Machine Learning (2023) 112:3917–3943
https://doi.org/10.1007/s10994-023-06358-1

Learning logic programs by explaining their failures

Rolf Morel[1] · **Andrew Cropper**[1]

Received: 7 February 2022 / Revised: 1 May 2023 / Accepted: 14 June 2023 /
Published online: 7 August 2023

Abstract
Scientists form hypotheses and experimentally test them. If a hypothesis fails (is refuted), scientists try to *explain* the failure to eliminate other hypotheses. The more precise the failure analysis the more hypotheses can be eliminated. Thus inspired, we introduce failure explanation techniques for inductive logic programming. Given a hypothesis represented as a logic program, we test it on examples. If a hypothesis fails, we explain the failure in terms of failing sub-programs. In case a positive example fails, we identify failing sub-programs at the granularity of literals. We introduce a failure explanation algorithm based on analysing branches of SLD-trees. We integrate a meta-interpreter based implementation of this algorithm with the test-stage of the Popper ILP system. We show that fine-grained failure analysis allows for learning fine-grained constraints on the hypothesis space. Our experimental results show that explaining failures can drastically reduce hypothesis space exploration and learning times.

Keywords Relational learning · Inductive logic programming · Failure explanation

1 Introduction

Explanations are ubiquitous in our cognitive lives (Keil & Wilson, 2000). They are crucial to the process of forming hypotheses, testing them on data, analysing the results, and forming new hypotheses, that is to say, to science (Popper, 1963). For instance, imagine Alice is a chemist trying to synthesise a vial of a compound from two substances (e.g. *synth(thaum,slood,octiron)*). Alice can perform actions, such as fill a vial with a substance (*fill(Vial,Sub)*) or mix two vials (*mix(V1,V2,V3)*), and sequence them to form a hypothesis, e.g.:

Editors: Alireza Tamaddoni-Nezhad, Alan Bundy, Luc De Raedt, Artur d'Avila Garcez, Sebastijan Dumančić, Cèsar Ferri, Pascal Hitzler, Nikos Katzouris, Denis Mareschal, Stephen Muggleton, Ute Schmid.

✉ Rolf Morel
rolf.morel@cs.ox.ac.uk

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

[1] University of Oxford, Oxford, UK

$$synth(A, B, C) \leftarrow fill(V1, A), fill(V1, B), mix(V1, V1, C)$$

This hypothesis says that to synthesise a vial of compound C, fill vial *V1* with substance A, fill vial *V1* with substance B, and mix vial *V1* with itself to form C.

When Alice experimentally tests this hypothesis she finds that it *fails*. From this failure Alice concludes (**C1**) that hypotheses which add further actions (i.e. literals) will also fail. However, as Alice observed that the second action caused the failure, she can *explain* the failure as "vial *V1* cannot be filled a second time". This allows her to conclude (**C2**) that any hypothesis that includes *fill(V1,A)* and *fill(V1,B)* will fail. Clearly, conclusion **C2** allows Alice to eliminate more hypotheses than **C1**. That is, by explaining failures Alice can better form new hypotheses.

We formalise this mode of reasoning for explaining failures of logical theories. We do so in the context of inductive program synthesis, where the goal is to machine learn computer programs from data (Ehud, 1983). Existing inductive logic programming (ILP) approaches fail to generalise from observed failures. Many ILP systems (Ahlgren & Yuen, 2013; Cropper & Morel, 2021; Law, 2018) only learn from the failure of an entire hypothesis—as Alice does when she concludes **C1**—and cannot explain why a hypothesis fails, e.g. cannot reason like Alice does to conclude **C2**. Some systems can identify parts of a program that cause a failure, but cannot learn from this information. For instance, (Cropper & Muggleton, 2016) will repeatedly retry failing program fragments.

We address these limitations by automatically explaining program failures, taking inspiration from algorithmic debugging (Caballero et al., 2017). The idea is to analyse the failure of a hypothesis to identify *sub-programs* that also fail. To illustrate, consider hypothesis H_1:

$$\{\ droplast(A, B) \leftarrow empty(A), tail(A, B)\ \}$$

If *droplast([1,2],[1])* is a positive example, then H_1 does not cover this example. From this failure we can learn that H_1's sub-program { *droplast(A,B)* ← *empty(A)* } also does not cover this example. We show that by identifying failing sub-programs and accumulating constraints generated from them, we can eliminate more hypotheses (e.g. any single clause program that expands the above sub-program). When the overhead of failure explanation is low, our approach reduces learning times.

Most logic program debugging systems (Köhler et al., 2012; Thompson & Sullivan, 2020) and some synthesis systems (Ehud, 1983; Raghothaman, 2020) can identify a subset of clauses as being the cause of a failure. We additionally identify literals *within* clauses responsible for failure [without the requirement of trace-complete examples needed by theory revision systems such as FORTE Richards and Mooney (1995)]. We show that this fine-grained failure analysis allows for learning finer-grained constraints on the hypothesis space.

Our contributions are:

- We relate logic programs that fail on examples to their failing sub-programs. For wrong answers we identify clauses. For missing answers we additionally identify literals within clauses.
- We show that hypotheses that are specialisations and generalisations of failing sub-programs can be eliminated, and prove that hypothesis space pruning based on sub-programs is more effective than pruning without them.

- We introduce Hempel, an ILP system extending the Popper ILP system, which analyses SLD-trees to automatically explain failures in terms of sub-programs.
- We experimentally show that failure explanation can drastically reduce (i) hypothesis space exploration and (ii) learning times.

2 Related work

Program synthesis Inductive program synthesis systems automatically generate computer programs from specifications, typically input/output examples (Ehud, 1983). This topic interests researchers from many areas of machine learning, including Bayesian inference (Silver et al., 2020) and neural networks (Ellis et al., 2018). We focus on ILP techniques, which induce logic programs (Muggleton, 1991).

Recursion Both classical ILP systems (Blockeel & Raedt, 1998; Muggleton, 1995; Srinivasan, 2001) as well as many modern ones, e.g. (Ahlgren & Yuen, 2013), struggle to learn recursive programs, or cannot learn them at all, e.g. (Schüller & Benz, 2018) and FastLAS Law et al. (2020). By contrast, our system, Hempel, can learn recursive programs and thus programs that generalise to input sizes it was not trained on. Compared to many modern ILP systems (Evans & Grefenstette, 2018; Evans et al., 2021; Kaminski et al., 2019), Hempel supports large and infinite domains, which is important when reasoning about complex data structures, such as lists. In addition, unlike many state-of-the-art systems (Cropper & Muggleton, 2016; Evans & Grefenstette, 2018; Hocquette & Muggleton, 2020; Kaminski et al., 2019), Hempel does not require metarules (i.e. program templates) to restrict the hypothesis space.

Algorithmic debugging Algorithmic debugging (Caballero et al., 2017) explains failures in terms of sub-programs. Alongside his seminal work on logic program synthesis, Shapiro (Ehud, 1983) introduced the notion of *debugging trees* for semi-automated identification of failing clauses. Only being able to return clauses responsible for entailing an atom is still the standard for logic programming debugging (Köhler et al., 2012; Thompson & Sullivan, 2020). Unlike these systems, we automatically identify literals within clauses which cause an atom to not be entailed, and integrate the failure explanation process in a program synthesis system.

Theory revision and repair Shapiro's Model Inference System (MIS) (Ehud, 1983) is a theory revision system which, through interaction with a user, is capable of synthesising programs. MIS uses SLD-trees to determine which clauses of a program are responsible for entailing a negative example, at which point the user needs to say which of these clauses is wrong. To cover a non-covered positive example, additional clauses get added, possibly involving user-interaction, without regard for why the current clauses do not entail this example. By contrast, Hempel does not require an oracle and can automatically identify clauses and literals within clauses as being responsible for not entailing a positive example.

There are theory revision systems (Wrobel, 1996) able to identify literals as *revision points* within theories, though often with limitations. Some require user-interaction (Pazzani & Brunk, 1991; Raedt & Bruynooghe, 1992). FORTE (Richards & Mooney, 1995) uses hill-climbing to gradually revise a theory, heuristically following revisions that improve training accuracy. Unlike FORTE, Hempel is guaranteed to find an optimal solution if one exists. FORTE can automatically identify responsible literals of a sub-program, given that the examples are trace-complete, i.e. all necessary recursive calls of the target predicate are included as positive examples. Our failure explanation algorithm

automatically identifies responsible clauses and literals which cause a program to not entail an atom, without any condition on the examples.

In general, theory revision and theory repair (Bundy & Mitrovic, 2016) are concerned with updating a current hypothesis by applying generalisation and specialisation operators to the identified revision points. Whereas these systems refine a single program at a time, HEMPEL uses the failure of a (sub-)program to *refine the hypothesis space*, each time pruning away a large class of programs.

Failure explanation Some modern ILP systems can be said to have a degree of failure explanation.

METAGOL (Cropper & Muggleton, 2016) is a meta-interpreter which uses examples to drive the search, gradually building up a program whilst partially evaluating it on an example. When a failure occurs, Metagol knows it is due to the last literal that was added, which causes it to backtrack. However, due to its iterative deepening strategy, METAGOL will reconsider these program fragments many times, and has no way to learn from failures. By contrast, HEMPEL learns constraints which ensure that failing program fragments are never reconsidered.

ILASP3 (Law, 2018) learns recursive ASP programs, with *partial interpretations* serving as examples. It starts by enumerating the space of candidate rules, assigning each an id. Next a select-test-constrain loop selects a hypothesis, a subset of the candidate clauses, based solely on constraints over the ids. When a model of a selected hypothesis does not correctly extend the given partial interpretations, the hypothesis fails with the model being its *violating reason*. Constraints can be derived from a violating reason by checking which combinations of candidate rules also have it as a model, which is an expensive operation. HEMPEL's learning of constraints by identifying sub-programs is more efficient and, by defining its hypothesis selection problem over literals, it is not restricted to identifying just clauses as causing a failure.

