimal solution size.

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 cannot move upwards any more*. 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.

5.2.1 Materials

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. We provide four dyadic relations as BK: *move_right*, *move_left*, *move_up*, and *move_down*, which change the state, e.g. *move_right((2,2),(3,2))*. Again, note that this problem representation is not necessarily the most compact and may not be the best representation for certain systems.

We compare Popper, Enumerate, Metagol, ILASP2, and ILASP3. We do not use FastLAS because it does not support recursion. To fairly compare the systems, we again try to use settings so that each system considers approximately the same hypothesis space.

Popper settings We allow Popper and Enumerate to use at most 3 unique variables, at most 2 body literals, and at most 2 clauses. 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.

Metagol settings We provide Metagol with the metarules in Figure 10. These metarules constitute an almost[14] complete set of metarules for a singleton-free fragment of monadic and dyadic Datalog (Cropper and Tourret, 2019).

ILASP2 and ILASP3 settings We again run both ILASP2 and ILASP3 with the same settings[15], so we simply refer to both as ILASP. We run ILASP as with the '–no-constraints' and '–no-aggregates' flags. We tell ILASP that each predicate is *positive*, which prevents ILASP from generating body literals using negation. We set ILASP to use at most 3 unique

[14] Cropper and Tourret 2019 show that it is impossible to generate a finite and complete set of metarules for a singleton-free fragment of monadic and dyadic Datalog.

[15] We consulted the ILASP authors for suggestions on which settings to run ILASP2 and ILASP3 with.

variables and at most 2 body literals ('-ml=2' and '–max-rule-length=3'). As in the primorial experiment, when we increased these parameters, ILASP struggled to find any solutions in the given time.

```
P(A):-Q(A).
P(A):-Q(A),R(A).
P(A):-Q(A,B),R(B).
P(A):-Q(A,B),P(B).
P(A):-Q(A,B),R(A,B).
```

```
P(A,B):-Q(B,A).
P(A,B):-Q(A,B),R(A,B).
P(A,B):-Q(A),R(A,B).
P(A,B):-Q(A,B),R(B).
P(A,B):-Q(A,C),R(C,B).
P(A,B):-Q(A,C),P(C,B).
```

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

5.2.2 Methods

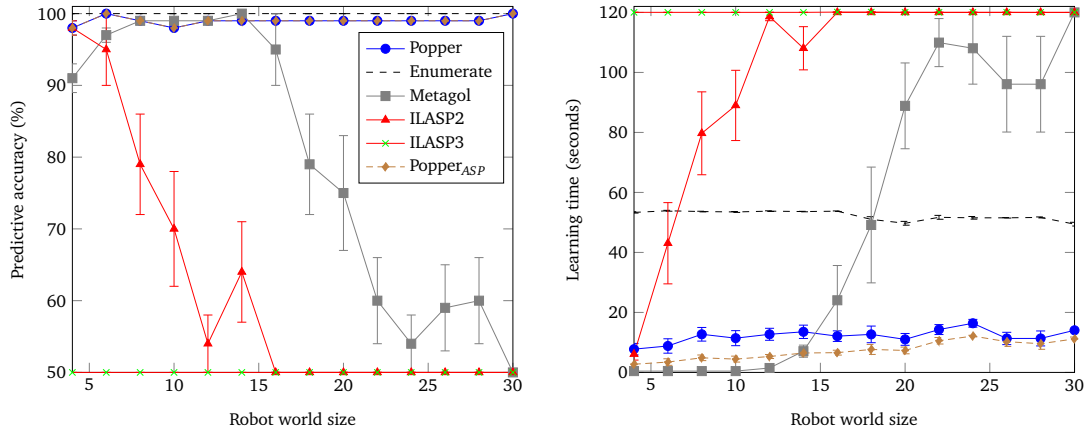
We run the experiment with an $n \times n$ grid world for each n in $\{4, 6, 8, \ldots, 28, 30\}$. To generate examples, for start states, we uniformly sample positions that are not at the top of the world. For the positive examples, the end state is the topmost position, e.g. (x, n) where n is the grid size. For negative examples, the end state is the topmost position but has the wrong horizontal coordinate, e.g. $(4, n)$ when starting at $(2, 3)$. We sample with replacement 5 positive and 5 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 2 minutes per task. We repeat each experiment 10 times and plot the standard error.

5.2.3 Results

Figure 11 shows the results. Enumerate achieves the best predictive accuracy out of all the systems. For small hypothesis spaces, this result is unsurprising because Enumerate tests every hypothesis. However, the predictive accuracy difference between Enumerate and Popper is negligible. Popper is 5 times quicker than Enumerate.

The learning times of Popper and Enumerate remain almost constant as the grid size grows. The reason 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 on the learning time of Popper and Enumerate 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.

