Learning Logic Programs by Explaining Failures

Rolf Morel, Andrew Cropper
University of Oxford
{rolf.morel, andrew.cropper}@cs.ox.ac.uk

Abstract
Scientists form hypotheses and experimentally test them. If a hypothesis fails (is refuted), scientists try to explain the failure to eliminate other hypotheses. We introduce similar explanation techniques for inductive logic programming (ILP). We build on the ILP approach learning from failures. Given a hypothesis represented as a logic program, we test it on examples. If a hypothesis fails, we identify clauses and literals responsible for the failure. By explaining failures, we can eliminate other hypotheses that will provably fail. We introduce a technique for failure explanation based on analysing SLD-trees. We experimentally evaluate failure explanation in the POPPER ILP system. Our results show that explaining failures can drastically reduce learning times.

1 Introduction
The process of forming hypotheses, testing them on data, analysing the results, and forming new hypotheses is the foundation of the scientific method [Popper, 2002]. For instance, imagine that Alice is a chemist trying to synthesise a vial of the compound octiron from substances thaum and slood. To do so, Alice can perform actions, such as fill a vial with a substance (fill(Vial,Sub)) or mix two vials (mix(V1,V2,V3)). One such hypothesis is:

\[ \text{synth}(A,B,C) \leftarrow \text{fill}(V1,A), \text{fill}(V1,B), \text{mix}(V1,V1,C) \]

This hypothesis says to synthesise compound \( C \), fill vial \( V1 \) with substance \( A \), fill vial \( V1 \) with substance \( B \), and then mix vial \( V1 \) with itself to form \( C \).

When Alice experimentally tests this hypothesis she finds that it fails. From this failure, Alice deduces that hypotheses that add more actions (i.e. literals) will also fail (C1). Alice can, however, go further and explain the failure as “vial \( V1 \) cannot be filled a second time”, which allows her to deduce that any hypothesis that includes \( \text{fill}(V1,A) \) and \( \text{fill}(V1,B) \) will fail (C2). Clearly, conclusion C2 allows Alice to eliminate more hypotheses than C1, that is, by explaining failures Alice can better form new hypotheses.

Our main contribution is to introduce similar explanation techniques for inductive program synthesis, where the goal is to machine learn computer programs from data [Shapiro, 1983]. We build on the inductive logic programming (ILP) approach learning from failures and its implementation called POPPER [Cropper and Morel, 2021]. POPPER learns logic programs by iteratively generating and testing hypotheses. When a hypothesis fails on training examples, POPPER examines the failure to learn constraints that eliminate hypotheses that will provably fail as well. A limitation of POPPER is that it only derives constraints based on entire hypotheses (as Alice does for C1) and cannot explain why a hypothesis fails (cannot reason as Alice does for C2).

We address this limitation by explaining failures. The idea is to analyse a failed hypothesis to identify sub-programs that also fail. We show that, by identifying failing sub-programs and generating constraints from them, we can eliminate more hypotheses, which can in turn improve learning performance. By the Blumer bound [1987], searching a smaller hypothesis space should result in fewer errors compared to a larger space, assuming a solution is in both spaces.

Our approach builds on algorithmic debugging [Caballero et al., 2017]. We identify sub-programs of hypotheses by analysing paths in SLD-trees. In similar work [Shapiro, 1983; Law, 2018], only entire clauses can make up these sub-programs. By contrast, we can identify literals responsible for a failure within a clause. We extend POPPER with failure explanation and experimentally show that failure explanation can significantly improve learning performance.

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.
- We prove that hypothesis space pruning based on sub-programs is more effective than pruning without them.
- We introduce an SLD-tree based technique for failure explanation. We introduce \( \text{POPPER}_X \), which adds the ability to explain failures to the POPPER ILP system.
- We experimentally show that failure explanation can drastically reduce (i) hypothesis space exploration and (ii) learning times.
2 Related Work

Inductive program synthesis systems automatically generate computer programs from specifications, typically input/output examples [Shapiro, 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]. In contrast to neural approaches, ILP techniques can generalise from few examples [Cropper et al., 2020]. Moreover, because ILP uses logic programming as a uniform representation for background knowledge (BK), examples, and hypotheses, it can be applied to arbitrary domains without the need for hand-crafted, domain-specific neural architectures. Finally, due to logic’s similarity to natural language, ILP learns comprehensible hypotheses.

