# Honey, I Shrunk the Hypothesis Space (Through Logical Preprocessing)

ANDREW CROPPER*, ELLIS Institute, Finland and University of Helsinki, Finland
FILIPE GOUVEIA, LASIGE, Informática, Faculdade de Ciências, Universidade de Lisboa, Portugal
DAVID M. CERNA, Czech Academy of Sciences Institute of Computer Science, Czechia and Dynatrace Research, Austria

Inductive logic programming (ILP) is a form of logical machine learning. The goal is to search a hypothesis space for a hypothesis that generalises training examples and background knowledge. We introduce an approach that *shrinks* the hypothesis space before an ILP system searches it. Our approach uses background knowledge to find rules that cannot be in an optimal hypothesis regardless of the training examples. For instance, our approach discovers relationships such as *even numbers cannot be odd* and *prime numbers greater than 2 are odd*. It then removes violating rules from the hypothesis space. We implement our approach using answer set programming and use it to shrink the hypothesis space of a constraint-based ILP system. Our experiments on multiple domains, including visual reasoning and game playing, show that our approach can substantially reduce learning times whilst maintaining predictive accuracies. For instance, given just 10 seconds of preprocessing time, our approach can reduce learning times from over 10 hours to only 2 seconds.

**JAIR Track:** Constraint Programming and Machine Learning

**JAIR Associate Editor:** Christian Bessiere

**JAIR Reference Format:**
Andrew Cropper, Filipe Gouveia, and David M. Cerna. 2026. Honey, I Shrunk the Hypothesis Space (Through Logical Preprocessing). *Journal of Artificial Intelligence Research* 85, Article 48 (April 2026), 42 pages. DOI: 10.1613/jair.1.21708

## 1 Introduction

Inductive logic programming (ILP) (Cropper and Dumancic 2022; S. Muggleton 1991) is a form of logical machine learning. The goal is to search a hypothesis space for a hypothesis that generalises training examples and background knowledge (BK). The distinguishing feature of ILP is that it uses logical rules to represent hypotheses, examples, and BK.

As with all forms of machine learning, a major challenge in ILP is searching vast hypothesis spaces. To illustrate this challenge, consider a learning task where we want to generalise examples of an arbitrary relation *h*. Assume we can build rules using the unary relations *odd*, *even*, and *int* and the binary relations *head*, *tail*, *len*, *lt*, and *succ*. The *rule space* is the set of all possible rules and contains rules such as:

*Corresponding Author.

Authors' Contact Information: Andrew Cropper, andrew.cropper@helsinki.fi, ORCID: 0000-0002-4543-7199, ELLIS Institute, Finland and University of Helsinki, Finland; Filipe Gouveia, ORCID: 0000-0003-1852-2782, jfgouveia@ciencias.ulisboa.pt, LASIGE, Informática, Faculdade de Ciências, Universidade de Lisboa, Portugal; David M. Cerna, ORCID: 0000-0002-6352-603X, dcerna@cs.cas.cz, david.cerna@dynatrace.com, Czech Academy of Sciences Institute of Computer Science, Czechia and Dynatrace Research, Austria.


DOI: 10.1613/jair.1.21708

$r_1$ = $h \leftarrow$ *len(A,B)*
$r_2$ = $h \leftarrow$ *tail(A,A)*
$r_3$ = $h \leftarrow$ *tail(A,B), tail(B,A)*
$r_4$ = $h \leftarrow$ *head(A,B), head(A,C)*
$r_5$ = $h \leftarrow$ *tail(A,B), tail(B,C), tail(A,C)*
$r_6$ = $h \leftarrow$ *tail(A,A), head(A,B), odd(B)*
$r_7$ = $h \leftarrow$ *head(A,B), odd(B), even(B)*
$r_8$ = $h \leftarrow$ *head(A,B), int(B), odd(B)*
$r_9$ = $h \leftarrow$ *head(A,B), succ(B,C), succ(C,D), lt(B,D)*

The hypothesis space is the powerset of the rule space, so it can be enormous. In this simple scenario, if we allow rules to contain at most four literals and four variables, there are at least 3,183,545 possible rules and thus $2^{3,183,545}$ hypotheses.