Metagol slightly outperforms Popper in terms of learning times for grid worlds less than 14, but has worse predictive accuracy. However, as the grid size grows, Metagol's performance quickly degrades. Metagol's predictive accuracy drops because of learning timeouts, i.e. if Metagol fails to learn a solution then it only achieves default predictive accuracy (50%). For a grid size of 30, 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.



(a) Predictive accuracies (b) Learning times

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

Popper outperforms ILASP2 both in terms of predictive accuracies and learning times. ILASP2 struggles because it ground the rules in the hypothesis space with respect to the examples and BK, which is infeasible on non-trivial grid sizes, and is why its performance suffers as the domain size grows. ILASP2 outperforms ILASP3 because once ILASP2 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 is only a single ILASP positive example, and no negative examples) it is unnecessary and computationally expensive. We refer the reader to Law's thesis 2018 for a detailed comparison of ILASP2 and ILASP3[16].

To show the versatility of Popper, we modified Popper to test programs using ASP rather than Prolog. In other words, instead of learning Prolog programs, we set Popper to learn Datalog programs. Figure 11 shows the results as Popper$_{ASP}$. As expected, there is no difference in terms of predictive accuracies but Popper$_{ASP}$ can learn programs quicker than Popper because, in this problem, testing hypotheses using ASP is quicker than with Prolog.

The results from this experiment suggest that the answers to questions **Q1** and **Q2** are yes. The results also suggest that the answer to **Q3** is that Popper scales well and better-than state-of-the-art ILP systems with respect to 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. Metagol, ILASP2, and ILASP3

[16] We thank the ILASP2 and ILASP3 authors for this explanation.

can learn recursive programs. However, as the previous experiment showed, ILASP2 and ILASP3 struggle on large domains. We therefore compare Popper against Enumerate and Metagol.

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 tradic (`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 for them that generalise, an ILP system needs 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 very strong inductive bias[17].

We give each system the predicate declarations shown in Figure 12. Note that we use `increment/2` only 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 recursion[18] on almost every problem and could not find any solutions.

Popper and Enumerate settings We set Popper and Enumerate to use at most 6 unique variables, at most 5 body literals, and at most 2 clauses. For each BK relation, we also provide both systems with simple types and argument directions (whether input or output). In Section 5.5, we evaluate how sensitive Popper is to these parameters. 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.

Metagol settings For Metagol, we use almost the same metarules as in the previous robot experiment (Figure 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. Note that if we pick specific metarules for each task, then Metagol would perform better. To aid Metagol, we therefore replace the *inverse* metarule with the *identify* 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 is example-driven. Therefore, to aid Metagol, we provide the examples in increasing size (i.e. the length of the input lists).

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

[17] As mentioned in Section 2.3, some inverse entailment methods (Muggleton, 1995) might sometimes learn solutions for them. However, these approaches would need an 'base case' example to learn the base case of a recursive program, and then an example to learn the inductive base, and preferably in that order. 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.

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,B).
droplast	Drop the last element	droplast(A,B):—tail(A,B), tail (B,C),empty(C). 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):—even(A),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):—empty(A). 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 4: List transformation problems.

```
body_pred(head,2).          body_pred(empty,1).
body_pred(tail,2).          body_pred(zero,1).
body_pred(increment,2).     body_pred(one,1).
body_pred(decrement,2).     body_pred(even,1).
body_pred(geq,2).           body_pred(odd,1).
```

Fig. 12: Predicate declarations used by Popper, Enumerate, and Metagol in the list transformation experiments. We also provide `head_pred(P,A)` and `body_pred(P,A)` declarations, where `P` and `A` are the target predicate symbol and arity respectively.

50%. Each list is randomly generated and has a maximum length of 50. We sample the list elements uniformly at random from the set $\{1, 2, \ldots, 100\}$ (this choice is arbitrary and Popper and Metagol can handle much larger values). We measure the predictive accuracy and learning times. We enforce a timeout of 2 minutes per task. We repeat each experiment 10 times and plot the standard error.

5.3.3 Results

Table 5 shows the results. Popper equals or outperforms Enumerate on all the tasks in terms of predictive accuracies. Popper outperforms Enumerate on all but one of the tasks in terms of learning times. The exception is the `last` problem, where it is easier to simply enumerate all programs rather than use constraints. However, this difference is negligible. This result again suggests that the answer to **Q1** is yes.