Many ILP systems [Muggleton, 1995; Blockeel and Raedt, 1998; Srinivasan, 2001; Ahlgren and Yuen, 2013; Inoue et al., 2014; Schüller and Benz, 2018; Law et al., 2020] either cannot or struggle to learn recursive programs. By contrast, POPPER X can learn recursive programs and thus programs that generalise to input sizes it was not trained on. Compared to many modern ILP systems [Law, 2018; Evans and Grefenstette, 2018; Kaminski et al., 2019; Evans et al., 2021], POPPER X supports large and infinite domains, which is important when reasoning about complex data structures, such as lists. Compared to many state-of-the-art systems [Cropper and Muggleton, 2016; Evans and Grefenstette, 2018; Kaminski et al., 2019; Hocquette and Muggleton, 2020; Patschantz and Muggleton, 2021] POPPER X does not need metarules (program templates) to restrict the hypothesis space.

Algorithmic debugging [Caballero et al., 2017] explains failures in terms of sub-programs. Similarly, in databases provenance is used to explain query results [Cheney et al., 2009]. In seminal work on logic program synthesis, Shapiro [1983] analysed debugging trees to identify failing clauses. By contrast, our failure analysis reasons about concrete SLD-trees. Both ILASP3 [Law, 2018] and the remarkably similar ProSynth [Raghothaman et al., 2020] induce logic programs by precomputing every possible clause and then using a select-test-and-constrain loop. This precompute step is infeasible for clauses with many literals and restricts their failure explanation to clauses. By contrast, POPPER X does not precompute clauses and can identify clauses and literals within clauses responsible for failure.

POPPER [Cropper and Morel, 2021] learns first-order constraints, which can be likened to conflict-driven clause learning [Silva et al., 2009]. Failure explanation in POPPER X can therefore be viewed as enabling POPPER to detect smaller conflicts, yielding smaller yet more general constraints that prune more effectively.

3 Problem Setting

We now reiterate the LFF problem [Cropper and Morel, 2021] as well as the relation between constraints and failed hypotheses. We then introduce failure explanation in terms of sub-programs. We assume standard logic programming definitions [Lloyd, 2012].

3.1 Learning From Failures

To define the LFF problem, we first define predicate declarations and hypothesis constraints. LFF uses predicate declarations as a form of language bias, defining which predicate symbols may appear in a hypothesis. A predicate declaration is a ground atom of the form head_pred(p, a) or body_pred(p, a) where p is a predicate symbol of arity a. Given a set of predicate declarations D, a definite clause C is declaration consistent when two conditions hold (i) if p/m is the predicate in the head of C, then head_pred(p, m) is in D, and (ii) for all q/n predicate symbols in the body of C, body_pred(q, n) is in D.

To restrict the hypothesis space, LFF uses hypothesis constraints. Let E be a language that defines hypotheses, i.e. a meta-language. Then a hypothesis constraint is a constraint expressed in E. Let C be a set of hypothesis constraints written in a language L. A set of definite clauses H is consistent with C if, when written in L, H does not violate any constraint in C.

We now define the LFF problem, which is based on the ILP learning from entailment setting [Raedt, 2008]:

**Definition 3.1 (LFF input).** A LFF input is a tuple \((E^+, E^-, B, D, C)\) where \(E^+\) and \(E^-\) are sets of ground atoms denoting positive and negative examples respectively; \(B\) is a Horn program denoting background knowledge; \(D\) is a set of predicate declarations; and \(C\) is a set of hypothesis constraints.

A definite program is a hypothesis when it is consistent with both \(D\) and \(C\). We denote the set of such hypotheses as \(H_{D,C}\).

We define a LFF solution:

**Definition 3.2 (LFF solution).** Given an input tuple \((E^+, E^-, B, D, C)\), a hypothesis \(H \in H_{D,C}\) is a solution when \(H\) is complete \((\forall e \in E^+, B \cup H \models e)\) and consistent \((\forall e \in E^-, B \cup H \not\models e)\).

If a hypothesis is not a solution then it is a failure and a failed 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 worked example of LFF is included in Appendix A.