Most research tackles this challenge by developing techniques to efficiently search large hypothesis spaces (Ahlgren and Yuen 2013; Blockeel and De Raedt 1998; Cropper and Morel 2021; S. Muggleton 1995; S. H. Muggleton et al. 2015; Quinlan 1990; Srinivasan 2001; Zeng et al. 2014). In other words, most research assumes a fixed hypothesis space and focuses on efficiently searching that space.

Rather than develop a novel search algorithm, we introduce a way to *shrink* the hypothesis space before giving it to an ILP system. The idea is to use the given BK to find rules that cannot be in an optimal hypothesis regardless of the concept we want to learn, where an optimal hypothesis minimises a cost function on the training data, such as finding a minimal description length hypothesis (Hocquette, Niskanen, Järvisalo, et al. 2024). We call such rules *pointless rules*.

To illustrate our idea, consider the previous scenario. Assume, for simplicity, that the rule space contains only those nine rules, rather than all 3,183,545 rules. Also assume that we have BK with only the facts:

```
head(ijcai,i) head(ecai,e) head(cai,c)
tail(ijcai,jcai) tail(jcai,cai) tail(ecai,cai) tail(cai,ai) tail(ai,i)
len(ijcai,5) len(jcai,4) len(cai,3) len(ecai,4) len(ai,2) len(i,1)
int(1) int(2) int(3) int(4)
succ(1,2) succ(2,3) succ(3,4)
even(2) even(4)
odd(1) odd(3)
lt(1,2) lt(1,3) lt(1,4) lt(2,3) lt(2,4) lt(3,4)
```

We use this BK to find four types of pointless rules: *unsatisfiable*, *implication reducible*, *recall reducible*, and *singleton reducible*.

An *unsatisfiable* rule can never be true regardless of the concept we want to learn, i.e. irrespective of specific training examples. For instance, given this BK, if we adopt a closed world assumption (Reiter 1977), then, because there is no fact of the form *tail(A,A)*, we can deduce that *tail/2* is irreflexive and remove rules $r_2$ and $r_6$ from the rule space because their bodies are unsatisfiable. The rule space is now:

$r_1$ = $h \leftarrow$ *len(A,B)*
$r_3$ = $h \leftarrow$ *tail(A,B), tail(B,A)*
$r_4$ = $h \leftarrow$ *head(A,B), head(A,C)*
$r_5$ = $h \leftarrow$ *tail(A,B), tail(B,C), tail(A,C)*
$r_7$ = $h \leftarrow$ *head(A,B), odd(B), even(B)*
$r_8$ = $h \leftarrow$ *head(A,B), int(B), odd(B)*
$r_9$ = $h \leftarrow$ *head(A,B), succ(B,C), succ(C,D), lt(B,D)*

Similarly, we can deduce that *tail/2* is asymmetric and antitransitive and remove rules $r_3$ and $r_5$ from the rule space. The rule space is now:

$r_1$ = *h* ← *len(A,B)*
$r_4$ = *h* ← *head(A,B), head(A,C)*
$r_7$ = *h* ← *head(A,B), odd(B), even(B)*
$r_8$ = *h* ← *head(A,B), int(B), odd(B)*
$r_9$ = *h* ← *head(A,B), succ(B,C), succ(C,D), lt(B,D)*

Finally, we can deduce that *odd/1* and *even/1* are mutually exclusive and remove rule $r_7$. The rule space is now:

$r_1$ = *h* ← *len(A,B)*
$r_4$ = *h* ← *head(A,B), head(A,C)*
$r_8$ = *h* ← *head(A,B), int(B), odd(B)*
$r_9$ = *h* ← *head(A,B), succ(B,C), succ(C,D), lt(B,D)*

An *implication reducible* rule contains an implied literal. For instance, because *odd(A)* implies *int(A)*, any rule with both *int(A)* and *odd(A)* in the body has a redundant literal so cannot be in an optimal hypothesis. Therefore, we can remove rule $r_8$ from the rule space. The rule space is now:

$r_1$ = *h* ← *len(A,B)*
$r_4$ = *h* ← *head(A,B), head(A,C)*
$r_9$ = *h* ← *head(A,B), succ(B,C), succ(C,D), lt(B,D)*

Similarly, if *succ(B,C)* and *succ(C,D)* both hold then *lt(B,D)* must hold because $D = B + 2$. Therefore, any rule with these three literals has a redundant literal so cannot be in an optimal hypothesis. Therefore, we can remove rule $r_9$ from the rule space. The rule space is now:

$r_1$ = *h* ← *len(A,B)*
$r_4$ = *h* ← *head(A,B), head(A,C)*

A *recall reducible* rule contains a redundant literal determined by the number of ground instances deducible from the BK. For instance, because *head/2* is functional, any rule with both *head(A,B)* and *head(A,C)* is reducible because $B = C$. Therefore, we can remove rule $r_4$ from the rule space. The rule space is now:

$r_1$ = *h* ← *len(A,B)*

Finally, a *singleton reducible* rule contains a literal that is always true. For instance, if $A$ is always a list then the literal *len(A,B)* in rule $r_1$ is always true because every list has a length and the variable $B$ is unconstrained. In other words, because the variable $B$ only appears once in this rule, the literal *len(A,B)* is redundant because it has no discriminatory power. Therefore, we can remove rule $r_1$ from the hypothesis space. The rule space is now empty.

We show that we can remove hypotheses that contain any of these types of pointless rules from the hypothesis space without removing optimal hypotheses. As this scenario shows, removing hypotheses with pointless rules can substantially shrink the hypothesis space before even looking at the given training examples. In other words, our approach can discover where not to search before searching for a hypothesis.

To find unsatisfiable and implication reducible rules, we use answer set programming (ASP) (Gebser et al. 2012). Specifically, we build small rules and use ASP to check whether these rules are unsatisfiable or implication reducible with respect to the BK. We repeat this process until reaching a user-defined timeout. The output of this stage is a set of unsatisfiable and implication reducible rules.

To find recall reducible rules, we use a bottom-up approach similar to Savnik and Flach (1993) for discovering functional dependencies in a database and Struyf and Blockeel (2003) for building efficient queries. For a background relation and any subsequence of its arguments, we calculate the maximum number of answer substitutions. For instance, in the BK given above, *succ/2* has at most three answer substitutions because there are

only three ground instances. Likewise, for the literal *succ(A,B)*, if $A$ is ground then there is at most one answer substitution for $B$. The output of this stage is a set of recall reducible rules.

To find singleton reducible rules we use ASP to determine whether a relation is partial or total and use that information to deduce singleton reducible rules. The output of this stage is a set of singleton reducible rules.

We use the set of pointless rules to shrink the hypothesis space of a constraint-based ILP system (Cropper and Morel 2021; Hocquette, Niskanen, Järvisalo, et al. 2024). For instance, if we discover that *even* and *odd* are mutually exclusive, we build constraints that remove rules that contain both the body literals *odd(A)* and *even(A)*. Likewise, if we discover that the literals *succ(B,C)*, *succ(C,D)*, and *lt(B,D)* are implication reducible, the constraints remove rules that contain all these literals in them. Our constraints remove non-optimal hypotheses from the hypothesis space so that an ILP system never considers them when searching for a hypothesis.

Our shrinking approach has many benefits. The approach is system-agnostic and could be used by any ILP system. For instance, Aleph's rule pruning mechanism (Srinivasan 2001) could use the constraints. The approach is a stand-alone preprocessing step, independent of a specific learning algorithm or learning task. Therefore, we can use our approach as a preprocessing step on a set of BK shared by many tasks. The main benefit is a drastic reduction in learning times without reducing predictive accuracy. For instance, given only 10 seconds to shrink the hypothesis space, our approach can reduce learning time from over 10 hours to just a few seconds.

*Novelty and Contributions.* The key novelty of this paper is an approach that automatically shrinks the hypothesis space of an ILP system as a preprocessing step by finding pointless rules. The impact is vastly reduced learning times, demonstrated in diverse domains, sometimes reducing learning times by 99%.