Like ILASP3, PROSYNTH (Raghothaman, 2020) precomputes every possible clause and employs a select-test-constrain loop over clause ids. PROSYNTH uses the notion of *query provenance* (Cheney et al., 2009) for identifying which clauses of a hypothesis are responsible for (not) entailing an example, encoding identified subsets as constraints. PROSYNTH learns Datalog programs, which is just a fragment of the definite programs which can be learned by HEMPEL. Additionally, HEMPEL's failure explanation is finer grained as it also identifies which literals cause failure.

Learning from failures Our system builds on POPPER (Cropper & Morel, 2021), see Sect. 5. POPPER learns first-order constraints by a process that is similar to conflict-driven clause learning (João, 2009). The constraints that Popper learns are always based on entire hypotheses (i.e. it only reasons as Alice does for conclusion **C1** in the introduction). HEMPEL's failure explanation can hence be viewed as allowing POPPER to detect smaller, finer-grained conflicts, yielding smaller and more general constraints which prune more effectively (which brings the reasoning about failures up to the level of conclusion **C2**).

3 Problem setting

In this section, we (i) describe our problem setting; (ii) relate specialisations and generalisations to missing and incorrect answers; (iii) define failing sub-programs; and (iv) show that sub-programs lead to better pruning.

Preliminaries. We assume standard logic programming definitions (Lloyd, 2012). We define θ-subsumption (Midelfart, 1999; Plotkin, 1971). A clause C_1 *subsumes* a clause C_2 iff there exists a substitution θ such that $C_1\theta \subseteq C_2$. A clausal theory T_1 subsumes a clausal theory T_2 iff $\forall C_2 \in T_2, \exists C_1 \in T_1$ such that C_1 subsumes C_2. Subsumption implies entailment, i.e. if T_1 subsumes T_2 then $T_1 \models T_2$.

3.1 Learning from failures

We adopt the learning from failures (LFF) approach to ILP (Cropper & Morel, 2021). Let $\mathcal{H}$ be a set of hypotheses, where each hypothesis is a definite program (a set of definite clauses). Hypothesis space pruning is made explicit in LFF by means of *hypothesis constraints*. For our purposes, it suffices to see a hypothesis constraint as a set of programs, typically related by their syntax, where the purpose of this set is to *prune*, i.e. rule out, these hypotheses. For example, given a program P, a hypothesis constraint could prune any program $Q \in \mathcal{H}$ such that $P \subseteq Q$, i.e. any program that adds clauses to P. Given a set of hypothesis constraints $C = \{C_1, \ldots, C_n\}$, $\mathcal{H}_C = \mathcal{H} \backslash (C_1 \cup \ldots \cup C_n)$ denotes the set of all hypotheses *not* pruned by the individual constraints.

We define LFF's input[1] and introduce our running example:

Definition 1 (**LFF input**) A *LFF* input is a tuple $(E^+, E^-, \mathcal{H}, B, C)$ where E^+ and E^- are sets of ground atoms denoting positive and negative examples respectively; $\mathcal{H}$ is a set of hypotheses; B is a definite program denoting background knowledge[2]; and C is a set of hypothesis constraints.

Example 1 To illustrate LFF, consider an input for learning a *droplast/2* program. Suppose our hypotheses $\mathcal{H}$ are definite programs with *droplast/2* in the head of each clause and *droplast/2*, *empty/1*, *head/2*, *tail/2* and *cons/3* occurring in bodies. Our background knowledge B consists of definitions for these predicates, except for *droplast/2*. $E^+ = \{droplast([1,2,3],[1,2]), droplast([1,2],[1])\}$ and $E^- = \{droplast([1,2],[])\}$ are our positive and negative examples. Our set of hypothesis constraints C is initially empty.

We define a LFF solution:

Definition 2 (**LFF solution**) Given an input tuple $(E^+, E^-, \mathcal{H}, B, C)$, a hypothesis $H \in \mathcal{H}_C$ is a *solution* when H is *complete* ($\forall e \in E^+,\ B \cup H \models e$) and *consistent* ($\forall e \in E^-,\ B \cup H \not\models e$).

If a hypothesis is not a solution then it is a *failing* hypothesis. A hypothesis H is *incomplete* when $\exists e^+ \in E^+,\ H \cup B \not\models e^+$. A hypothesis H is *inconsistent* when $\exists e^- \in E^-,\ H \cup B \models e^-$. A hypothesis H_1 is a *specialisation* of hypothesis H_2 when H_2 subsumes H_1. Symmetrically, a hypothesis H_1 is a *generalisation* of hypothesis H_2 when H_1 subsumes H_2.

Key to LFF is the ability to learn hypothesis constraints from failed hypotheses. Given an incomplete hypothesis H, a *specialisation constraint* prunes specialisations of H.

[1] We work with a more abstract LFF input than its original definition: our hypothesis spaces and its constraints are just sets rather than sets being represented by formulae in a constraint satisfaction language.

[2] The background knowledge program can make use of functional symbols.

Similarly, given an inconsistent hypothesis H', a *generalisation constraint* prunes generalisations of H'. These constraints are *sound*, that is, they do not prune solutions.

3.2 Missing and incorrect answers

Given background knowledge B, the failure of a hypothesis H is due to at least one example. We adopt the following terminology from the algorithmic debugging community (Caballero et al., 2017; Ehud, 1983). A positive example e^+ is a *missing answer* when $B \cup H \not\models e^+$. Similarly, a negative example e^- is an *incorrect answer* when $B \cup H \models e^-$. We relate missing and incorrect answers to specialisations and generalisations. If H has a missing answer e^+, then, as a specialisation H' of H entails at most as much as H, e^+ is a missing answer of H' as well. Hence all specialisations of H are incomplete and can be eliminated. Similarly, as generalisations of H entail at least as much as H, if e^- is an incorrect answer of H, all generalisations of H are inconsistent and can be pruned.

Example 2 (**Missing answers and specialisations**) Given the LFF input from Example 1, consider the following *droplast* hypothesis:

$$H_1 = \left\{ droplast(A,B) \leftarrow empty(A), tail(A,B) \right\}$$

Both $droplast([1,2,3],[1,2])$ and $droplast([1,2],[1])$ are missing answers of H_1, so H_1 is incomplete and we can prune its specialisations, e.g. programs that add literals to the clause.

Example 3 (**Incorrect answers and generalisations**) Consider hypothesis H_2:

$$H_2 = \left\{ \begin{array}{l} droplast(A,B) \leftarrow tail(A,C), tail(C,B) \\ droplast(A,B) \leftarrow tail(A,B) \end{array} \right\}$$

In addition to being incomplete, H_2 is inconsistent because of the incorrect answer $droplast([1,2],[])$, so along with specialisations we can prune the generalisations of H_2, e.g. programs with additional clauses.

3.3 Failing sub-programs

We now consider explaining failures in terms of failing sub-programs. The idea is to identify sub-programs that cause the failure. Consider the following two examples:

Example 4 (**Explain missing answer**) Consider previously defined H_1 and positive example $e^+ = droplast([1,2],[1])$. An explanation for why H_1 does not entail e^+ is that $empty([1,2])$ fails. It follows that e^+ is a missing answer of $H_1' = \left\{ droplast(A,B) \leftarrow empty(A) \right\}$. As H_1' is incomplete we can prune all of its specialisations.

Example 5 (**Explain incorrect answer**) Consider negative example $e^- = droplast([1,2],[])$ and H_2. The first clause of H_2 always entails e^- irrespective of other clauses being part of the hypothesis. It follows that e^- is an incorrect answer of $H_2' = \{ droplast(A,B) \leftarrow tail(A,C), tail(C,B) \}$. As H_2' is inconsistent we can prune all of its generalisations.

Note that when a system like POPPER observes that H_2 fails, it is not able to prune based on H_2'. Whilst costly, an ILP system like ProSynth could learn that H_2' fails. Given H_1 and its failure, POPPER, ILASP3 and ProSynth are unable to determine it is possible to prune based on H_1'.

We now define a *sub-program*:

Definition 3 (**Sub-program**) A definite program P is a *sub-program* of a definite program Q if and only if either:

- P is the empty set
- there exists clauses $C_p \in P$ and $C_q \in Q$ such that $C_p \subseteq C_q$ and $P \backslash \{C_p\}$ is a sub-program of $Q \setminus \{C_q\}$

In this definition, arguments of literals must be syntactically the same[3] for the clause subset check to succeed. In functional program synthesis, sub-programs are typically defined by leaving out nodes in the parse tree of the original program [e.g., (Feng, 2018)]. Our definition generalises this idea by allowing for arbitrary ordering of clauses and literals.

In the above examples, H_1' is a sub-program of H_1 and so is H_2' of H_2. Note that clauses and literals can be dropped at the same time, e.g. $\{ droplast(A,B) \leftarrow tail(A,C) \}$ is another sub-program of H_2.

We define the failing sub-programs problem:

Definition 4 (**Failing sub-programs**) Given definite program P and sets of examples E^+ and E^-, the *failing sub-programs problem* is to find all sub-programs of P that do not entail an example of E^+ or do entail an example of E^-.

By definition, a failing sub-program has a missing answer and/or an incorrect answer. Hence we can always prune specialisations and/or generalisations of a failing sub-program. We show that sub-programs are effective at pruning:

Theorem 1 (**Better pruning**) *Let H be a definite program that fails and P ($\neq H$) be a sub-program of H that fails. Let $C(H)$ and $C(P)$ be the specialisation and/or generalisation constraints derivable for H and P, respectively. If neither of (i) P is a specialisation of H, H is incomplete and P is not inconsistent, or (ii) P is a generalisation of H, H is inconsistent and P is not incomplete, apply, then $\mathcal{H}_{C(H) \cup C(P)} \subset \mathcal{H}_{C(H)}$, i.e. constraints derived for P prune programs not pruned by constraints derived for H.*

[3] Our definition hence insists on variable names in literals of a sub-program Q being the same as variable names in the corresponding literals of program P.

Proof By case distinction on how P and H are related by subsumption. Note that because $P \neq H$, either P and H are not related by subsumption, or P subsumes H, or H subsumes P.