3.2 Specialisation and Generalisation Constraints

The key idea of LFF is to learn constraints from failed hypotheses. Cropper and Morel [2021] introduce constraints based on subsumption [Plotkin, 1971] and theory subsumption [Middelfrt, 1999]. A clause \(C_1\) subsumes a clause \(C_2\) if and only if there exists a substitution \(\theta\) such that \(C_1\theta \subseteq C_2\). 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 \(C_1\theta \subseteq C_2\). Subsumption implies entailment, i.e. if \(T_1 \preceq T_2\) then \(T_1 \models T_2\). A clausal theory \(T_1\) is a specialisation of a clausal theory \(T_2\) if and only if \(T_2 \preceq T_1\). A clausal theory \(T_1\) is a generalisation of a clausal theory \(T_2\) if and only if \(T_1 \preceq T_2\).

Hypothesis constraints prune the hypothesis space. Generalisation constraints only prune generalisations of inconsistent hypotheses. Specialisation constraints only prune specialisations of incomplete hypotheses. Generalisation and specialisation constraints are sound in that they do not prune solutions [Cropper and Morel, 2021].
3.3 Missing and Incorrect Answers

We follow Shapiro [1983] in identifying examples as responsible for the failure of a hypothesis \( H \) given background knowledge \( B \). 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 each specialisation of \( H \) has \( e^+ \) as a missing answer, so the specialisations of \( H \) are incomplete and can be eliminated. If \( H \) has an incorrect answer \( e^- \), then each generalisation of \( H \) has \( e^- \) as an incorrect answer, so the generalisations of \( H \) are inconsistent and can be eliminated.

Example 1 (Missing answers and specialisations). Consider the following \( \text{droplast} \) hypothesis:

\[
H_1 = \{ \text{droplast}(A,B) \leftarrow \text{empty}(A), \text{tail}(A,B) \}
\]

Both \( \text{droplast}([1, 2], [1]) \) and \( \text{droplast}([1, 2], [1]) \) 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 2 (Incorrect answers and generalisations). Consider the hypothesis \( H_2: \)

\[
H_2 = \{ \text{droplast}(A,B) \leftarrow \text{tail}(A,C), \text{tail}(C,B) \}, \text{droplast}(A,B) \leftarrow \text{tail}(A,B) \}
\]

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

3.4 Failing Sub-programs

We now extend LFF by 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 3 (Explain incompleteness). Consider the positive example \( e^+ = \text{droplast}([1, 2], [1]) \) and the previously defined hypothesis \( H_1 \). An explanation for why \( H_1 \) does not entail \( e^+ \) is that \( \text{empty}([1, 2]) \) fails. It follows that the program \( H_1^* = \{ \text{droplast}(A,B) \leftarrow \text{empty}(A) \} \) has \( e^+ \) as a missing answer and is incomplete, so we can prune all specialisations of it.

Example 4 (Explain inconsistency). Consider the negative example \( e^- = \text{droplast}([1, 2], []) \) and the previously defined hypothesis \( H_2 \). The first clause of \( H_2 \) always entails \( e^- \) regardless of other clauses in the hypothesis. It follows that the program \( H_2^* = \{ \text{droplast}(A,B) \leftarrow \text{tail}(A,C), \text{tail}(C,B) \} \) has \( e^- \) as an incorrect answer and is inconsistent, so we can prune all generalisations of it.

We now define a sub-program:

Definition 3.3 (Sub-program). The definite program \( P \) is a sub-program of the definite program \( Q \) if and only if either:

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

In functional program synthesis, sub-programs (sometimes called partial programs) are typically defined by leaving out nodes in the parse tree of the original program [Feng et al., 2018]. Our definition generalises this idea by allowing for arbitrary ordering of clauses and literals.

We now define the failing sub-programs problem:

Definition 3.4 (Failing sub-programs problem). Given the 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 entail an example of \( E^- \).

By definition, a failing sub-program is incomplete and/or inconsistent, so, by Section 3.2, we can always prune specialisations and/or generalisations of a failing sub-program.

Remark 1 (Undecidability). The failing sub-programs problem is undecidable in general as deciding entailment can be reduced to it.

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. Specialisation and generalisation constraints for \( P \) can always achieve additional pruning versus those only for \( H \).

Proof. Suppose \( H \) is a specialisation of \( P \). If \( P \) is incomplete, then among the specialisations of \( P \), which are all prunable, is \( H \) and its specialisations. If \( P \) is inconsistent, \( P \)’s generalisations do not completely overlap with \( H \)’s generalisations and specialisations (using that \( P \neq H \)). Hence, pruning \( P \)’s generalisations prunes programs not pruned by \( H \). The case where \( H \) is a generalisation of \( P \) is analogous. In the remaining case, where \( H \) and \( P \) are not related by subsumption, it is immediate that the constraints derived for \( P \) prune a distinct part of the hypothesis space.