	Accuracies			Times		
Name	**Popper**	**Enumerate**	**Metagol**	**Popper**	**Enumerate**	**Metagol**
addhead	**100** ± 0	**100** ± 0	50 ± 0	**1** ± 0	3 ± 0	120 ± 0
dropk	**100** ± 0	50 ± 0	50 ± 0	**1** ± 0	120 ± 0	120 ± 0
droplast	**100** ± 0	50 ± 0	50 ± 0	**39** ± 4	120 ± 0	120 ± 0
evens	**100** ± 0	50 ± 0	55 ± 5	**4** ± 0.41	120 ± 0	109 ± 11
finddup	99 ± 0	80 ± 0	**100** ± 0	13 ± 2	57 ± 18	**2** ± 0
last	**100** ± 0	**100** ± 0	**100** ± 0	0.72 ± 0.11	**0.55** ± 0.08	0.83 ± 0.09
len	**100** ± 0	50 ± 0	50 ± 0	**7** ± 1	120 ± 0	120 ± 0
member	**100** ± 0	**100** ± 0	75 ± 8	**0.14** ± 0.01	2 ± 0.01	0.42 ± 0.01
sorted	**100** ± 6	50 ± 0	50 ± 0	**77** ± 7	120 ± 0	120 ± 0
threesame	**99** ± 0	**99** ± 0	**99** ± 0	**0.32** ± 0.02	0.47 ± 0.04	0.35 ± 0.06

Table 5: List transformation predictive accuracies and learning times. We round predictive accuracies to integer values. We round learning times over 1 second to the nearest second. The error is standard error.

Popper equals or outperforms Metagol on all but one task in terms of predictive accuracy. The exception is the `finddup` problem, where there is only a 1% difference. Popper outperforms Metagol in terms of learning times in almost all cases. Note that Metagol could never learn a solution for `dropk` because it does not support triadic literals because of the metarule constraints. This result again suggests that the answer to **Q2** is yes.

5.4 Scalability

Our primorial experiment showed that Popper scales well in the size of the optimal solution size compared to Enumerate, ILASP, FastLAS, and Metagol. Our robot experiment showed that Popper scales well in the size of the domain compared to ILASP, FastLAS, and Metagol. The purpose of this experiment is to evaluate how well Popper scales in terms of the (1) number of examples, and (2) the size of examples. To do so, we repeat the `last` experiment from Section 5.3, where Popper and Metagol achieved similar performance.

5.4.1 Materials

We use the same materials as Section 5.3.

5.4.2 Settings

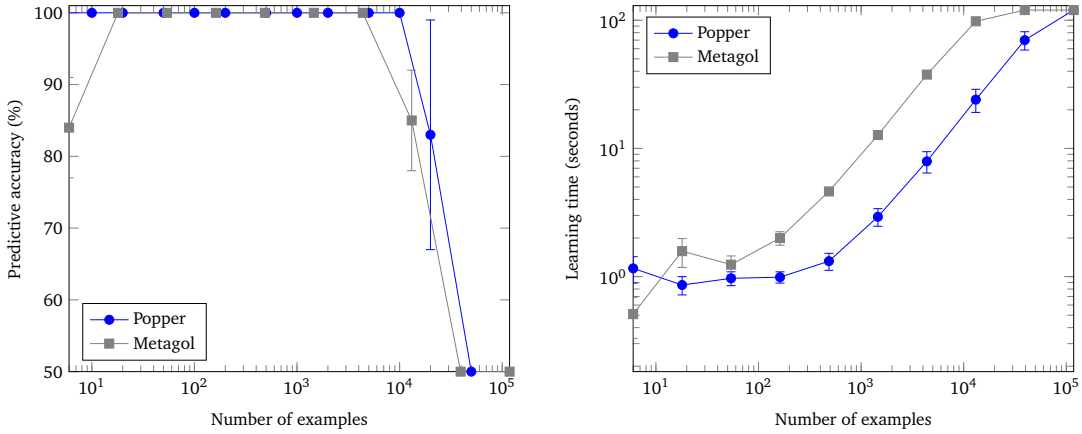
We run two experiments. In the first experiment we vary the number of examples. In the second experiment we vary the size of the examples (the size of the input list). For each experiment, we measure the predictive accuracy and learning times averaged over 10 repetitions.

Number of examples For each n in $\{1, 2, \ldots, 10\}$, we generate 3^n positive and 3^n negative training examples, and 1000 positive and 1000 negative testing examples, where each list has a maximum length of 100 and each element is a random integer from the range 1 to 1000.

Example size For each s in $\{50, 100, 150, \ldots, 500\}$, 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 to 1000.

5.4.3 Results

Figure 13 shows the results when varying the number of training examples. The predictive accuracies of Popper and Metagol are almost identical until around 10^4 examples. Given this many examples, Metagol struggles to find a solution in two minutes and eventually converges on the default predictive accuracy (50%). After almost 10^5 examples, Popper struggles to find a solution in two minutes. The reason is simply the overhead of testing hypotheses on that many examples. The log-log plot in Figure 13 shows that the learning times of both systems scale linearly in the number of examples, although Popper is twice as quick as Metagol in almost all cases. These results suggest that the answer to **Q3** is that Popper scales well with respect the number of examples.