Suppose H subsumes P, i.e. P is a specialisation of H. If H is incomplete, then all of H's specialisations can be pruned, which includes P and its specialisations. Hence if P is only incomplete then no additional pruning can be achieved, which is exception (i). If P is (additionally) inconsistent, then P's generalisations can be pruned. In addition to H being among P's generalisations, there are also programs incomparable with H among P's generalisations, so more pruning can be achieved.

Now suppose P subsumes H, i.e. P is a generalisation of H. If H is inconsistent, then all of H's generalisations can be pruned, which includes P and its generalisations. Hence if P is only inconsistent then no additional pruning can be achieved, which is exception (ii). If P is (additionally) incomplete, then P's specialisations can be pruned. In addition to H being among P's specialisations, there are also programs incomparable with H among P's specialisations, so more pruning can be achieved.

In the remaining case, where H and P are not related by subsumption, it is immediate that the specialisation/generalisation constraints derived for P prune a distinct part of the hypothesis space, e.g. H's constraints do not prune P. □

4 Failure explanation algorithm

We now present a method for identifying failing sub-programs. The approach is based on the observation that branches of an SLD-tree correspond to sub-programs. Our algorithm identifies clauses responsible for entailing a negative example. It is when a program fails to prove entailment that our approach distinguishes itself. Namely, we also identify literals *within* clauses which cause a positive example to not be entailed. As the presented method relies on SLD-resolution, from this point on we assume left-to-right evaluation of literals within clauses.

4.1 SLD-trees

In algorithmic debugging, missing and incorrect answers help characterise which parts of a *debugging tree* are wrong (Caballero et al., 2017). Debugging trees can be seen as generalising SLD-trees, with the latter representing the search for a refutation (Nienhuys-Cheng & de Wolf, 1997). We address the failing sub-programs problem by analysing SLD-trees, only identifying a subset of them. A *branch* in a SLD-tree is a path from the root *goal* to a leaf. Each goal on a branch has a *selected atom*, on which resolution is performed to derive child goals. A branch that ends in an empty leaf is called *successful*, as such a path represents a refutation. Otherwise a branch is *failing*. Note that selected atoms on a branch identify a subset of the literals of a program.

```
def failing_subprogs⁻(B, H, e⁻):
    T = SLD-tree of B ∪ H ∪ {¬e⁻}
    subprogs = {}
    for every successful branch λ of T:
        H′ = sub-program of H identified by
             H's clauses that occur in λ
        subprogs = subprogs ∪ {H′}
    return subprogs
```

```
def failing_subprogs⁺(B, H, e⁺):
    T = SLD-tree of B ∪ H ∪ {¬e⁺}
    subprogs = {}
    for every failing branch λ of T:
        H′ = sub-program of H identified by
             H's literals that occur in λ
        if SLD-res. fails to prove B ∪ H′ ⊨ e⁺:
            subprogs = subprogs ∪ {H′}
    return subprogs
```

Fig. 1 Identify failing sub-programs from branches in SLD-trees

4.2 Identifying sub-programs

Let B be a definite program, H be a hypothesis, and e be a atom.[4] The SLD-tree T for $B \cup H \cup \{\neg e\}$, with $\neg e$ as the root, proves $B \cup H \models e$ iff T contains a successful branch. Given a branch λ of T, we define the λ-sub-program of H. A literal L of H occurs in λ *-sub-program* H' if and only if L occurs as a selected atom[5] in λ or L was used to produce a resolvent that occurs in λ. The former case is for literals in the body of clauses and the latter for head literals. Now consider the SLD-tree T' for $B \cup H' \cup \{\neg e\}$ with $\neg e$ as root. As all literals necessary for λ occur in $B \cup H'$, the branch λ must occur in T' as well.

Suppose e^- is an incorrect answer for hypothesis H. Then the SLD-tree for $B \cup H \cup \{\neg e^-\}$ has a successful branch λ. The literals of H necessary for this branch are also present in λ-sub-program H', hence e^- is also an incorrect answer of H'. Now suppose e^+ is a missing answer of H. Let T be the SLD-tree for $B \cup H \cup \{\neg e^+\}$ and λ' be any failing branch of T. The literals of H in λ' are also present in λ'-sub-program H''. While λ' must be a failing branch present in the SLD-tree of $B \cup H'' \cup \{\neg e^+\}$, this is, in general, insufficient for concluding that this SLD-tree has no successful branch. Hence whether e^+ is indeed a missing answer of H'' needs to be verified.

Figure 1 shows the corresponding procedures for deriving failing sub-programs, in the case of a negative example and a positive example, respectively. Note that hypothesis H can refer to library B but B is not allowed to refer to H. Hence whilst resolving a selected literal of H defined by B with clauses of B we cannot encounter literals of H. Therefore, for failure explanation purposes, we need not inspect the part of the SLD-tree for $B \cup H \cup \{\neg e\}$ that deals with determining whether a literal defined by B holds or not. This is equivalent to viewing B as a (possibly infinite) set of facts, i.e. resolving a selected literal defined by B always returns directly. This is how we will treat resolving literals of B from this point on.

The following example illustrates identifying sub-programs from the SLD-trees of a recursive program.

Example 6 Let H be the following recursive `droplast/2` hypothesis, where the name `droplast` has been shortened to `dl`:

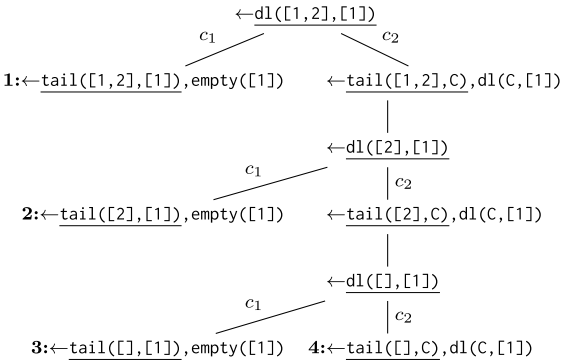
c_1 : `dl(A,B) :- tail(A,B), empty(B).`

c_2 : `dl(A,B) :- tail(A,C), dl(C,B).`

[4] While in our application to synthesis we only use ground atoms e, the failure explanation algorithm presented in this section also works when e is non-ground.

[5] Note that resolution might have unified arguments of L to produce the selected atom.

Suppose B includes the usual definitions for `tail/2` and `empty/1`. Testing whether $B \cup H \models$ `dl([1,2],[1])` holds is done by SLD-resolution. The SLD-tree for $B \cup H \cup \{\neg$`dl([1,2],[1])`$\}$ is:



Each node is a goal and has its selected literal underlined. The SLD-tree has four branches, each of them failing. The branch marked '**1:**' identifies the sub-program $P_1 =$ { `dl(A,B):- tail(A,B).` } as only clause c_1 is used and only its head and first body literal are evaluated. The branches marked '**2:**' and '**3:**' identify the sub-program $P_2 =$ { `dl(A,B):- tail(A,B). dl(A,B):- tail(A,C),dl(C,B).` } as both clauses are used though the second literal of c_1 is never selected while all of the literals of c_2 are. The branch marked '**4:**' never uses clause c_1 and hence identifies sub-program $P_3 = \{c_2\}$. Retesting `dl([1,2],[1])` on these sub-programs confirms that they fail.

Now consider testing for $B \cup H \models$ `dl([1,2],[])`. The SLD-tree for $B \cup H \cup \{\neg$`dl([1,2],[])`$\}$ has failing branches but also a successful one: ← `dl([1,2],[])` $\xrightarrow{c_2}$ ← `tail([1,2],C),dl(C,[])` — ← `dl([2],[])` $\xrightarrow{c_1}$ ← `tail([2],[]),empty([])` — ← `empty([])` — □. As this branch used all clauses, it identifies H itself as responsible. On the other hand, the SLD-tree for $B \cup H \models$ `dl([1],[])` has a successful branch only using c_1: ← `dl([1],[])` $\xrightarrow{c_1}$ ← `tail([1],[]),empty([])` — ← `empty([])` — □. Hence it identifies $P_4 = \{c_1\}$ as the responsible sub-program.

5 Implementation

Before introducing our ILP system, Hempel, we discuss our implementation of the failure explanation algorithm.

5.1 Meta-Interpreter for failure explanation

We implement our failure explanation algorithm by a meta-interpreter, `mi` $_{tr}$, where this meta-interpreter is best understood as instrumenting the program such that executing it keeps track of which parts of the program actually got executed.

Given a background knowledge program B and an atom G, $\texttt{mi}_{tr}$ keeps track of which literals of a definite program P have been encountered along each branch of the SLD-tree of $B \cup P \cup \{\neg G\}$. For each literal of the hypothesis P being evaluated we keep track of one bit of information: whether this literal[6] has been seen along the current branch or not. $\texttt{mi}_{tr}$ maintains a bitset, which we refer to as a *trace*, containing a unique bit for each literal of the hypothesis.

The meta-interpreter assumes a program transformation $X(\cdot)$ has been applied to the program (where, for notational convenience, clauses are represented by disjunctions):

$$\begin{aligned} X(P) &= \{X(C, C_{idx}) \mid C \in P\} \\ &= \{\bigvee \begin{cases} \neg X(A, C_{idx}, L_{idx}) \text{ if } L = \neg A \\ X(L, C_{idx}, L_{idx}) \quad \text{otherwise} \end{cases} \mid L \in C \wedge C \in P\} \end{aligned}$$

Before defining $X(\cdot,\cdot,\cdot)$, we specify how bitsets are derived. C_{idx} and L_{idx} correspond to the index of clause C within P and the index of L within C, respectively. The function $bitset(\cdot,\cdot)$ converts a clause index and literal index within that clause to a bitset with a unique bit set for these inputs. $X(L, C_{idx}, L_{idx}) := \texttt{mi}(L, bitset(C_{idx}, L_{idx}))$, if the predicate of L is defined by P. Otherwise $X(L, C_{idx}, L_{idx}) := \texttt{call}(L, bitset(C_{idx}, L_{idx}))$, i.e. in the case the predicate of L is defined by the background knowledge.