4 Implementing Failure Explanation

We now describe our failure explanation technique, which identifies sub-programs by identifying both clauses and literals within clauses responsible for failure. Subsequently we summarise the POPPER ILP system before introducing our extension of it: POPPER\( _\lambda \).

4.1 SLD-trees and Sub-programs

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 and Wolf, 1997]. Exploiting their granularity, we analyse SLD-trees to address the failing sub-programs problem, only identifying a subset of them.

A branch in a SLD-tree is a path from the root \( \text{goal} \) 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.

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

Suppose $e^-$ is an incorrect answer for hypothesis $H$. Then the SLD-tree for $B \cup H' \cup \{ \neg e^- \}$, with $\neg e^-$ as root, has a successful branch $\lambda$. The literals of $H$ necessary for this branch are also present in $\lambda$-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 $B \cup H \cup \{ \neg e^+ \}$, with $\neg e^+$ as root, and $\lambda'$ be any failing branch of $T$. The literals of $H$ in $\lambda'$ are also present in $\lambda'$-sub-program $H''$. This is however insufficient for concluding that the SLD-tree for $H''$ has no successful branch. Hence it is not immediate that $e^+$ is a missing answer for $H''$. In case that $H''$ is a specialisation of $H$ we can conclude that $e^+$ is a missing answer.

4.2 POPPER

POPPER tackles the LFF problem (Definition 3.1) using a generate, test, and constrain loop. A logical formula is constructed and maintained whose models correspond to Prolog programs. The first stage is to generate a model and convert it to a program. The program is tested on all positive and negative examples. The number of missing and incorrect answers determine whether specialisations and/or generalisations can be pruned. When a hypothesis fails, new hypothesis constraints (Section 3.2) are added to the formula, which eliminates models and thus prunes the hypothesis space. POPPER then loops back to the generate stage.

Smaller programs prune more effectively, which is partly why POPPER searches for hypotheses by their size (number of literals)$^2$. Yet there are many small programs that POPPER does not consider well-formed that achieve significant, sound pruning. Consider the sub-program $H'_1 = \{ droplast(A,B) \leftarrow empty(A) \}$ from Example 3. POPPER does not generate $H'_1$ as it does not consider it a well-formed hypothesis (as the head variable $B$ 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.

4.3 POPPER$^X$

We now introduce POPPER$^X$, which extends POPPER with SLD-based failure explanation. Like POPPER, any generated hypothesis $H$ is tested on the examples. However, additionally, for each tested example we obtain the selected atoms on each branch of the example’s SLD-tree, which correspond to sub-programs of $H$. As shown, sub-programs derived from incorrect answers have the same incorrect answers. For each such identified inconsistent sub-program $H'$ of $H$ we tell the constrain stage to prune generalisations of $H'$. Sub-programs derived from missing answers are tested, now without obtaining their SLD-trees. If a sub-program $H''$ of $H$ is incomplete we inform the constrain stage to prune specialisations$^3$ of $H''$.

POPPER generates elimination constraints when a hypothesis entails none of the positive examples [Cropper and Morel, 2021].

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

As in POPPER, we prune by elimination constraints if no positive examples are entailed.

POPPER$^X$ tackles the LFF problem (Definition 3.1) using a generate, test, and constrain stage to prune specialisations.

5 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?

A positive answer to Q1 does not imply a positive answer for Q2 because of the potential overhead of failure explanation. Identifying sub-programs requires computational effort and the additional constraints could potentially overwhelm a learner. For example, as well as identifying sub-programs, POPPER$^X$ needs to derive more constraints, ground them, and have a solver reason over them. These operations are all costly.

To answer Q1 and Q2, we compare POPPER$^X$ against POPPER. The addition of failure explanation is the only difference between the systems and in all the experiments the settings for POPPER$^X$ and POPPER are identical. We do not compare against other state-of-the-art ILP systems, such as Metagol [Cropper and Muggleton, 2016] and ILASP3 [Law, 2018] because such a comparison cannot help us answer Q1 and Q2. Moreover, POPPER$^X$ has been shown to outperform these two systems on problems similar to the ones we consider [Cropper and Morel, 2021].