This paper substantially extends our earlier work (Cropper and Hocquette 2023b). In that preliminary work, we introduced the idea of using BK to discover constraints on hypotheses, such as that a number cannot be both even and odd. The preliminary work used a human-provided predefined set of properties, such as *antitransitivity* and *irreflexivity*. In this work, we generalise the idea to four types of pointless rules, where implication reducible and singleton reducible are new. Almost all the content in this paper is new.

Overall, our contributions are:

- We introduce the hypothesis space reduction problem, where the goal is to find a subset of a hypothesis space which still contains all optimal hypotheses.
- We define pointless rules, i.e. unsatisfiable, implication reducible, recall reducible, and singleton reducible rules. We show that hypotheses with pointless rules cannot be optimal (Propositions 1, 3, 5, and 8).
- We describe a hypothesis space shrinking approach and implement it in a system called Shrinker. We prove that Shrinker only removes non-optimal hypotheses from the hypothesis space (Proposition 9).
- We use Shrinker to bootstrap a constraint-based ILP system (Cropper and Morel 2021; Hocquette, Niskanen, Järvisalo, et al. 2024).
- We experimentally show on multiple domains that, given only 10 seconds preprocessing time, Shrinker can drastically reduce ILP learning times, sometimes from over 10 hours to 2 seconds.

## 2 Related Work

### 2.1 Program Synthesis

ILP is a form of program synthesis, where the goal is to automatically generate computer programs from examples. This topic, which Gulwani et al. (2017) consider the holy grail of AI, interests a broad community (Ellis et al. 2018; Evans and Grefenstette 2018). Although our approach could be applied to any form of program synthesis, it is well-suited to ILP because ILP's logical representation naturally supports declarative knowledge through logical constraints.

## 2.2 Rule Induction

ILP approaches induce rules from data, similar to rule learning methods (Fürnkranz and Kliegr 2015). It is difficult to compare ILP methods with recent rule mining techniques, such as AMIE+ (Galárraga et al. 2015) and RDFRules (Zeman et al. 2021). Most rule-mining methods are limited to unary and binary relations and require facts as input. They also typically operate under an open-world assumption. By contrast, ILP usually operates under a closed-world assumption, supports relations of any arity, and can learn from definite programs as background knowledge.

## 2.3 ILP

Our approach automatically shrinks the hypothesis space by finding rules that cannot be in an optimal hypothesis. Many systems allow humans to manually specify conditions for when a rule cannot be in a hypothesis (Law et al. 2014; S. Muggleton 1995; Srinivasan 2001). Most systems only reason about the conditions *after* constructing a hypothesis, such as Aleph's rule pruning mechanism (Srinivasan 2001). By contrast, we automatically discover constraints as a preprocessing step and remove rules that violate them from the hypothesis space *before* searching for a hypothesis.

## 2.4 Bottom Clauses

Many ILP systems, such as Aleph, use bottom clauses (S. Muggleton 1995) or variants, such as kernel sets (Ray 2009), to restrict the hypothesis space. The bottom clause of an example $e$ is the most specific clause that entails $e$. Bottom clauses can also be seen as informing an ILP system where to search because an ILP system never needs to consider a rule more specific than the bottom clause. Our approach is similar, as it restricts the hypothesis space. There are, however, many differences. Bottom clauses are example specific. To find a rule to cover an example, a learner constructs the bottom clause for that specific example, which it uses to bias the search. Building a bottom clause can be expensive and the resulting bottom clause can be very large (S. Muggleton 1995). Moreover, as it contains all literals that can be true, a bottom clause will likely include many redundant literals, such as *int(A)* and *odd(A)*. By contrast, our approach finds such reducible rules and removes them from the hypothesis space. In addition, in the worst case, a learner needs to build a bottom clause for every positive training example. Indeed, the Kernel set, as used by xhail (Ray 2009), is built from all positive examples at once. This approach is expensive when there are many examples. By contrast, our bias discovery approach is task independent and only uses the BK, not the training examples. In other words, the running time of our approach is independent of the number of examples. Because of this difference, we can reuse any discovered bias across examples and tasks. For instance, if we discover that the successor relation (*succ*) is asymmetric, we can reuse this bias across multiple tasks. Moreover, bottom clauses introduce several fundamental limitations, such as making it difficult for the corresponding ILP system to learn recursive hypotheses and hypotheses with predicate invention (Cropper and Dumancic 2022). By contrast, our approach does not consider examples and only looks at the given BK to reduce the hypothesis space so it can be used by systems that can learn recursive hypotheses and perform predicate invention. Another difference is that bottom clauses are essential for inverse entailment methods (S. Muggleton 1995). By contrast, our approach is an independent preprocessing step that allows us to amortise its cost (such as by using a timeout). Finally, our shrinking approach could help bottom clause approaches, such as to bootstrap a constraint-based inverse entailment approach (Ahlgren and Yuen 2013) or to provide constraints for Aleph's clause pruning conditions (Srinivasan 2001).