Figure 2 lists the code for meta-interpreter $\texttt{mi}_{tr}$. Given an atom G and program $X(P)$, we can evaluate G as a goal using the meta-interpreter by invoking $\texttt{mi}_{tr}(\texttt{mi}(G,0),0,\texttt{Trace})$, where 0 denotes the empty bitset. When this call succeeds, Trace will have become unified with a bitset identifying all literals that occurred on the first successful branch in the SLD-tree of $B \cup P \cup \{\neg G\}$. If evaluation of $\texttt{mi}_{tr}(\texttt{mi}(G,0),0,\texttt{Trace})$ fails then there is no successful branch in the SLD-tree of $B \cup P \cup \{\neg G\}$. In this case $\texttt{mi}_{tr}$ will have asserted traces for each unsuccessful branch, via a non-logical predicate assert_failed_trace[7]. Upon $\texttt{mi}_{tr}(\texttt{mi}(G,0),0,\texttt{Trace})$ having failed, all these asserted traces can be inspected to obtain the corresponding sub-programs.

Note that $\texttt{mi}_{tr}$ only does a constant number of additional (bitset unioning / logical *or*) operations at every node of the SLD-tree of $B \cup P \cup \{\neg G\}$ involving literals of H (that is, resolving literals defined B is relegated to the normal interpreter). Hence the SLD-tree of $B \cup X(P) \cup \{\neg\, \texttt{mi}_{tr}(\texttt{mi}(G, 0), 0, \texttt{Trace})\}$ is only a constant factor bigger than the original. It follows that the overhead $\texttt{mi}_{tr}$ incurs from identifying sub-programs is directly proportional to the size of the SLD-tree generated during normal execution, i.e. the algorithm for identifying sub-programs has linear complexity (and leaves the part of the SLD-tree which is resolving literals of B with clauses of B untouched, incurring no overhead). This approach does not address non-termination issues of (recursive) programs, i.e. if executing the original program led to an infinite branch in the SLD-tree then executing the meta-interpreter instead will also yield an infinite branch. For sub-programs identified on missing answers, we still need to re-evaluate the sub-programs. If $P = \{C_1, \ldots, C_n\}$, then there are $\prod_{1 \leq i \leq n} \#literals(C_i)$ distinct sub-programs of P, i.e. the possible combinations of prefixes of P's clauses, that could be identified for retesting.

[6] Note that the meta-interpreter only keeps track of seen literals of the hypothesis, not of any literals occurring in the background knowledge.

[7] *Asserting a trace* can be done in constant time, e.g. by putting the trace in a hashmap or prepending the trace to the front of a list of failed traces.

```
1   mi_tr(true,Trace,Trace).
2   mi_tr((HeadOfBody,TailOfBody),Tr_in,Tr_out) :-
3       mi_tr(HeadOfBody,Tr_in,Tr_mid),
4       mi_tr(TailOfBody,Tr_mid,Tr_out).
5   mi_tr(mi(G,I),Tr_in,Tr_out) :-
6       clause(mi(G,J),Body),
7       Tr_head is Tr_in ∨ I ∨ J,
8       mi_tr(Body,Tr_head,Tr_out).
9   mi_tr(call(G,I),Tr_in,Tr_out) :-
10      Tr_out is Tr_in ∨ I,
11      (call(G) *-> true ; assert_failed_trace(Tr_out),fail).
```

Fig. 2 Meta-interpreter $\texttt{mi}_{tr}$. $\texttt{mi}_{tr}$ keeps track of a trace of literal indices encountered along each SLD-branch. The `∨` operator takes two bitsets and produces their union (like taking the logical *or* of two integers). `call(G)` just interpreters (complex) term `G` as an atom and evaluates it. The semantics of `G *-> Then; Else` are that if `G` ever succeeds the entire construct acts as if it were `G,Then`, otherwise it acts as if it just were `Else`. `clause(Head,Body)` unifies with any definite clause the Prolog interpreter knows about. `Body` is a cons-list of atoms which terminates in `true`

5.2 Hempel

We now introduce Hempel, an ILP system based on popper (Cropper & Morel, 2021), which supports failure explanation. Hempel tackles the LFF problem (Definition 1) using a *generate*, *test*, and *constrain* loop. Hempel maintains a logical formula (expressed as an answer set program) whose models correspond to the viable hypotheses, i.e. each model represents a unique Prolog program.

The generate stage is identical to that of popper and searches for a model of the formula which it converts to a program. In the test stage, a thus generated hypothesis H is tested on positive and negative examples. Hempel incorporates Algorithm 1, running it for each tested example. Meta-interpreter $\texttt{mi}_{tr}$ is used to determine clauses and literals that occur along branches responsible for a failure. From this information Hempel reconstructs the corresponding sub-programs. If sub-program H' is derived from a branch for a missing answer, H' gets retested, this time using standard SLD-resolution. The test stage tells the constrain stage the number of missing and incorrect answers of a (sub-)program. This determines whether its specialisations[8] and/or generalisations should be pruned. For each failed hypothesis and each of its failing sub-programs, new hypothesis constraints are added to the formula, eliminating models, thereby pruning the hypothesis space. As in general failing sub-programs need not be specialisations/generalisations of H, pruning for sub-programs is in addition to the pruning which the constrain stage already does for H in popper. Finally, Hempel loops back to the generate stage.

Smaller programs prune more effectively, which is partly why popper and Hempel search for hypotheses by increasing size[9] (in terms of number of literals). Yet there are many small programs that popper does not consider well-formed that lead to significant pruning. Consider the sub-program $H'_1 = \{\ droplast(A,B) \leftarrow empty(A)\ \}$ from Example 4. popper does not generate H'_1 as it does not consider it a well-formed hypothesis (as the head variable B

[8] popper and Hempel generate *elimination constraints* when a hypothesis entails none of the positive examples (Cropper & Morel, 2021).

[9] The other reason is to find *optimal* solutions, i.e. those with the minimal number of literals.

$$\mathcal{H}_1 = \left\{ \begin{array}{l} \mathtt{h}_1 = \{\mathtt{droplast(A,B)\text{:-}\ empty(A),tail(A,B).}\} \\ \mathtt{h}_2 = \{\mathtt{droplast(A,B)\text{:-}\ empty(A),cons(C,D,A),tail(D,B).}\} \\ \mathtt{h}_3 = \left\{ \begin{array}{l} \mathtt{droplast(A,B)\text{:-}\ tail(A,C),tail(C,B).} \\ \mathtt{droplast(A,B)\text{:-}\ tail(A,B).} \end{array} \right\} \\ \mathtt{h}_4 = \{\mathtt{droplast(A,B)\text{:-}\ empty(A),tail(A,B),head(A,C),head(B,C).}\} \\ \mathtt{h}_5 = \left\{ \begin{array}{l} \mathtt{droplast(A,B)\text{:-}\ tail(A,C),tail(C,B).} \\ \mathtt{droplast(A,B)\text{:-}\ tail(A,B),tail(B,A).} \end{array} \right\} \\ \mathtt{h}_6 = \left\{ \begin{array}{l} \mathtt{droplast(A,B)\text{:-}\ tail(A,B),empty(B).} \\ \mathtt{droplast(A,B)\text{:-}\ cons(C,D,A),droplast(D,E),cons(C,E,B).} \end{array} \right\} \\ \mathtt{h}_7 = \left\{ \begin{array}{l} \mathtt{droplast(A,B)\text{:-}\ tail(A,C),tail(C,B).} \\ \mathtt{droplast(A,B)\text{:-}\ tail(A,B).} \\ \mathtt{droplast(A,B)\text{:-}\ tail(A,C),droplast(C,B).} \end{array} \right\} \end{array} \right\}$$

Fig. 3 LFF hypothesis space considered in Example 7

does not occur in the body). Yet precisely because this sub-program has so few body literals is why it is so effective at pruning specialisations.

The following example demonstrates the loop used by HEMPEL and POPPER, and how failure explanation can lead to fewer loop iterations.

Example 7 We illustrate HEMPEL, and how it differs from POPPER, by running its loop on LFF input $(E^+, E^-, \mathcal{H}, B, C)$ from Example 1. For demonstration purposes we use the simplified hypothesis space $\mathcal{H}_1 \subseteq \mathcal{H}_C$ of Fig. 3. Our positive examples are $e_1^+ = \mathit{droplast}([1,2,3],[1,2])$ and $e_2^+ = \mathit{droplast}([1,2],[1])$, and our negative example is $e_1^- = \mathit{droplast}([1,2],[])$.

First we induce a program by a generate-test-and-constrain loop *without* failure explanation. This first sequence is representative of POPPER's execution:

1. POPPER starts by generating $\mathtt{h}_1$. $B \cup \mathtt{h}_1$ fails to entail e_1^+ and e_2^+ and correctly does not entail e_1^-. Hence only specialisations of $\mathtt{h}_1$ are pruned, namely $\mathtt{h}_4$.
2. POPPER subsequently generates $\mathtt{h}_2$. $B \cup \mathtt{h}_2$ fails to entail e_1^+ and e_2^+ and is correct on e_1^-. Hence specialisations of $\mathtt{h}_2$ are pruned, of which there are none in $\mathcal{H}_1$.
3. POPPER next generates $\mathtt{h}_3$. $B \cup \mathtt{h}_3$ does not entail the positive examples, but does entail negative example e_1^-. Hence specialisations and generalisations of $\mathtt{h}_3$ are pruned, meaning only generalisation $\mathtt{h}_7$.
4. POPPER generates $\mathtt{h}_5$. $B \cup \mathtt{h}_5$ is correct on none of the examples. Hence specialisations and generalisations of $\mathtt{h}_5$ are pruned, of which there are none in $\mathcal{H}_1$.
5. POPPER generates $\mathtt{h}_6$. $B \cup \mathtt{h}_6$ is correct on all the examples and hence $\mathtt{h}_6$ is returned.