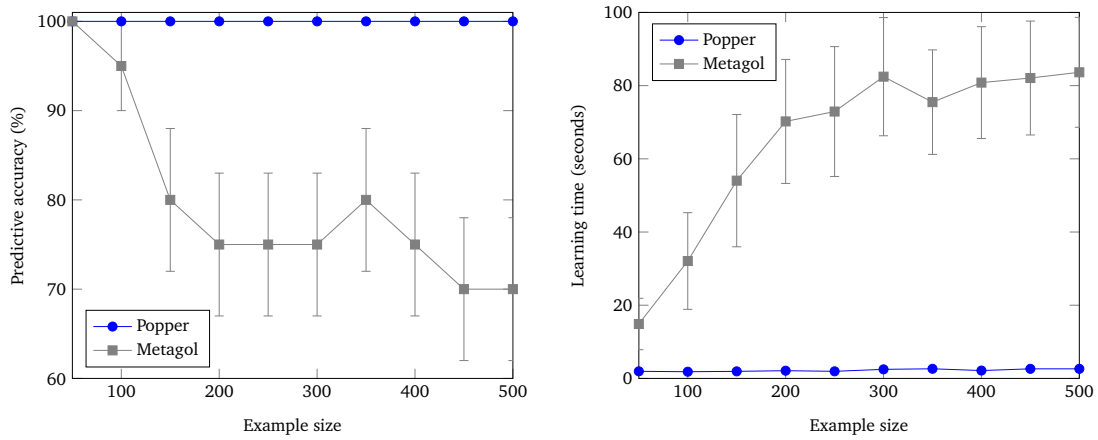


(a) Predictive accuracies

(b) Learning times

Fig. 13: The experimental results for the `last` task when varying the number of training examples.

Figure 14 shows the results when varying the size of the input (i.e. the size of the input list). The predictive accuracies and learning times of Popper remain almost constant as the size of the input grows. The mean learning times of Popper for examples of length 50 and 500 are 2 and 3 seconds respectively. 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 15 and 84 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 **Q2** is yes and the answer to **Q3** is that Popper scales well with respect to the size of examples.



(a) Predictive accuracies (b) Learning times

Fig. 14: The experimental results for the `last` task when varying the size (list length) of training examples.

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 Section 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

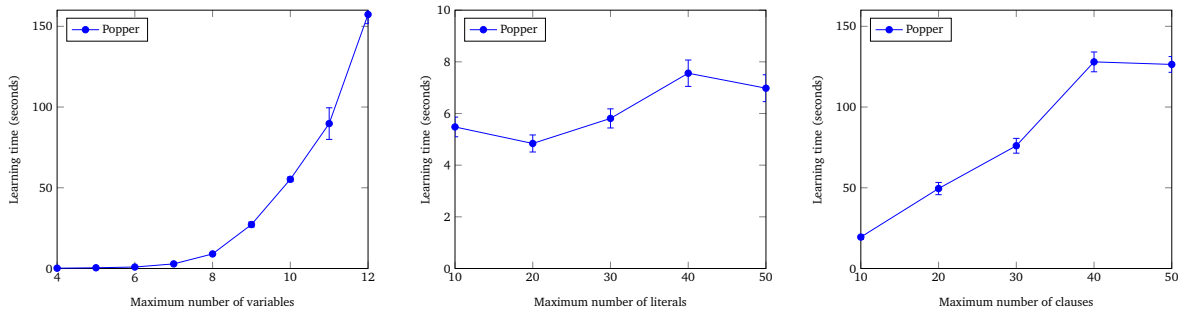
Figure 15 shows the experimental results. The results show that Popper is sensitive to the maximum number of unique variables, which has a strong influence on learning times. This result follows from Proposition 1 because more variables implies more ways to form literals in a clause. Somewhat surprisingly, 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 ILASP and many program synthesis systems, especially SAT approaches.

The results show that Popper is mostly insensitive to the maximum number of clauses. The main reason is because of the way Popper searches for programs of increasing size. For instance, for a program of size 4 (e.g. with four literals), due to constraints on the

hypothesis space (Section 4.1), it is impossible to generate a program with three clauses, since each clause must have a head and body literal.

Overall these results suggest that Popper is scales well with the maximum number of clauses and body literals parameters, but struggles with large values for the maximum number of unique variables.



(a) (b) (c)

Fig. 15: 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).

6 Conclusions and limitations

We have introduced an ILP approach called *learning programs by learning from failures*. Our approach 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 Section 3.5, we introduced three types of constraints: *generalisation*, *specialisation*, and *elimination* and proved their soundness in that they do not prune optimal solutions (Definition 14). This loop repeats until (1) the learner finds an optimal solution, or (2) there are no more hypotheses to test.