We run the experiments on a 10-core server (at 2.2GHz) with 30 gigabytes of memory (note that POPPER and POPPER$^X$ only run on a single CPU). When testing individual examples, we use an evaluation timeout of 33 milliseconds.

5.1 Experiment 1: Robot Planning

The goal of this experiment is to evaluate whether failure explanation can improve performance when progressively increasing the size of the target program. We therefore need a problem where we can vary the program size. We consider a robot strategy learning problem. There is a robot that can move in four directions in a grid world, which we restrict to being the SDR (dimensions $1 \times 10$). The robot starts in the lower left corner and needs to move to a position to its right. In this experiment, failure explanation should determine that any strategy that moves up, down, or left can never succeed and thus can never appear in a solution.

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 allow one clause with up to 10 body literals and 11 variables. We use hypothesis constraints to ensure this clause is forward-chained [Kaminski et al., 2019], which means body literals modify the state one after another.

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

**Results.** Figure 1a shows that POPPER\textsubscript{X} substantially outperforms POPPER in terms of learning time. Whereas POPPER\textsubscript{X} needs around 80 seconds to find a 10 move solution, POPPER exceeds the 10 minute timeout when looking for a six move solution. The reason for the improved performance is that POPPER\textsubscript{X} generates far fewer programs, as failure explanation will, for example, prune all programs whose first move is to the left. For instance, to find a five literal solution, POPPER generates 1300 programs, whereas POPPER\textsubscript{X} only generates 62. When looking for a 10 move solution, POPPER\textsubscript{X} only generates 1404 programs in a hypothesis space of 1.4 million programs. These results show that, compared to POPPER, POPPER\textsubscript{X} generates substantially fewer programs and requires less learning time. The results from this experiment strongly suggest that the answer to questions Q1 and Q2 is yes.

![Figure 1](image1.png)

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

Figure 1: Results of robot planning experiment. The x-axes denote the number of body literals in the solution, i.e. the number of moves required.

### 5.2 Experiment 2: String Transformations

We now explore whether failure explanation can improve learning performance on real-world string transformation tasks. We use a standard dataset [Lin et al., 2014; Cropper, 2019] formed of 312 tasks, each with 10 input-output pair examples. For instance, 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 BK, we give each system the monadic predicates is\textsubscript{upper}case, is\textsubscript{empty}, is\textsubscript{space}, is\textsubscript{letter}, is\textsubscript{number} and dyadic predicates mk\textsubscript{upper}case, mk\textsubscript{lower}case, skip\textsubscript{l}, copy\textsubscript{skipl}, copy\textsubscript{l}. For each monadic predicate we also provide a predicate that is its negation. We allow up to 3 clauses with 4 body literals and up to 5 variables per clause.

**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 120 seconds per task. We repeat each experiment 10 times, once for each distinct subset, and record means and standard errors.

![Figure 2](image2.png)

Figure 2: String transformation results. The ratio of number of programs that POPPER\textsubscript{X} needs versus POPPER is plotted against the ratio of learning time needed on that problem.

**Results.** For 52 problems both POPPER and POPPER\textsubscript{X} find solutions\textsuperscript{4}. On 11 tasks POPPER timeouts, and on 7 of these in all trials. POPPER\textsubscript{X} finds solutions on these same 11 tasks, with timeouts in some trials on only 6 tasks. As relational solutions are allowed, many solutions are not ideal, e.g. allowing for optionally copying over a character.

Looking at Figure 2 plots ratios of generated programs and learning times. Each point represents a single problem. The x-axis is the ratio of programs that POPPER\textsubscript{X} generates versus the number of programs that POPPER generates. The y-value is the ratio of learning time of POPPER\textsubscript{X} versus POPPER. These ratios are acquired by dividing means, the mean of POPPER\textsubscript{X} over that of POPPER.

Looking at x-axis values, of the 52 problems plotted 50 require fewer programs when run with POPPER\textsubscript{X}. Looking at the y-axis, the learning times of 51 problems are faster on POPPER\textsubscript{X}. Note that either failure explanation is very effective or its influence is rather limited, which we explore more in the next experiment.

Overall, these results show that, compared to POPPER, POPPER\textsubscript{X} almost always needs fewer programs and less time to learn programs. This suggests that the answer to questions Q1 and Q2 is yes.