Now we consider learning by a generate-test-and-constrain loop *with* failure explanation. The following execution sequence is representative of HEMPEL:

1. HEMPEL starts by generating $\mathtt{h}_1$. $B \cup \mathtt{h}_1$ fails to entail e_1^+ and e_2^+ and correctly does not entail e_1^-. Failure explanation identifies sub-program $\mathtt{h}_1' = \{\mathtt{droplast(A,B)\ :-\ empty(A).}\}$. $\mathtt{h}_1'$ fails in the same way as $\mathtt{h}_1$. Hence specialisations of both $\mathtt{h}_1$ and $\mathtt{h}_1'$ get pruned, namely $\mathtt{h}_2$ and $\mathtt{h}_4$.
2. HEMPEL subsequently generates $\mathtt{h}_3$. $B \cup \mathtt{h}_3$ does not entail the positive examples, but does entail negative example e_1^-. Failure explanation identifies sub-program

$h'_3 = \{\texttt{droplast(A,B)} : -\texttt{tail(A,C)}, \texttt{tail(C,B)}.\}$. $B \cup h'_3$ fails in the same way as h_3. Hence specialisations and generalisations of h_3 and h'_3 get pruned, meaning h_5 and h_7.

3. HEMPEL next generates h_6. $B \cup h_6$ is correct on all the examples and hence h_6 is returned.

The difference in these two execution sequences is illustrative of how failure explanation, by way of sub-programs, can help prune away significant parts of the hypothesis space.

6 Experiments

We claim that failure explanation can improve learning performance. Our experiments therefore aim to answer the questions:

Q1 Can failure explanation prune more programs?
Q2 Can failure explanation reduce learning times?

Note that an affirmative answer to **Q1** does not imply that **Q2** is the case, as potentially the overhead of failure explanation exceeds the benefits of the pruning it achieves.

To answer **Q1** and **Q2**, we compare HEMPEL against POPPER. The addition of failure explanation is the only difference between the systems. In each of the experiments, the settings for HEMPEL and POPPER are identical. Though control over a system's failure explanation capabilities is required to help answer **Q1** and **Q2**, we nevertheless include a comparison against state-of-the-art ILP system (Cropper & Muggleton, 2016) and the classical ILP system (Srinivasan, 2001).

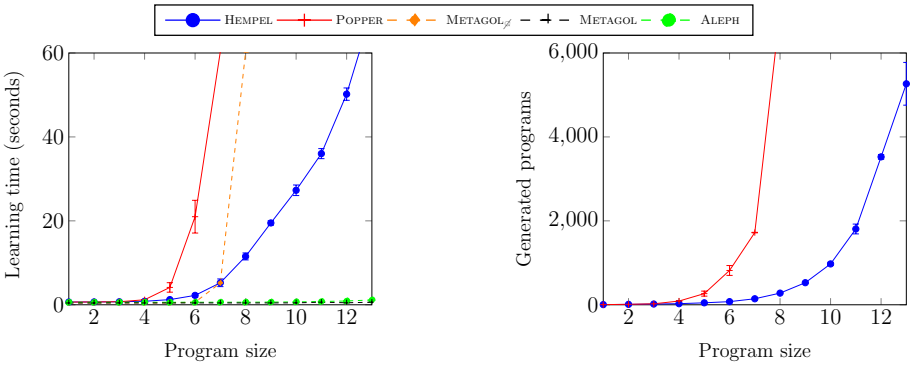
We run the experiments on a 10-core server (at 2.2GHz) with 30 gigabytes of memory (note that all the systems only run on a single CPU). When testing individual examples, we use an evaluation timeout of 2 milliseconds. In the tables reporting results, we highlight the entry with the best result per row by making it bold.

6.1 Experiment 1: robot route planning

We first evaluate the potential performance improvement of failure explanation as a function of target program size. We select a contrived setting where failure explanation ought to be very effective: a basic route planning problem. A robot resides in a grid world and can move in four directions. The robot starts in the lower left corner and needs to move to a position to its right. Unbeknownst to the robot, it has been restricted to a corridor (dimensions 14×1). In this experiment, failure explanation should determine that any strategy that moves up, down, or starts by moving left can never succeed.

Settings. An example is an atom $f(s_1, s_2)$, with start (s_1) and end (s_2) states. A state is a pair of discrete coordinates (x, y). We provide four dyadic relations as BK: *move_right*, *move_left*, *move_up*, and *move_down*, which change the state, e.g. *move_right*$((2,2),(3,2))$. We ensure that our hypotheses are forward-chained (Kaminski et al., 2019), meaning body literals modify the state one after another. We supply METAGOL with the following metarules: $P(A,B) \leftarrow Q(A,B)$ and $P(A,B) \leftarrow Q(A,C), R(C,B)$ and $P(A,B) \leftarrow Q(B,A)$.

Systems. In comparing systems, we try to ensure that hypothesis spaces are as similar as possible. For HEMPEL, POPPER and ALEPH we allow one clause with up to 13 body literals and 14 variables. METAGOL is the only system that uses predicate invention, i.e. learns clauses with invented predicate symbols. As reusing invented predicates leads



(a) Learning time. (b) Number of programs.

Fig. 4 Results of robot planning experiment.The x-axes denote the number of body literals in the solution, i.e. the number of moves required. Standard error is plotted but is always negligible for Hempel

to exponentially shorter programs for this problem, we use both Metagol and a version of Metagol where reuse of invented predicates is disabled:

$$\text{Metagol}_{\not\in}$$

.

Method. The start state is (0, 0) and the end state is $(n, 0)$, for n in $1, 2, 3, \ldots, 13$. Each trial has only one (positive) example: $f((0, 0), (n, 0))$. We measure learning times and, for Popper and Hempel, the number of generated programs. We enforce a timeout of 60 s per task. We repeat each experiment 10 times and plot the mean and standard error.

Results. Fig. 4a shows that Hempel substantially outperforms Popper in terms of learning time. The reason for the improved learning time is that Hempel generates far fewer programs, see Fig. 4b. For example, upon Hempel generating one program that starts by moving left, failure explanation determines any program whose first move is to the left is going to fail and hence all these programs get pruned.

Figure 4a also shows that Hempel outperforms

$$\text{Metagol}_{\not\in}$$

. Because

$$\text{Metagol}_{\not\in}$$

is example-driven it is effective in pruning programs that try to move out of the corridor. Yet, as explained in Sect. 2, at bigger program sizes its reconsidering of already seen programs is very costly.

Aleph and normal Metagol always find the solution, even at size 13, witin 1.5 s. For Metagol, this is due to reusing invented predicates. For example, the size 12 solution that Metagol finds has only eight body literals, versus the 12 that Hempel needs. For Aleph, the bottom-clause construction is very effective in only considering moves that are actually

allowed. However, the performance of these systems does not have bearing on whether failure explanation is effective or not.

The results from this simple experiment strongly suggest that the answer to questions **Q1** and **Q2** is yes.

6.2 Experiment 2: programming puzzles

This experiment evaluates whether failure explanation can improve performance when learning programs for recursive list problems, which other state-of-the-art ILP systems (Law, 2018; Evans & Grefenstette, 2018; Kaminski et al., 2019) struggle to solve. We show that Hempel can drastically outperform Popper, Metagol and Aleph on the same 10 problems used to evaluate Popper (Cropper & Morel, 2021), plus three additional ones: *reverse*, *odd1even2*, *sumlist*.

Settings. We provide as BK the monadic relations *empty*, *zero*, *one*, *even*, *odd*, the dyadic relations *element*, *head*, *tail*, *increment*, *decrement*, *geq*, and the triadic relations *cons*, *snoc*, *sum*. With a single fixed hypothesis space for these problems, Popper exhibits significant variance between learning times across problems (ranging from sub-second times for at least four problems to many minutes on others). To control for this variance, we select hypothesis space settings on a per problem basis, such that Popper has to do non-trivial search but can still find solutions for each problem within the timeout. See Appendix 1 for the exact settings.

Systems. For Hempel and Popper, we provide simple types and mark arguments of predicates as either input or output. For Metagol, we use the same metarules used to evaluate it against Popper (Cropper & Morel, 2021), listed in Appendix 1. Because Metagol uses metarules and invented predicates, its hypothesis space is similar but not identical to that of Hempel and Popper. For Aleph we provide mode declarations and determinations which encode the exact same information made available to Hempel. We use the same Aleph settings used to compare it against Popper (Cropper & Morel, 2021): we set the maximum variable depth and clause length to six and the number of search nodes is limited to 30000.

Method. We generate 10 positive and 10 negative examples per problem. Each example is randomly generated from lists up to length 50, whose integer elements are sampled from 1 to 100. We test on 100 positive and 100 negative randomly sampled examples, giving a default accuracy of 50%. We measure learning time, number of programs generated and predictive accuracy. We also measure the time spent in the three distinct stages of Popper and Hempel. We repeat each experiment 20 times and record the mean and standard error. We enforce a 60 s timeout.

Results.

Hempel's accuracy is at least 98% on all problems, see Table 1. Both Hempel and Popper always terminate before the timeout and score 100% on the same ten problems.

Table 1 shows the learning times in relation to the number of programs generated. Crucially, it includes the ratio of the mean of Hempel over the mean of Popper. On these 13 problems, Hempel always considers fewer hypotheses than Popper. On seven problems less than 50% of the original number of programs is considered while only on three problems over 80% is still needed.

To illustrate why failure explanation is effective, we consider the *dropk* problem. In a particular run, Popper generates 471 single-clause programs which have `f(A,B,C):-tail(A,C)` as a sub-program. On the same examples, Hempel identifies this as a failing

Table 1 Results for Hempel and Popper for Experiment 2. Left, the average number of programs generated by each system.