We implemented our idea 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. To improve efficiency, Popper uses multi-shot solving to combine the three stages. We showed that Popper is sound and complete with respect to optimal solutions (Theorem 1).

We evaluated our approach on three diverse domains (number theory problems, robot strategies, and list transformations). Our experiment results show that (1) constraints drastically reduce the hypothesis space, (2) Popper can substantially outperform state-of-the-art ILP systems Metagol, ILASP2, ILASP3, and FastLAS, both in terms of predictive accuracies and learning times, (3) Popper scales well with respect to domain size,

the number of training examples, and the size of the training examples, and (4) Popper is reasonably robust to its parameters.

6.1 Limitations and future work

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

6.1.1 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, 2019b; Dumancic et al., 2019). Popper does not currently support predicate invention, but we plan to support it in future work. 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. Alternatively Popper could mimic ILASP by support *prescriptive* predicate invention (Cropper et al., 2019a), where the arity and (in ILASP's case, argument types) are pre-specified by the language bias. Most of the results in this paper should extend to both approaches.

6.1.2 Noise

Most ILP systems handle noisy (misclassified) examples (Table 1). Popper does not currently support noisy examples. Our initial results suggest that 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, our early results suggest that noise handling increases learning times, which future work should explore.

6.1.3 Hypotheses

In most of our experiments Popper learns definite programs and tests them using Prolog. However, in Section 5.2, Popper learns Datalog programs and tests 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.

6.1.4 Better search

Popper is only one implementation of our learning from failures idea. An advantage of our separate three staged approach is that it allows for a variety of algorithms and implementations. Moreover, 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. For instance, in Section 5.2, we used ASP, rather than Prolog, to test hypotheses, which, in some cases, reduced learning times by 50%. Moreover, by decomposing the ILP problem into three

stages, our approach might mitigate the combinatorial and grounding problems faced by systems that solve the ILP problem as a single (and often very large) SAT problem (Corapi et al., 2011; Law et al., 2014; Kaminski et al., 2018; Evans and Grefenstette, 2018; Evans et al., 2019).

6.1.5 Better constraints

Hypothesis constraints are central to our idea. Popper uses predefined constraints to prune redundant programs from the hypothesis space (Section 4.1), such as recursive programs without a base case and subsumption redundant program. A key idea of our approach is to learn 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 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 (parameterised 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 Section 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 non-recursive 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; Raedt and Bruynooghe, 1993; Nienhuys-Cheng and Wolf, 1997) and theories (Nienhuys-Cheng and 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 thank Tobias Kaminski, Sebastijan Dumančić, and Richard Evans for extremely valuable feedback on the paper. We thank Mark Law for helping us to run ILASP2, ILASP3, and FastLAS in the experiments and for answering our (many) questions on the ILASP systems.

References

John Ahlgren and Shiu Yin Yuen. Efficient program synthesis using constraint satisfaction in inductive logic programming. *J. Mach. Learn. Res.*, 14(1):3649–3682, 2013. URL http://dl.acm.org/citation.cfm?id=2627674.

Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. Constraint-based synthesis of datalog programs. In J. Christopher Beck, editor, *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*, pages 689–706. Springer, 2017. 10.1007/978-3-319-66158-2_44. URL https://doi.org/10.1007/978-3-319-66158-2_44.

Duangtida Athakravi, Domenico Corapi, Krysia Broda, and Alessandra Russo. Learning through hypothesis refinement using answer set programming. In Gerson Zaverucha, Vítor Santos Costa, and Aline Paes, editors, *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*, pages 31–46. Springer, 2013. 10.1007/978-3-662-44923-3_3. URL https://doi.org/10.1007/978-3-662-44923-3_3.

Liviu Badea. A refinement operator for theories. In Céline Rouveirol and Michèle Sebag, editors, *Inductive Logic Programming, 11th International Conference, ILP 2001, Strasbourg, France, September 9-11, 2001, Proceedings*, volume 2157 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2001. 10.1007/3-540-44797-0_1. URL https://doi.org/10.1007/3-540-44797-0_1.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 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, 2017. URL https://openreview.net/forum?id=ByldLrqlx.

Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. *Artif. Intell.*, 101(1-2):285–297, 1998. 10.1016/S0004-3702(98)00034-4. URL https://doi.org/10.1016/S0004-3702(98)00034-4.

Ivan Bratko. Refining complete hypotheses in ILP. In Saso Dzeroski and Peter A. Flach, editors, *Inductive Logic Programming, 9th International Workshop, ILP-99, Bled, Slovenia, June 24-27, 1999, Proceedings*, volume 1634 of *Lecture Notes in Computer Science*, pages 44–55. Springer, 1999. 10.1007/3-540-48751-4_6. URL https://doi.org/10.1007/3-540-48751-4_6.