### 5.3 Experiment 3: Programming Puzzles

This experiment evaluates whether failure explanation can improve performance when learning programs for recursive list problems, which are notoriously difficult for ILP systems. Indeed, other state-of-the-art ILP system [Law, 2018; Evans and Grefenstette, 2018; Kaminski et al., 2019] struggle to solve these problems. We use the same 10 problems used by [Cropper and Morrel, 2021] to show that POPPER drastically outperforms METAGOL [Cropper and Muggleton, 2016] and ALEPH [Srinivasan, 2001]. The 10 tasks include a mix of monadic (e.g. evens and sorted), dyadic (e.g. drop\textsubscript{last} and find\textsubscript{dup}), and triadic (drop\textsubscript{k}) target predicates. Some problems are functional (e.g. last and len) and some are relational (e.g. find\textsubscript{dup} and member).

\textsuperscript{4}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.
**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 relation cons. We provide simple types and mark the arguments of predicates as either input or output. We allow up to two clauses with five body literals and up to five variables per clause.

**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 a 1000 positive and a 1000 negative randomly sampled examples. We measure overall learning time, number of programs generated, and predictive accuracy. We also measure the time spent in the three distinct stages of POPPER and POPPER.X. We repeat each experiment 25 times and record the mean and standard error.

We provide as BK the monadic relations empty, zero, one, even, odd, the dyadic relations element, head, tail, increment, decrement, geq, and the triadic relation cons. We provide simple types and mark the arguments of predicates as either input or output. We allow up to two clauses with five body literals and up to five variables per clause.

**Results.** Both systems are equally accurate, except on sorted where POPPER scores 98% and POPPER.X 99%. Accuracy is 98% on dropk and 99% on both finddup and threesome. All other problems have 100% accuracy.

Table 1 shows the learning times in relation to the number of programs generated. Crucially, it includes the ratio of the mean of POPPER.X over the mean of POPPER. On these 10 problems, POPPER.X always considers fewer hypotheses than POPPER. Only on three problems is over 90% of the original number of programs considered. On the len problem, POPPER.X only needs to consider 10% of the number of hypotheses.

As seen from the ratio columns, the number of generated programs correlates strongly with the learning time (0.96 correlation coefficient). Only on three problems is POPPER.X slightly slower than POPPER. Hence POPPER can be negatively impacted by failure explanation, however, when POPPER.X is faster, the speed-up can be considerable.

To illustrate how failure explanation can drastically improve pruning, consider the following hypothesis that POPPER.X considers in the len problem:

\[
\text{f}(A,B) : - \text{element}(A,D), \text{odd}(D), \text{even}(D), \text{tail}(A,C), \text{element}(C,B). 
\]

Failure explanation identifies the failing sub-program:

\[
\text{f}(A,B) : - \text{element}(A,D), \text{odd}(D), \text{even}(D). 
\]

As should be hopefully clear, generating constraints from this smaller failing program, which is not a POPPER hypothesis, leads to far more effective pruning.

**Limitations.** We have shown that identifying failing sub-programs will lead to more constraints and thus more pruning of the hypothesis space (Theorem 1), which our experiments empirically confirm. We have not, however, quantified the theoretical effectiveness of pruning by sub-programs, nor have we evaluated improvements in predictive accuracy, which are implied by the Blumer bound [Blumer et al., 1987]. Future work should address both of these limitations. Although we have shown that failure explanation can drastically reduce learning times, we can still significantly improve our approach. For instance, reconsider the failing sub-program \(f(A,B) : - \text{element}(A,D), \text{odd}(D), \text{even}(D)\) from Section 5.3. We should be able to identify that the two literals odd(D) and even(D) can never both hold in the body of a clause, which would allow us to prune more programs. Finally, in future work, we will want to explore whether our inherently interpretable failure explanations can aid explainable AI and ultra-strong machine learning [Michie, 1988; Muggleton et al., 2018].
References

[Ahlgren and Yuen, 2013] John Ahlgren and Shiu Yin Yuen. Efficient program synthesis using constraint satisfaction in inductive logic programming. *JMLR*, 2013.

[Blockeel and Raedt, 1998] Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. *AIJ*, 1998.

[Blumer et al., 1987] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Occam’s razor. *Inf. Process. Lett.*, 1987.

[Caballero et al., 2017] Rafael Caballero, Adrián Riesco, and Josep Silva. A survey of algorithmic debugging. *ACM Comput. Surv.*, 2017.