## 2.5 Redundancy in ILP

Most work in ILP focuses on improving search efficiency for a fixed hypothesis space. There is little work on hypothesis space reduction. Srinivasan and Kothari (2005) introduce methods to reduce the dimensionality of

bottom clauses using statistical methods by compressing bottom clauses. The authors state that the resulting lower dimensional space translates directly to a smaller hypothesis space. We differ because we do not use examples in the preprocessing step and do not need to build bottom clauses, alleviating all the issues in the aforementioned bottom clause section. Fonseca et al. (2004) define *self-redundant* clauses, similar to our definition of an implication reducible rule (Definition 10). Their definition does not guarantee that a redundant clause's refinements (specialisations) are redundant. By contrast, we prove that specialisations of an implication reducible rule are reducible (Proposition 2). Moreover, the authors do not propose detecting such rules. Instead, they expect users to provide information about them. By contrast, we introduce SHRINKER, which automatically finds reducible and indiscriminate rules. Raedt and Ramon (2004) check whether a rule has a redundant atom before testing it on examples. If so, it avoids the coverage check. This approach requires *anti-monotonic* constraints, where if a constraint holds for a rule then it holds for all its generalisations, but not necessarily its specialisations. By contrast, we find properties that allow us to prune specialisations of a rule. Moreover, the approach of Raedt and Ramon (2004) does not explicitly identify implications between literals and thus could keep building rules with the same implied literals. By contrast, SHRINKER explicitly finds implications between literals to prune the hypothesis space. Zeng et al. (2014) prune rules with simple forms of syntactic redundancy. For instance, for the rule $h(X) \leftarrow p(X,Y), p(X,Z)$, the authors detect that $p(X,Z)$ is duplicate to the literal $p(X,Y)$ under the renaming $Z \mapsto Y$, where $Z$ and $Y$ are not in other literals. By contrast, we detect semantic redundancy by finding unsatisfiable, implication reducible, recall reducible, and singleton reducible rules. Cropper and Tourret (2020) eliminate redundant metarules (second-order rules) to improve the performance of metarule-based ILP approaches (Dai and S. Muggleton 2021; Kaminski et al. 2019; S. H. Muggleton et al. 2015). Metarules are templates that define the possible syntax of a hypothesis. By contrast, we use an ILP system that does not need metarules.

## 2.6 Rule Selection

Many recent systems formulate the ILP problem as a rule selection problem (Bembenek et al. 2023; Corapi et al. 2011; Kaminski et al. 2019; Law et al. 2014). These approaches precompute every possible rule in the hypothesis space and then search for a subset that generalises the examples. Because they precompute all possible rules, they cannot learn rules with many literals. Moreover, because they precompute all possible rules, they can build pointless rules. For instance, if allowed to use the relations *int/1* and *even/1*, ASPAL (Corapi et al. 2011) and ILASP (Law et al. 2014) will precompute all possible rules with *int(A)* and *even(A)* in the body. By contrast, we implement SHRINKER to work with the constraint-based ILP system POPPER (Cropper and Morel 2021; Hocquette, Niskanen, Järvisalo, et al. 2024). We bootstrap POPPER with the constraints discovered by SHRINKER. For instance, if SHRINKER identifies that the pair of literals *int(A)* and *even(A)* are implication reducible, the constraints prohibit POPPER from building a rule with both those literals.