Alonzo Church. A note on the entscheidungsproblem. *J. Symb. Log.*, 1(1):40–41, 1936. 10.2307/2269326. URL https://doi.org/10.2307/2269326.

William W. Cohen. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artif. Intell.*, 68(2):303–366, 1994. 10.1016/0004-3702(94)90070-1. URL https://doi.org/10.1016/0004-3702(94)90070-1.

Domenico Corapi, Alessandra Russo, and Emil Lupu. Inductive logic programming as abductive search. In Manuel V. Hermenegildo and Torsten Schaub, editors, *Technical Communications of the 26th International Conference on Logic Programming, ICLP 2010, July 16-19, 2010, Edinburgh, Scotland, UK*, volume 7 of *LIPIcs*, pages 54–63. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010. 10.4230/LIPIcs.ICLP.2010.54. URL https://doi.org/10.4230/LIPIcs.ICLP.2010.54.

Domenico Corapi, Alessandra Russo, and Emil Lupu. Inductive logic programming in answer set programming. In Stephen Muggleton, Alireza Tamaddoni-Nezhad, and

Francesca A. Lisi, editors, *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*, pages 91–97. Springer, 2011. 10 .1007/978-3-642-31951-8_12. URL https://doi.org/10.1007/978-3-642-31951-8_12.

Vítor Santos Costa, Ashwin Srinivasan, Rui Camacho, Hendrik Blockeel, Bart Demoen, Gerda Janssens, Jan Struyf, Henk Vandecasteele, and Wim Van Laer. Query transformations for improving the efficiency of ILP systems. *J. Mach. Learn. Res.*, 4:465–491, 2003. URL http://jmlr.org/papers/v4/costa03a.html.

Andrew Cropper. Forgetting to learn logic programs. *CoRR*, abs/1911.06643, 2019a. URL http://arxiv.org/abs/1911.06643.

Andrew Cropper. Playgol: Learning programs through play. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 6074–6080. ijcai.org, 2019b. 10.24963/ijcai.2019/841. URL https://doi.org/10.24963/ijcai.2019/841.

Andrew Cropper and Sebastijan Dumancic. Learning large logic programs by going beyond entailment. *CoRR*, abs/2004.09855, 2020. URL https://arxiv.org/abs/2004.09855.

Andrew Cropper and Stephen H. Muggleton. Learning efficient logical robot strategies involving composable objects. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 3423–3429. AAAI Press, 2015. URL http://ijcai.org/Abstract/15/482.

Andrew Cropper and Stephen H. Muggleton. Metagol system. https://github.com/metagol/metagol, 2016. URL https://github.com/metagol/metagol.

Andrew Cropper and Sophie Tourret. Logical reduction of metarules. *Machine Learning*, Nov 2019. ISSN 1573-0565. 10.1007/s10994-019-05834-x. URL https://doi.org/10.1007/s10994-019-05834-x.

Andrew Cropper, Alireza Tamaddoni-Nezhad, and Stephen H. Muggleton. Meta-interpretive learning of data transformation programs. In Katsumi Inoue, Hayato Ohwada, and Akihiro Yamamoto, editors, *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*, pages 46–59. Springer, 2015. 10.1007/978-3-319-40566-7_4. URL https://doi.org/10.1007/978-3-319-40566-7_4.

Andrew Cropper, Richard Evans, and Mark Law. Inductive general game playing. *Machine Learning*, Nov 2019a. ISSN 1573-0565. 10.1007/s10994-019-05843-w. URL https://doi.org/10.1007/s10994-019-05843-w.

Andrew Cropper, Rolf Morel, and Stephen Muggleton. Learning higher-order logic programs. *Machine Learning*, Dec 2019b. ISSN 1573-0565. 10.1007/s10994-019-05862-7. URL https://doi.org/10.1007/s10994-019-05862-7.

Andrew Cropper, Sebastijan Dumancic, and Stephen H. Muggleton. Turning 30: New ideas in inductive logic programming. *CoRR*, abs/2002.11002, 2020. URL https://arxiv.org/abs/2002.11002.

Sebastijan Dumancic, Tias Guns, Wannes Meert, and Hendrik Blockeel. Learning relational representations with auto-encoding logic programs. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 6081–6087. ijcai.org, 2019.

10.24963/ijcai.2019/842. URL https://doi.org/10.24963/ijcai.2019/842.

Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. Learning libraries of subroutines for neurally-guided bayesian program induction. In *NeurIPS 2018*, pages 7816–7826, 2018. URL http://papers.nips.cc/paper/8006-learning-libraries-of-subroutines-for-neurallyguided-bayesian-program-induction.