[Cheney et al., 2009] James Cheney, Laura Chiticiariu, and Wang Chiew Tan. Provenance in databases: Why, how, and where. *Found. Trends Databases*, 2009.

[Cropper and Morel, 2021] Andrew Cropper and Rolf Morel. Learning programs by learning from failures. *Machine Learning*, 2021. To appear.

[Cropper and Muggleton, 2016] Andrew Cropper and Stephen H. Muggleton. Learning higher-order logic programs through abstraction and invention. In *IJCAI*, 2016.

[Cropper et al., 2020] Andrew Cropper, Sebastijan Dumanic, and Stephen H. Muggleton. Turning 30: New ideas in inductive logic programming. In *IJCAI*, 2020.

[Cropper, 2019] Andrew Cropper. Playgol: Learning programs through play. *IJCAI*, 2019.

[Ellis et al., 2018] 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.

[Evans and Grefenstette, 2018] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *JAIR*, 2018.

[Evans et al., 2021] Richard Evans, José Hernández-Orallo, Johannes Welbl, Pushmeet Kohli, and Marek Sgoret. Making sense of sensory input. *Artificial Intelligence*, 2021.

[Feng et al., 2018] Yu Feng, Ruben Martins, Olsbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. In *PLDI*, 2018.

[Hocquette and Muggleton, 2020] Céline Hocquette and Stephen H. Muggleton. Complete bottom-up predicate invention in meta-interpretive learning. In *IJCAI*, 2020.

[Inoue et al., 2014] Katsumi Inoue, Tony Ribeiro, and Chiaki Sakama. Learning from interpretation transition. *Machine Learning*, 2014.

[Kaminski et al., 2019] Tobias Kaminski, Thomas Eiter, and Katsumi Inoue. Meta-interpretive learning using hex-programs. In *IJCAI*, 2019.

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

[Law, 2018] Mark Law. *Inductive learning of answer set programs*. PhD thesis, Imperial College London, UK, 2018.

[Lin et al., 2014] Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua B. Tenenbaum, and Stephen Muggleton. Bias reformulation for one-shot function induction. In *ECAI*, 2014.

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

[Michie, 1988] Donald Michie. Machine learning in the next five years. In *EWIS*, 1988.

[Midelfart, 1999] Herman Midelfart. A bounded search space of clausal theories. In *ILP*, 1999.

[Muggleton et al., 2018] S.H. Muggleton, U. Schmid, C. Zeller, A. Tamaddoni-Nezhad, and T. Besold. Ultra-strong machine learning - comprehensibility of programs learned with ILP. *Machine Learning*, 2018.

[Muggleton, 1991] Stephen Muggleton. Inductive logic programming. *New Generation Comput.*, 1991.

[Muggleton, 1995] Stephen Muggleton. Inverse entailment and progol. *New Generation Comput.*, 1995.

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

[Patsantzis and Muggleton, 2021] S. Patsantzis and Stephen Muggleton. Inverse entailment and progol. *New Generation Comput.*, 2021.

[Popper, 2002] K.R. Popper. *Conjectures and Refutations: The Growth of Scientific Knowledge*. Routledge, 2002.

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

[Raghothaman et al., 2020] Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. Provenance-guided synthesis of datalog programs. *PACMPL*, 2020.

[Schüller and Benz, 2018] Peter Schüller and Mishal Benz. Best-effort inductive logic programming via fine-grained cost-based hypothesis generation. *Machine Learning*, 2018.

[Shapiro, 1983] Ehud Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, USA, 1983.

[Silva et al., 2009] João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*. 2009.

[Silver et al., 2020] Tom Silver, Kelsey R. Allen, Alex K. Lew, Leslie Pack Kaelbling, and Josh Tenenbaum. Few-shot bayesian imitation learning with logical program policies. In *AAAI*, 2020.

[Srinivasan, 2001] A. Srinivasan. The ALEPH manual. 2001.
A Appendix: LFF Example

Example 5. To illustrate LFF, consider learning a droplast/2 program. Suppose our predicate declarations D are head, pred(droplast,2), denoting that we want to learn a droplast/2 relation, and body, pred(empty,1), body, pred(head,2), body, pred(tail,2), and body, pred(cons,3). Suitable definitions for the provided body predicate declarations constitute our background knowledge B. To allow for learning a recursive program, we also supply the predicate declaration body, pred(droplast,2). Let $e_1^+$ = droplast([1, 2, 3], [1, 2]), $e_2^+$ = droplast([1, 2], [1]), and $e_1^-$ = droplast([1, 2], []). Then $E^+ = \{ e_1^+, e_2^+ \}$ and $E^- = \{ e_1^- \}$ are the positive and negative examples, respectively. Our initial set of hypothesis constraints C only ensure that hypotheses are well-formed, e.g. that each variable that occurs in the head of a rule occurs in the rule’s body.