## 2.7 Constraints

Many systems use constraints to restrict the hypothesis space (Ahlgren and Yuen 2013; Corapi et al. 2011; Cropper and Morel 2021; Hocquette, Niskanen, Morel, et al. 2024; Inoue et al. 2013; Kaminski et al. 2019; Law et al. 2014; Schüller and Benz 2018). For instance, the Apperception engine (Evans, Hernández-Orallo, et al. 2021) has several built-in constraints, such as a *unity condition*, which requires that objects are connected via chains of binary relations. By contrast, we automatically discover constraints before searching for a hypothesis. Users can provide constraints to many systems. For instance, if a user knows that two literals are unsatisfiable, they could provide this information to ILASP in the form of *meta-constraints* or ALEPH via user-defined pruning conditions. However, in these cases, a user gives the constraints as input to the system. By contrast, our approach automatically discovers such constraints without user intervention.

### 2.8 Preprocessing

A key feature of our approach is that we shrink the hypothesis space before giving it an ILP system to search for a hypothesis. Our approach can, therefore, be seen as a preprocessing step, which has been widely studied in AI, notably to reduce the size of a SAT instance (Eén and Biere 2005). There is other work on preprocessing in ILP, notably to infer types from the BK (McCreath and Sharma 1995; Picado et al. 2017). For instance, McCreath and Sharma (1995) automatically deduce mode declarations from the BK, such as types and whether arguments should be ground. We do not infer types but remove unsatisfiable, implication reducible, recall reducible, and singleton reducible rules from the hypothesis space. Moreover, because we use constraints, we can reason about properties that modes cannot capture, such as antitransitivity, mutual exclusivity, and implications. Bridewell and Todorovski (2007) learn structural constraints over the hypothesis space in a multi-task setting. By contrast, we discover biases before solving any task. Other preprocessing approaches in ILP focus on reducing the size of BK (Dumancic et al. 2021; Dumančić et al. 2019) or predicate invention (Cropper 2019; Hocquette and S. H. Muggleton 2020). By contrast, we discover constraints in the BK to shrink the hypothesis space. As a preprocessing step, Struyf and Blockeel (2003) calculate the recall of the given background relations to order the literals in a rule to reduce the time it takes to check a rule against the examples and BK. We use a similar algorithm, but rather than order literals in a rule, we use recall to define recall reducible rules (Definition 11) and build constraints from them to shrink the hypothesis space.

### 2.9 Redundancy in AI

Plotkin (1971) uses subsumption to decide whether a literal is redundant in a first-order clause. Joyner (1976) investigates the same problem, which he calls *clause condensation*, where a condensation of a clause $C$ is a minimum cardinality subset $C'$ of $C$ such that $C' \models C$. Gottlob and Fermüller (1993) improve Joyner's algorithm and show that determining whether a clause is condensed is coNP-complete. Detecting and eliminating redundancy is useful in many areas of computer science and has received much attention from the theorem-proving community (Hoder et al. 2012; Khasidashvili and Korovin 2016; Vukmirovic et al. 2023). In the SAT community, redundancy elimination techniques, such as blocked clause elimination (Kullmann 1999), play an integral role in modern solvers. Similar to the goal of our paper, the goal of these redundancy elimination methods is to soundly identify derivationally irrelevant input.

## 3 Problem Setting

We now define the notation used in this paper, provide a short introduction to ILP, define the *hypothesis space reduction problem*, and then define four types of pointless rules that we can remove from the hypothesis space without removing optimal hypotheses.