Kevin Ellis, Maxwell I. Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a REPL. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, pages 9165–9174, 2019. URL http://papers.nips.cc/paper/9116-write-execute-assess-program-synthesis-with-a-repl.

Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *J. Artif. Intell. Res.*, 61:1–64, 2018. 10.1613/jair.5714. URL https://doi.org/10.1613/jair.5714.

Richard Evans, José Hernández-Orallo, Johannes Welbl, Pushmeet Kohli, and Marek J. Sergot. Making sense of sensory input. *CoRR*, abs/1910.02227, 2019. URL http://arxiv.org/abs/1910.02227.

Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 420–435. ACM, 2018. 10.1145/3192366.3192382. URL https://doi.org/10.1145/3192366.3192382.

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012. 10.2200/S00457ED1V01Y201211AIM019. URL https://doi.org/10.2200/S00457ED1V01Y201211AIM019.

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + control: Preliminary report. *CoRR*, abs/1405.3694, 2014. URL http://arxiv.org/abs/1405.3694.

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo. *Theory Pract. Log. Program.*, 19(1):27–82, 2019. 10.1017/S1471068418000054. URL https://doi.org/10.1017/S1471068418000054.

Tobias Kaminski, Thomas Eiter, and Katsumi Inoue. Exploiting answer set programming with external sources for meta-interpretive learning. *Theory Pract. Log. Program.*, 18(3-4):571–588, 2018. 10.1017/S1471068418000261. URL https://doi.org/10.1017/S1471068418000261.

Brenden M. Lake, Tomer D. Ullman, Joshua B. Tenenbaum, and Samuel J. Gershman. Building machines that learn and think like people. *CoRR*, abs/1604.00289, 2016. URL http://arxiv.org/abs/1604.00289.

Mark Law. *Inductive learning of answer set programs*. PhD thesis, Imperial College London, UK, 2018. URL http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.762179.

Mark Law, Alessandra Russo, and Krysia Broda. Inductive learning of answer set programs. In Eduardo Fermé and João Leite, editors, *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*, pages 311–325. Springer, 2014. 10.1007/978-3-319-11558-0_22. URL https://doi.org/10.1007/

978-3-319-11558-0_22.

Mark Law, Alessandra Russo, and Krysia Broda. Learning weak constraints in answer set programming. *Theory Pract. Log. Program.*, 15(4-5):511–525, 2015. 10.1017/S1471068415000198. URL https://doi.org/10.1017/S1471068415000198.

Mark Law, Alessandra Russo, and Krysia Broda. Iterative learning of answer set programs from context dependent examples. *Theory Pract. Log. Program.*, 16(5-6): 834–848, 2016. 10.1017/S1471068416000351. URL https://doi.org/10.1017/S1471068416000351.

Mark Law, Alessandra Russo, and Krysia Broda. Inductive learning of answer set programs from noisy examples. *Advances in Cognitive Systems*, 2018.

Mark Law, Alessandra Russo, Elisa Bertino, Krysia Broda, and Jorge Lobo. FastLAS: scalable inductive logic programming incorporating domain-specific optimisation criteria. In *AAAI*, 2020.

Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua B. Tenenbaum, and Stephen Muggleton. Bias reformulation for one-shot function induction. In Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan, editors, *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*, pages 525–530. IOS Press, 2014. 10.3233/978-1-61499-419-0-525. URL https://doi.org/10.3233/978-1-61499-419-0-525.

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

Donald Michie. Machine learning in the next five years. In Derek H. Sleeman, editor, *Proceedings of the Third European Working Session on Learning, EWSL 1988, Turing Institute, Glasgow, UK, October 3-5, 1988*, pages 107–122. Pitman Publishing, 1988.

Herman Midelfart. A bounded search space of clausal theories. In Saso Dzeroski and Peter A. Flach, editors, *Inductive Logic Programming, 9th International Workshop, ILP-99, Bled, Slovenia, June 24-27, 1999, Proceedings*, volume 1634 of *Lecture Notes in Computer Science*, pages 210–221. Springer, 1999. 10.1007/3-540-48751-4_20. URL https://doi.org/10.1007/3-540-48751-4_20.

Rolf Morel, Andrew Cropper, and C.-H. Luke Ong. Typed meta-interpretive learning of logic programs. In Francesco Calimeri, Nicola Leone, and Marco Manna, editors, *Logics in Artificial Intelligence - 16th European Conference, JELIA 2019, Rende, Italy, May 7-11, 2019, Proceedings*, volume 11468 of *Lecture Notes in Computer Science*, pages 198–213. Springer, 2019. 10.1007/978-3-030-19570-0_13. URL https://doi.org/10.1007/978-3-030-19570-0_13.