We now consider learning a solution for LFF input $(E^+, E^-, B, D, C)$, where, for demonstration purposes, we use the simplified hypothesis space $\mathcal{H}_1 \subseteq \mathcal{H}_{D, C}$ of figure 4. The order the hypotheses are considered in is by their number of literals. Pruning is achieved by adding additional hypothesis constraints. First we learn by a generate-test-and-constrain loop without failure explanation. This first sequence is representative of POPPER’s execution:

1. POPPER starts by generating $h_1$. $B \cup h_1$ fails to entail $e_1^+$ and $e_2^+$ and correctly does not entail $e_1^-$. Hence only specialisations of $h_1$ get pruned, namely $h_4$.
2. POPPER subsequently generates $h_2$. $B \cup h_2$ fails to entail $e_1^+$ and $e_2^+$ and is correct on $e_1^-$. Hence specialisations of $h_2$ get pruned, of which there are none in $\mathcal{H}_1$.
3. POPPER subsequently generates $h_3$. $B \cup h_3$ does not entail the positive examples, but does entail negative example $e_1^-$. Hence specialisations and generalisations of $h_3$ get pruned, meaning only generalisation $h_7$.
4. POPPER subsequently generates $h_5$. $B \cup h_5$ is correct on none of the examples. Hence specialisations and generalisations of $h_5$ get pruned, of which there are none in $\mathcal{H}_1$.

5. POPPER subsequently generates $h_6$. $B \cup h_6$ is correct on all the examples and hence is returned.

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

1. POPPERX starts by generating $h_1$. $B \cup h_1$ fails to entail $e_1^+$ and $e_2^+$ and correctly does not entail $e_1^-$. Failure explanation identifies sub-program $h_1' = \{ \text{droplast}(A,B) :- \text{empty}(A). \}$. $h_1'$ fails in the same way as $h_1$. Hence specialisations of both $h_1$ and $h_1'$ get pruned, namely $h_2$ and $h_4$.
2. POPPERX subsequently generates $h_3$. $B \cup h_3$ does not entail the positive examples, but does entail negative example $e_1^-$. Failure explanation identifies sub-program $h_3' = \{ \text{droplast}(A,B) :- \text{tail}(A,C), \text{tail}(C,B). \}$. $B \cup h_3'$ 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. POPPERX subsequently generates $h_6$. $B \cup h_6$ is correct on all the examples and hence is returned.

The difference in these two execution sequences is illustrative of how failure explanation can help prune away significant parts of the hypothesis space.

\[
\mathcal{H}_1 = \begin{cases} 
  h_1 = \{ \text{droplast}(A,B) :- \text{empty}(A),\text{tail}(A,B). \} \\
  h_2 = \{ \text{droplast}(A,B) :- \text{empty}(A),\text{cons}(C,D,A),\text{tail}(D,B). \} \\
  h_3 = \{ \text{droplast}(A,B) :- \text{tail}(A,C),\text{tail}(C,B). \} \\
  h_4 = \{ \text{droplast}(A,B) :- \text{tail}(A,B). \} \\
  h_5 = \{ \text{droplast}(A,B) :- \text{tail}(A,C),\text{tail}(C,B). \} \\
  h_6 = \{ \text{droplast}(A,B) :- \text{tail}(A,B),\text{empty}(A). \} \\
  h_7 = \{ \text{droplast}(A,B) :- \text{tail}(A,C),\text{tail}(C,B). \} \\
  h_8 = \{ \text{droplast}(A,B) :- \text{tail}(A,C),\text{droplast}(C,B). \} \\
  h_9 = \{ \text{droplast}(A,B) :- \text{tail}(A,C),\text{tail}(C,B). \} \\
  h_{10} = \{ \text{droplast}(A,B) :- \text{tail}(A,C),\text{tail}(C,B). \} \\
\end{cases}
\]

Figure 4: LFF hypothesis space considered in Example 5.