### 3.1 Preliminaries

We assume familiarity with logic programming (Lloyd 2012) and ASP (Gebser et al. 2012) but we restate some key definitions. An *atom* is of the form $p(t_1, \ldots, t_a)$ where $p$ is a predicate symbol of arity $a$ and $t_1, \ldots, t_a$ are terms. We refer to a ground term as a constant. A *literal* is an atom or a negated atom. A *rule* $r$ is a definite clause of the form $h \leftarrow p_1, \ldots, p_n$ where $h, p_1, \ldots, p_n$ are literals, $head(r) = h$, and $body(r) = \{p_1, \ldots, p_n\}$. We denote the set of variables in a literal $l$ as $vars(l)$. The variables of a rule $r$, denoted $vars(r)$, is defined as $vars(head(r)) \cup \bigcup_{p \in body(r)} vars(p)$. Given a rule $r$ and a set of positive literals $C$, by $r \cup C$ we denote the rule $r'$ such that $head(r) = head(r')$ and $body(r') = body(r) \cup C$. A substitution $\theta = \{v_1/t_1, ..., v_n/t_n\}$ is the simultaneous replacement of each variable $v_i$ by its corresponding term $t_i$. A rule $r_1$ $\theta$-subsumes a rule $r_2$, denoted $r_1 \preceq_\theta r_2$, if and only if there exists a substitution $\theta$ such that $r_1\theta \subseteq r_2$. A rule $r_2$ is a specialisation of a rule $r_1$ if $r_1 \preceq_\theta r_2$.

We focus on a restricted form of $\theta$-subsumption (Plotkin 1971) which only considers whether the bodies of two rules with the same head literal are contained in one another. We define this restriction through a *sub-rule* relation:

**Definition 1 (Sub-rule).** Let $r_1$ and $r_2$ be rules, $head(r_1) = head(r_2)$, and $body(r_1) \subseteq body(r_2)$. Then $r_1$ is a *sub-rule* of $r_2$, denoted $r_1 \subseteq r_2$. A rule $r_1$ is a strict sub-rule of $r_2$, denoted $r_1 \subset r_2$, if $head(r_1) = head(r_2)$ and $body(r_1) \subset body(r_2)$.

A *definite program* is a set of definite rules. A *Datalog program* (Dantsin et al. 2001) is a definite program with some key restrictions, such as no function symbols. We use the term *hypothesis* synonymously with Datalog program.

We generalise the sub-rule relation to a *sub-hypothesis* relation. Unlike the sub-rule relation, the sub-hypothesis relation is not a restriction of $\theta$-subsumption:

**Definition 2 (Sub-hypothesis).** Let $h_1$ and $h_2$ be hypotheses and for all $r_1 \in h_1$ there exists $r_2 \in h_2$ such that $r_1 \subseteq r_2$. Then $h_1$ is a *sub-hypothesis* of $h_2$, denoted $h_1 \subseteq h_2$.

The sub-hypothesis relation captures a particular type of hypothesis we refer to as *basic*. These are hypotheses for which specific rules do not occur as part of a recursive predicate definition:

**Definition 3 (Basic rule).** Let $h$ be a hypothesis, $r_1$ be a rule in $h$, and for all $r_2$ in $h$, the predicate symbol of $head(r_1)$ does not occur in a literal in $body(r_2)$. Then $r_1$ is *basic* in $h$.

As we show below, under certain conditions we can prune hypotheses with respect to their basic rules.

## 3.2 Inductive Logic Programming

We formulate our approach in the ILP learning from entailment setting (De Raedt 2008). We define an ILP input:

**Definition 4 (ILP input).** An ILP input is a tuple $(E, B, \mathcal{H})$ where $E = (E^+, E^-)$ is a pair of sets of ground atoms denoting positive ($E^+$) and negative ($E^-$) examples, $B$ is background knowledge, and $\mathcal{H}$ is a hypothesis space, i.e a set of possible hypotheses.

We restrict background knowledge to Datalog programs. In the rest of this paper, we assume that the head literal of a rule in a hypothesis does not appear in the head of a rule in the BK.

We define a cost function:

**Definition 5 (Cost function).** Given an ILP input $(E, B, \mathcal{H})$, a cost function $cost_{E,B} : \mathcal{H} \rightarrow S$ maps hypotheses to a totally ordered set $S$.