Stephen Muggleton. Inductive logic programming. *New Generation Comput.*, 8(4):295–318, 1991. 10.1007/BF03037089. URL https://doi.org/10.1007/BF03037089.

Stephen Muggleton. Inverse entailment and progol. *New Generation Comput.*, 13 (3&4):245–286, 1995. 10.1007/BF03037227. URL https://doi.org/10.1007/BF03037227.

Stephen Muggleton, Luc De Raedt, David Poole, Ivan Bratko, Peter A. Flach, Katsumi Inoue, and Ashwin Srinivasan. ILP turns 20 - biography and future challenges. *Machine Learning*, 86(1):3–23, 2012. 10.1007/s10994-011-5259-2. URL https://doi.org/10.1007/s10994-011-5259-2.

Stephen H. Muggleton, Dianhuan Lin, Niels Pahlavi, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning: application to grammatical inference. *Machine Learning*, 94(1):25–49, 2014. 10.1007/s10994-013-5358-3. URL https://doi.org/10.1007/s10994-013-5358-3.

Stephen H. Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic Datalog: predicate invention revisited. *Machine Learning*, 100(1):49–73, 2015. 10.1007/s10994-014-5471-y. URL https://doi.org/10.1007/s10994-014-5471-y.

Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. *Foundations of Inductive Logic Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997. ISBN 3540629270.

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

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 522–538. ACM, 2016. 10.1145/2908080.2908093. URL https://doi.org/10.1145/2908080.2908093.

Karl Popper. *The logic of scientific discovery*. Routledge, 2005.

J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990. 10.1007/BF00117105. URL https://doi.org/10.1007/BF00117105.

Luc De Raedt. *Logical and relational learning*. Cognitive Technologies. Springer, 2008. ISBN 978-3-540-20040-6. 10.1007/978-3-540-68856-3. URL https://doi.org/10.1007/978-3-540-68856-3.

Luc De Raedt and Maurice Bruynooghe. Interactive concept-learning and constructive induction by analogy. *Machine Learning*, 8:107–150, 1992. 10.1007/BF00992861. URL https://doi.org/10.1007/BF00992861.

Luc De Raedt and Maurice Bruynooghe. A theory of clausal discovery. In Ruzena Bajcsy, editor, *Proceedings of the 13th International Joint Conference on Artificial Intelligence. Chambéry, France, August 28 - September 3, 1993*, pages 1058–1063. Morgan Kaufmann, 1993.

Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. Provenance-guided synthesis of datalog programs. *PACMPL*, 4(POPL):62:1–62:27, 2020. 10.1145/3371130. URL https://doi.org/10.1145/3371130.

Oliver Ray. Nonmonotonic abductive inductive learning. *J. Applied Logic*, 7(3):329–340, 2009. 10.1016/j.jal.2008.10.007. URL https://doi.org/10.1016/j.jal.2008.10.007.

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

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

Ashwin Srinivasan and Ravi Kothari. A study of applying dimensionality reduction to restrict the size of a hypothesis space. In Stefan Kramer and Bernhard Pfahringer, editors, *Inductive Logic Programming, 15th International Conference, ILP 2005, Bonn, Germany, August 10-13, 2005, Proceedings*, volume 3625 of *Lecture Notes in Computer Science*, pages 348–365. Springer, 2005. 10.1007/11536314_21. URL https://doi.org/10.1007/11536314_21.

Phillip D. Summers. A methodology for LISP program construction from examples. *J. ACM*, 24(1):161–175, 1977. 10.1145/321992.322002. URL http://doi.acm.org/10.1145/321992.322002.

William Yang Wang, Kathryn Mazaitis, and William W. Cohen. Structure learning via parameter learning. In Jianzhong Li, Xiaoyang Sean Wang, Minos N. Garofalakis, Ian Soboroff, Torsten Suel, and Min Wang, editors, *Proceedings of the 23rd ACM*

International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014, pages 1199–1208. ACM, 2014. 10.1145/2661829.2662022. URL https://doi.org/10.1145/2661829.2662022.

Antonius Weinzierl. Blending lazy-grounding and CDNL search for answer-set solving. In Marcello Balduccini and Tomi Janhunen, editors, *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*, pages 191–204. Springer, 2017. 10.1007/978-3-319-61660-5_17. URL https://doi.org/10.1007/978-3-319-61660-5_17.

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 $M = \texttt{head:-body}_1, \ldots, \texttt{body}_m$ be a metarule. We use the clause encoding function *encodeSizedClause* from section 4.3.2 to derive an encoding of a metarule.

Example 13 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 *Ms* 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 *Ms*. A clause C is an instance of metarule $M \in Ms$ 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 $M \in Ms$, generate the following rule of the first kind:

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

The second rule is:

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