	Number of programs			Learning time (sec)			Accuracy	
Problem	Popper	Hempel	Ratio	Popper	Hempel	Ratio	Popper	Hempel
Dropk	2585 ± 184	121 ± 57	**0.05**	35 ± 6	4 ± 2	**0.11**	99 ± 3	99 ± 3
Sumlist	2619 ± 23	127 ± 17	**0.05**	47 ± 3	3 ± 0.7	**0.07**	100 ± 0	100 ± 0
Len	2826 ± 19	172 ± 18	**0.06**	50 ± 3	3 ± 0.4	**0.06**	100 ± 0	100 ± 0
Last	477 ± 91	63 ± 25	**0.13**	13 ± 4	2 ± 0.6	**0.15**	100 ± 0	100 ± 0
Droplast	1718 ± 117	242 ± 75	**0.14**	41 ± 8	7 ± 2	**0.18**	100 ± 0	100 ± 0
Odd1even2	1324 ± 272	289 ± 98	**0.22**	17 ± 5	4 ± 2	**0.26**	100 ± 0	100 ± 0
Member	173 ± 36	64 ± 13	**0.37**	31 ± 10	17 ± 6	**0.54**	100 ± 0	100 ± 0
Threesame	136 ± 44	72 ± 41	**0.53**	10 ± 6	5 ± 4	**0.50**	100 ± 0	100 ± 0
Finddup	1167 ± 82	653 ± 51	**0.56**	10 ± 1	7 ± 0.6	**0.66**	99 ± 1	99 ± 1
Addhead	71 ± 24	41 ± 16	**0.57**	5 ± 2	5 ± 2	**0.98**	100 ± 0	100 ± 0
Sorted	861 ± 221	712 ± 148	**0.83**	32 ± 12	28 ± 8	**0.87**	**99 ± 4**	98 ± 5
Reverse	1227 ± 424	1025 ± 435	**0.84**	29 ± 8	28 ± 10	**0.97**	100 ± 0	100 ± 0
Evens	786 ± 7	754 ± 9	**0.96**	14 ± 0.9	16 ± 0.9	1.14	100 ± 0	100 ± 0

Middle, the (corresponding) average time to find a solution. Right, the average accuracy of solutions. The error is standard error. We round values over one to the nearest integer. Values under one we round to the most significant digit

sub-program of the first hypothesis it generates and hence immediately prunes all these specialisations. In total, Popper considers 851 programs with `f(A,B,C):-tail(A,C)` as a sub-program, whilst Hempel considers just 48.

Failure explanation need not always be effective at pruning. Consider an arbitrary run of the *evens* problem: Hempel takes 354 programs before it identifies a sub-program that is not a program it has seen before. In total Hempel prunes based on just 19 sub-programs. This can be ascribed to `evens(A)` being a monadic predicate: most of the sub-programs that Hempel finds are properly formed Popper programs that Hempel (and Popper) has already seen and learnt constraints from. On a particular run of *reverse*, Hempel identifies 135 not-before-seen sub-programs. The first sub-program (of the 5th hypothesis) prunes 112 of Popper's programs, the second sub-program only 26, the third 15, and from the 5th newly identified sub-program on, which already has four literals, only about three additional programs are pruned versus Popper. By contrast, the 10th *dropk* sub-program, of size three, still prunes 59 programs relative to Popper. The effectiveness of failure-explanation-based pruning appears to be strongly dependent on whether many small sub-programs can be identified.

As seen from the ratio columns of Table 1, the number of generated programs correlates strongly with the learning time (0.96 correlation coefficient). Only on one problem is Hempel slower than Popper. Hence outfitting Popper with failure explanation can occasionally affect it negatively, but this result demonstrates that at other times the speed-up can be considerable.

Figure 5 shows the relative time spent in each stage of Hempel and popper. We can infer the overhead of failure explanation by analysing SLD-trees from this figure. All problems from *odd1even2* to *evens* have Hempel spend more time on testing than popper. On *finddup*, *reverse* and *evens*, Hempel incurs considerable testing overhead. While for *finddup*

Table 2 Selection of programming puzzles for which there was high variance in Table 1

	Number of programs			Total time (sec)		
Problem	Popper	Hempel	Ratio	Popper	Hempel	Ratio
addhead*	42 ± 0.0	25 ± 0.8	**0.58**	5 ± 0.1	4 ± 0.2	**0.87**
reverse*	770 ± 2	539 ± 7	**0.70**	20 ± 0.9	17 ± 0.9	**0.83**
sorted*	599 ± 15	477 ± 9	**0.80**	21 ± 2	18 ± 1	**0.85**

Hypotheses spaces for these problems have been pre-pruned of all programs whose size is at least as large as that of the smallest solution. Total time measures the time, in seconds, required to show there is no solution in these hypothesis spaces

this effort translates into more effective pruning constraints, for *sorted* and *evens* this is not the case. Abstracting away from the implementation of failure explanation, we see that Popper outfitted with zero-overhead failing sub-program identification would have been strictly faster.

There is considerable variance in the number of generated programs and learning times on three problems. This is in large part due to the solver that is used, (Gebser et al., 2014), yielding models, i.e. hypotheses, non-deterministically. That is, there is no fixed order in which we see hypotheses, so, by chance, Hempel and Popper can come across a solution considerably sooner in one trial than in another. As a remedy for this variance, we re-run these three problems with their hypothesis spaces restricted to programs that are strictly smaller than solutions. In this setup, Hempel and Popper always terminate precisely at the point when they have shown that none of these hypotheses can be a solution. The results, which indeed have less variance, are in Table 2.

Table 3 shows the mean accuracy and learning times of Metagol and Aleph versus Hempel. Accuracy is below 67% for Aleph on all problems, which can be ascribed to Aleph struggling to learn recursive programs. Metagol cannot find solutions for problems which require arity-three predicates (unless given hand-crafted metarules), which is why 'Not Applicable' is listed for five problems. On another four problems, Metagol returns low accuracy hypotheses. Only on two problems does Metagol outperform Hempel. In general, Hempel is the more flexible system and outperforms Metagol and Aleph.

Overall, these results strongly suggest that the answer to questions **Q1** and **Q2** is yes.

6.3 Experiment 3: IGGP and Michalski trains

For the next experiment, we evaluate Hempel on problems where solutions are larger, either because they require many clauses or many literals in a clause. We consider two settings: classification in the form of Michalski train problems (Larson & Michalski, 1977) and inductive general game playing (Cropper et al., 2020). The problems in these two settings are sufficiently hard that solutions cannot always be found in a reasonable timeframe, hence we rely on Hempel's anytime capabilities to return the best scoring hypothesis it was able to find upon a timeout.

Michalski train problems concern classifying a train as either eastbound or westbound. The features available for classifying a train's heading are its cars and their features: if a car is long or short, how many wheels the car has, how many loads and which loads it is carrying, and, finally, whether the car's roof is open, closed or flat. The target predicate, `westbound/1`, acts as our classifier and BK predicates allow for inspecting features of

Table 3 Results for Hempel, Aleph and Metagol for Experiment 2. On the left the average time to find a solution

Problem	Learning time(sec)			Accuracy		
	Hempel	Aleph	Metagol	Hempel	Aleph	Metagol
Dropk	4 ± 2	7 ± 18	N/A	**99 ± 2**	50 ± 2	N/A
Sumlist	3 ± 0.7	60 ± 0.0	N/A	**100 ± 0**	50 ± 0	N/A
Len	3 ± 0.4	60 ± 0.1	60 ± 0.1	**100 ± 0**	50 ± 0	50 ± 0
Last	2 ± 0.6	1 ± 0.1	**0.7 ± 0.7**	**100 ± 0**	50 ± 0	**100 ± 0**
Droplast	7 ± 2	60 ± 0.0	N/A	**100 ± 0**	50 ± 0	N/A
Odd1even2	4 ± 2	56 ± 9	25 ± 25	**100 ± 0**	57 ± 17	85 ± 22
Member	17 ± 6	60 ± 0.1	0.3 ± 0.0	**100 ± 0**	50 ± 0	99 ± 0
Threesame	**5 ± 4**	55 ± 11	5 ± 12	**100 ± 0**	60 ± 20	**100 ± 0**
Finddup	7 ± 0.6	1 ± 0.5	2 ± 2	99 ± 1	50 ± 1	**100 ± 0**
Sorted	28 ± 8	0.7 ± 0.1	60 ± 0.1	**98 ± 5**	65 ± 6	50 ± 0
Addhead	5 ± 2	58 ± 12	N/A	**100 ± 0**	52 ± 10	N/A
Reverse	28 ± 10	36 ± 24	N/A	**100 ± 0**	50 ± 0	N/A
Evens	16 ± 0.9	60 ± 0.1	60 ± 0.1	**100 ± 0**	50 ± 0	50 ± 0

On the right the average accuracy of solutions. The error is standard error. We round values over one to the nearest integer. Values under one we round to the most significant digit

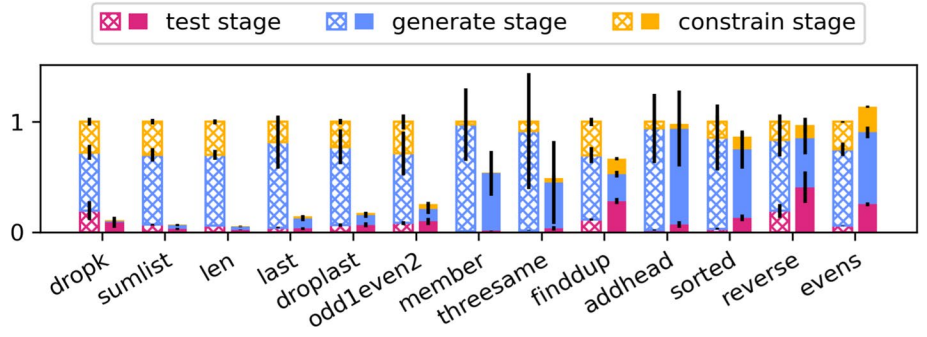


Fig. 5 Relative time spent in three stages of Popper, hatched and on the left, and Hempel, on the right. From bottom to top: testing, generating hypotheses, and imposing constraints. Mean times are shown and scaled by the total learning time of Popper. Bars are standard error

the trains to be classified. We consider the same 4 instances considered by Cropper Cropper (2022). An example of one of the higher quality hypotheses for the *trains4* problem is:

```
1 westbound(A):-has_car(A,C),roof_open(C),has_load(C,B),hexagon(B),three_load(B).
```

2 westbound(A):-has_car(A,B),has_load(B,D),diamond(D),has_load(B,C),rectangle(C).

Inductive General Game Playing concerns learning the rules of games from observations of these games being played. The goal is to synthesize a set of rules which are consistent with the *traces* generated by a game from the General Game Playing competition (Genesereth & Thielscher, 2014). The four games we consider are: *minimal-decay*, *rock, paper, scissors (rps)*, *buttons* and *coins*. In each case we learn the predicate `next`.

Settings & Systems For the trains problems, we provide two dyadic predicates, `has_car` and `has_load`, and 17 monadic predicates which encode features of cars and loads. We provide the types of arguments as well as whether they are inputs or outputs to Hempel, Popper and Aleph. We allow up to four clauses, and within each clause six variables and up to six body literals. No recursion is allowed. For Metagol we provide the same metarules as in the previous experiment. For Aleph we limit the search nodes to 30000.

For the IGGP problems we provide the monadic, dyadic and triadic predicates that encode the actions and information available to advance the game to the next state. For example, for *rps* we look for a definition of `next_score/3` given predicates `true_score/3`, `succ/2`, `does/3`, `wins/2`, `beats/2`, `different/2`.

Method We use the same instances of the problems considered by Cropper (2022). The four *trains* problems represent progressively harder instances, with *trains1* having a one clause six-literal solution and *trains4* needing 26 lals over four clauses for an optimal solution. Each trains problem has a 1000 examples available, though the distribution between positive and negative varies between tasks. We follow Cropper in that "we randomly sample the examples and split them into 80/20 train/test partitions." The four games are selected as representative instances of the larger IGGP dataset.

We measure learning time and predictive accuracy. We repeat each experiment 10 times and record the mean and standard error. We enforce a 300 s timeout.

Results Table 4 includes the results for Hempel and Popper. For the IGGP problems, we have that Hempel times out on *coins* and *buttons*, while Popper additionally times out on *minimal-decay*. On *rps* and *minimal-decay*, Hempel is able to find a solution with 100% accuracy. Note how Hempel only required around 250 programs for finding a solution for *rps* while Popper required over 10.000 programs. For *minimal-decay* Hempel needs to consider almost 2000 programs before coming across a solution while Popper cannot find one within the time limit.

In Table 5 we see the performance of Metagol and Aleph versus Hempel on the IGGP problems. As Metagol's metarules do not support arity-three predicates, we have that it is unable to find programs for *rps* and *coins*. On the other two problems, Metagol timeouts and hence achieves the default accuracy for these problems. On *coins*, both Hempel and Aleph achieve the default accuracy. On *rps*, Aleph does better than Hempel by virtue of its learning time, though Hempel still beats Metagol. On the three other games, Hempel does better than both Aleph and Metagol.

Referring back to Table 4, we see that Hempel outperforms Popper on the three more difficult trains problems. On *trains1* we see clearly the overhead of failure explanation. Even though Hempel requires less programs than Popper, testing 800 examples incurs 800 times the linear overhead of failure explanation (with regards to SLD-tree size) plus the cost of retesting failing sub-programs, of which there are more when we are dealing with bigger hypotheses. On the other three problems, the cost of failure explanation is outweighed by the pruning it achieves, with Hempel finding more accurate solutions. Not shown in Table 4, for the timeouts, Hempel spends a greater proportional of time in the test-stage than Popper, e.g. about two-thirds of the time on *trains4* versus just one-third of the time, respectively. This is likely attributable to the cost of retesting many sub-programs on the high number of examples.

From Table 5 we can see that Aleph's bottom clause construction-based learning procedure is quite effective, outperforming Hempel on all four trains problems. In turn, Hempel outperforms Metagol on all trains problems.

Table 4 Results for HEMPEL and POPPER for Experiment 3. Left, the average number of programs generated by each system

Problem	Number of programs			Learning time (sec)			Accuracy	
	POPPER	HEMPEL	Ratio	POPPER	HEMPEL	Ratio	POPPER	HEMPEL
Rps	10648 ± 38	250 ± 13	**0.02**	96 ± 2	25 ± 1	**0.26**	100 ± 0	100 ± 0
Minimal-decay	23171 ± 1538	1904 ± 76	**0.08**	300 ± 0.0	41 ± 2	**0.14**	94 ± 0	**100 ± 0**
Buttons	8022 ± 2265	1073 ± 144	0.13	300 ± 0.2	300 ± 0.0	1.00	90 ± 0	90 ± 0
Coins	9458 ± 934	535 ± 151	0.06	300 ± 0.0	300 ± 0.0	1.00	88 ± 3	85 ± 1
Trains1	28 ± 0.0	20 ± 0.3	**0.72**	**1.0 ± 0.0**	3 ± 0.0	2.99	100 ± 0	100 ± 0
Trains2	9410 ± 6144	306 ± 188	**0.03**	210 ± 137	15 ± 9	**0.07**	91 ± 5	**98 ± 2**
Trains4	11223 ± 377	1176 ± 25	0.10	300 ± 0.0	300 ± 0.0	1.00	78 ± 2	**89 ± 1**
Trains3	11278 ± 594	1315 ± 23	0.12	300 ± 0.0	300 ± 0.0	1.00	91 ± 2	**96 ± 1**

Middle, the (corresponding) average time to find a solution. Right, the average accuracy of solutions. The error is standard error. We round values over one to the nearest integer. Values under one we round to the most significant digit

Table 5 Results for HEMPEL, ALEPH and METAGOL for Experiment 3. On the left the average time to find a solution

Problem	Learning time (sec)			Accuracy		
	HEMPEL	ALEPH	METAGOL	HEMPEL	ALEPH	METAGOL
Rps	25 ± 1	**4 ± 0.1**	N/A	**100 ± 0**	**100 ± 0**	N/A
Minimal-decay	41 ± 2	4 ± 0.1	300 ± 0	**100 ± 0**	94 ± 0	88 ± 0
Buttons	300 ± 0	137 ± 4	300 ± 0	**90 ± 0**	87 ± 0	80 ± 0
Coins	300 ± 0	300 ± 0.0	N/A	**85 ± 1**	82 ± 0	N/A
Trains1	3 ± 0.0	**2 ± 0.3**	162 ± 38	100 ± 0	100 ± 0	100 ± 0
Trains2	15 ± 9	1 ± 0.1	218 ± 126	98 ± 2	**100 ± 0**	85 ± 6
Trains4	300 ± 0	215 ± 4	300 ± 0	89 ± 1	**100 ± 0**	67 ± 0
Trains3	300 ± 0	18 ± 0.9	300 ± 0	96 ± 1	**100 ± 0**	20 ± 0

On the right the average accuracy of solutions. The error is standard error. We round values over one to the nearest integer. Values under one we round to the most significant digit

Also for this experiment, the results indicate that the answer to questions **Q1** and **Q2** is yes, though with the note that larger hypotheses do appear to impact the effectiveness.

6.4 Experiment 4: string transformations

We now explore whether failure explanation can improve learning performance on real-world string transformation tasks. We hence restrict ourselves to comparing HEMPEL versus POPPER. We use a standard dataset (Lin et al., 2014; Cropper, 2019) formed of 312 tasks, each with 10 input–output pair examples. For example, task 81 has the following two input–output pairs:

Input	Output
"Alex","M",41,74,170	M
"Carly","F",32,70,155	F

Settings. As background knowledge, we give each system the monadic predicates *is_uppercase*, *is_empty*, *is_space*, *is_letter*, *is_number* and dyadic predicates *mk_uppercase*, *mk_lowercase*, *skip1*, *copyskip1*, *copy1*. For each monadic predicate we also provide a predicate that is its negation. We allow up to 3 clauses, with each clauses having a maximum of 4 body literals and up to 5 variables. We extend the test stage with a check whether the generated program is functional or not and prune for any non-functional program.

Method. The dataset has 10 positive examples for each problem. We perform cross validation by selecting 10 distinct subsets of 5 examples for each problem, using the other 5 to test. We measure learning times and number of programs generated. We enforce a timeout of 60 s per task. We repeat each experiment 10 times, once for each distinct subset, and record means and standard errors.

Results. In 132 problems both Hempel and Popper return programs which have non-zero accuracy on the test set. On 64 tasks Hempel scores better than Popper versus Popper scoring better on 20 tasks. For 54 problems at least one of Popper and Hempel finds solutions with over 90% mean accuracy. Hempel finds solutions[10] with 100% accuracy on 37 tasks, 3 more than Popper.

Figure 6 plots ratios of generated programs and learning times. Each of the 54 points represents a single problem where either Hempel or Popper scored over 90% mean accuracy. The x-axis is the ratio of number of programs that Hempel generates versus the number of programs that Popper generates. The y-value is the ratio of learning time of Hempel versus Popper. These ratios are acquired by dividing means, the mean of Hempel over that of Popper.

Looking at x-axis values, of the 54 problems plotted all require fewer programs when run with Hempel. Looking at the y-axis, the learning times of 51 problems are faster for Hempel.

Overall, these results show that, compared to Popper, Hempel typically needs fewer programs and less time to learn programs. This suggests that the answer to questions **Q1** and **Q2** is yes.

7 Conclusions

We introduced a method for using fine-grained failure explanation to derive fine-grained hypothesis space constraints. We illustrated this general method by a new SLD-based algorithm to identify failing sub-programs at the granularity of literals. We introduced an ILP system with failure explanation, Hempel, and experimentally showed that enabling failure explanation can drastically reduce hypothesis space exploration and learning times.

[10] Note that these problems are very difficult with many of them not having solutions given only our primitive BK and with the learned program restricted to defining a single predicate. Therefore, absolute performance should be ignored. The important result is the relative performance of the two systems.

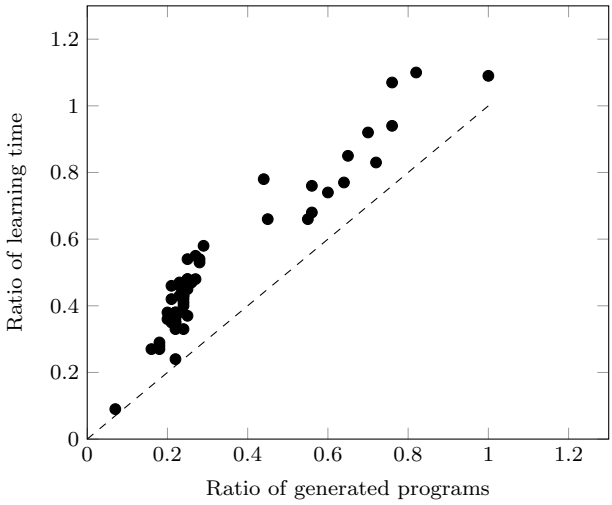


Fig. 6 String transformation results. The ratio of number of programs that HEMPEL needs versus POPPER is plotted against the ratio of learning time needed on that problem

7.1 Limitations and future work

Application of sub-program based failure explanation is not restricted to fully automated program synthesis. For example, our SLD-based algorithm could be used for explainable AI purposes, e.g. in interactive environments such as tutor systems which help teach Prolog.

While not documented here, our approach works without modification in combination with an extension of POPPER which supports predicate invention (Cropper & Morel, 2021). In an orthogonal direction, ILP noise handling methods could leverage failure explanation, e.g. by learning that the training error of a failing sub-program is as bad as the original program.

There are interesting theoretical questions to be worked out. As seen in Experiment 2, it appears that many smaller sub-programs are key to effective pruning. It should be possible to quantify the (theoretical) effectiveness of sub-program based pruning, e.g. with respect to the size of a sub-program and hypothesis space parameters such as the number of predicates. In general, future work should try to determine characteristics of problems that allow or preclude effective pruning based on failure explanation.

We require retesting of a sub-program derived from a hypothesis failing on a positive example to determine if this sub-program fails on the same example. This retesting is especially costly if there are many sub-programs, as is more likely to happen for bigger programs. Theoretical work is needed to identify cases where it follows from the original SLD-tree only having failing branches that the SLD-tree for the sub-program has no successful branch either. This would allow for eliding some of the expensive retesting that HEMPEL does.

Another major avenue for future work is leveraging fine-grained failure explanation for learning programs from logic fragments extending beyond definite programs. It should be possible to support negation-as-failure to a degree, e.g. by saying that clauses defining a predicate that occurred negated in a hypothesis are also responsible for a failure. Work on *justifications* for Answer Set Programming (Fandinno & Schulz, 2019) could be used for fine-grained pruning whilst learning ASP programs.

Although we have shown that failure explanation can drastically reduce learning times, there is still much scope for improvement. For instance, Experiment 2 had the following failing sub-program occur:

$$\{f(A, B) \leftarrow element(A, C), head(A, D), odd(C), even(C)\}$$

Straightforward reasoning tells us literal *head(A,D)* is not relevant to the failure of this sub-program. Furthermore, we should be able to lay the blame on just the last two literals.

Appendix

A Experiment 2: Metagol Settings

The following metarules were used for running Metagol in the programming puzzles experiment.

```
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(A,B).
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).
```

B Experiment 2: Hypothesis Space Settings

The following hypothesis space settings were used in the programming puzzles experiment:

Problem	max #clauses	max #literals	max #variables	sum/3	cons/3	snoc/3	head/2	tail/2	element/2	decrement/2	increment/2	qeg/2	even/1	odd/1	one/1	zero/1	empty/1
Aadd-head/2	3	7	6	x	x		x	x		x		x	x	x	x	x	x
Dropk/3	3	6	5	x	x		x	x		x	x	x	x	x	x	x	x
Droplast/2	3	6	5		x		x	x		x	x	x	x	x	x	x	x
Evens/1	2	6	5	x			x	x				x	x	x	x	x	x
Finddup/2	2	6	5	x			x	x	x	x		x	x	x	x	x	x
Last/2	3	7	6				x	x				x	x	x	x	x	x
Len/2	2	6	6	x			x	x		x		x	x	x	x	x	x
Member/2	3	7	6	x			x	x		x		x	x	x	x	x	x
Odd1even2/2	3	6	5	x			x	x				x	x	x	x	x	x
Reverse/2	3	5	5			x	x	x		x		x	x	x	x	x	x
Sorted/1	3	6	5				x	x		x		x	x	x	x	x	x
Sumlist/2	2	6	5	x	x		x	x		x		x	x	x	x	x	x
Three-same/1	3	7	6	x			x	x		x		x	x	x	x	x	x

References

Ahlgren, J., & Yuen, S.Y. (2013). Efficient program synthesis using constraint satisfaction in inductive logic programming. *JMLR*.

Blockeel, H., & De Raedt, L. (1998). Top-down induction of first-order logical decision trees. *AIJ*.

Bundy, A., & Mitrovic, B. (2016). *Reformation: A domain-independent algorithm for theory repair*. Technical report, University of Edinburgh.

Caballero, R., Riesco, A., & Silva, J. (2017). A survey of algorithmic debugging. *ACM Computing Surveys, 50*, 1–35.

Cheney, J., Chiticariu, L., & Tan, W. C. (2009). Provenance in databases: Why, how, and where. *Found. Trends Databases, 1*, 379–474.

Cropper, A. (2019). Playgol: Learning programs through play. *IJCAI*.

Cropper, A. (2022). Learning logic programs through divide, constrain, and conquer. In *AAAI*.

Cropper, A., Evans, R., & Law, M. (2020). Inductive general game playing. *Machine Learning, 109*, 1393–1434.

Cropper, A., & Morel, R. (2021). Learning programs by learning from failures. *Machine Learning, 110*, 801–856.

Cropper, A., & Morel, R. (2021). Predicate invention by learning from failures. *CoRR*. arxiv: abs/2104.14426.

Cropper, A., & Muggleton, S.H. (2016). *Metagol system*. https://github.com/metagol/metagol.

Ellis, K., Morales, L., Sablé-Meyer, M., Solar-Lezama, A., & Tenenbaum, J. (2018). Learning libraries of subroutines for neurally–guided Bayesian program induction. In *NeurIPS*.

Evans, R., & Grefenstette, E. (2018). Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research, 61*, 1–64.

Evans, R., Hernández-Orallo, J., Welbl, J., Kohli, P., & Sergot, M. (2021). Making sense of sensory input. *Artificial Intelligence, 293*, 103438.

Fandinno, Jorge, & Schulz, Claudia. (2019). Answering the "why'' in answer set programming–A survey of explanation approaches. *Theory and Practice of Logic Programmin, 19*(2), 114–203.

Feng, Y., Martins, R., Bastani, O., & Dillig, I. (2018). Program synthesis using conflict-driven learning. In *PLDI*.

Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2014). Clingo = ASP + control: Preliminary report. *CoRR*, arxiv: abs/1405.3694.

Genesereth, Michael R., & Thielscher, Michael: *General Game Playing*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, (2014).

Hocquette, C., & Muggleton, S.H. (2020). Complete bottom-up predicate invention in meta-interpretive learning. In *IJCAI*.

Kaminski, T., Eiter, T., & Inoue, K. (2019). Meta-interpretive learning using hex-programs. In *IJCAI*.

Keil, F. C., & Wilson, R. A. (2000). *Explanation and cognition*. MIT press.

Köhler, S., Ludäscher, B., & Smaragdakis, Y. (2012) Declarative datalog debugging for mere mortals. In *Datalog in academia and industry*.

Larson, J., & Michalski, R. S. (1977). Inductive inference of VL decision rules. *SIGART Newsletter, 63*, 38–44.

Law, M. (2018). *Inductive learning of answer set programs*. PhD thesis, Imperial College London, UK.

Law, M., Russo, A., Bertino, E., Broda, K., & Lobo, J. (2020). Fastlas: Scalable inductive logic programming incorporating domain-specific optimisation criteria. In *AAAI*.

Lin, D., Dechter, E., Ellis, K., Tenenbaum, J.B., & Muggleton, S. (2014). Bias reformulation for one-shot function induction. In *ECAI*.

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

Midelfart, H. (1999). A bounded search space of clausal theories. In *ILP*.

Muggleton, S. (1991). Inductive logic programming. *New Generation Computing, 8*, 295–318.
Muggleton, S. (1995). Inverse entailment and progol. *New Generation Computing, 13*, 245–286.
Nienhuys-Cheng, Shan-Hwei., & de Wolf, Ronald. (1997). *Foundations of Inductive Logic Programming*. Springer-Verlag.
Pazzani, Michael J., & Brunk, Clifford A. (1991). Detecting and correcting errors in rule-based expert systems: An integration of empirical and explanation-based learning. *Knowledge Acquisition, 3*(2), 157–173.
Plotkin, G.D. (1971). *Automatic methods of inductive inference*. PhD thesis, Edinburgh University, August.
Popper, K. R. (1963). *Conjectures and refutations: The growth of scientific knowledge*. Routledge.
De Raedt, L., & Bruynooghe, M. (1992). Interactive concept-learning and constructive induction by analogy. *Machine Learning, 8*, 107–150.
Raghothaman, M., Mendelson, J., Zhao, D., Naik, M., & Scholz, B. (20202) Provenance-guided synthesis of datalog programs. PACMPL.
Richards, Bradley L., & Mooney, Raymond J. (1995). Automated refinement of first-order horn-clause domain theories. *Machine Learning, 19*(2), 95–131.
Schüller, P., & Benz, M. (2018). Best-effort inductive logic programming via fine-grained cost-based hypothesis generation. *Machine Learning, 107*, 1141–1169.
Shapiro, E. Y. (1983). *Algorithmic program DeBugging*. MIT Press.
Marques Silva, J.P., Lynce, I., & Malik, S. (2009). Conflict-driven clause learning SAT solvers. In *Handbook of satisfiability*.
Silver, T., Allen, K.R., & Lew, A.K., Kaelbling, L.P. & Tenenbaum, J. (20202). Few-shot Bayesian imitation learning with logical program policies. In *AAAI*.
Srinivasan, A. (2001). The ALEPH manual.
Thompson, G., & Sullivan, A.K. (2020). Profl: a fault localization framework for prolog. In *ISSTA*.
Wrobel, S. (1996). First order theory refinement. *Advances in inductive logic programming, 32*, 14–33.