We use a cost function where $S = \mathbb{N}^2$ with a lexicographic order that first minimises the number of misclassified training examples and then minimises the number of literals in a hypothesis. Given a hypothesis $h$, a true positive is a positive example entailed by $h \cup B$. A true negative is a negative example not entailed by $h \cup B$. A false positive is a negative example entailed by $h \cup B$. A false negative is a positive example not entailed by $h \cup B$. We denote the number of false positives and false negatives of $h$ as $fp_{E,B}(h)$ and $fn_{E,B}(h)$ respectively. We consider a function $size : \mathcal{H} \rightarrow \mathbb{N}$, which evaluates the size of a hypothesis $h \in \mathcal{H}$ as the number of literals in it. In the rest of this paper, we use the cost function:

$$cost_{E,B}(h) = (fp_{E,B}(h) + fn_{E,B}(h), size(h))$$

Given an ILP input and a cost function $cost_{E,B}$, we define an *optimal* hypothesis:

**Definition 6 (Optimal hypothesis).** Given an ILP input $(E, B, \mathcal{H})$ and a cost function $cost_{E,B}$, a hypothesis $h \in \mathcal{H}$ is *optimal* with respect to $cost_{E,B}$ when $\forall h' \in \mathcal{H}$, $cost_{E,B}(h) \leq cost_{E,B}(h')$.

### 3.3 Hypothesis Space Reduction

Our goal is to reduce the hypothesis space without removing optimal hypotheses:

**Definition 7 (Hypothesis space reduction problem).** Given an ILP input $(E, B, \mathcal{H})$, the *hypothesis space reduction problem* is to find $\mathcal{H}' \subset \mathcal{H}$ such that if $h \in \mathcal{H}$ is an optimal hypothesis then $h \in \mathcal{H}'$.

In this paper, we shrink the hypothesis space by using the BK to find rules that cannot be in an optimal hypothesis. We consider four types of rules: *unsatisfiable*, *implication reducible*, *recall reducible*, and *singleton reducible*. We describe these in turn.

### 3.4 Unsatisfiable Rules

An unsatisfiable rule has a body that can never be true given the BK. For instance, consider the rule:

$$h \leftarrow \textit{even(A), odd(A), int(A)}$$

Assuming that the relations *odd* and *even* are mutually exclusive, this rule is unsatisfiable because its body is unsatisfiable.

As a second example, consider the rule:

$$h \leftarrow \textit{succ(A,B), succ(B,C), succ(A,C)}$$

Assuming a standard definition for the *succ/2* relation, this rule is unsatisfiable because *succ/2* is *antitransitive*.

We define an unsatisfiable rule:

**Definition 8 (Unsatisfiable rule).** Let $B$ be BK and $r$ be a rule with the body $b$. Then $r$ is unsatisfiable if there is no grounding substitution $\theta$ such that $B \models b\theta$.

We show that a hypothesis with an unsatisfiable rule cannot be optimal.

**Proposition 1 (Unsatisfiability soundness).** Let $B$ be BK, $h$ be a hypothesis, $r \in h$, $r' \subseteq r$, and $r'$ be unsatisfiable with respect to $B$. Then $h$ is not optimal.

Proof. Assume the opposite, *i.e.* that $h$ is optimal. Let $h' = h \setminus \{r\}$. Since $r' \subseteq r$ then $r'$ $\theta$-subsumes $r$ and thus $r' \models r$. Since $r'$ is unsatisfiable w.r.t. $B$ then $r$ is unsatisfiable w.r.t. $B$. Because $r$ is unsatisfiable, it does not influence the set of derivable facts of $h$, so $h$ and $h'$ entail exactly the same examples, i.e. $fp_{E,B}(h) = fp_{E,B}(h')$ and $fn_{E,B}(h) = fn_{E,B}(h')$. Since $h' = h \setminus \{r\}$ then $size(h') < size(h)$. Therefore, $cost_{E,B}(h') < cost_{E,B}(h)$, so $h$ cannot be optimal, contradicting the assumption. □

### 3.5 Implication Reducible Rules

An *implication reducible* rule has a body literal that is implied by other body literals. For example, consider the rules:

$$r_1 = h \leftarrow \textit{odd(A), int(A)}$$
$$r_2 = h \leftarrow \textit{odd(A)}$$

Assuming that *odd* implies *int*, the rule $r_1$ is implication reducible because its body contains a redundant literal, *int(A)*. Therefore, $r_1$ is logically equivalent to $r_2$ (with respect to BK).

As a second example, consider the rule:

$$h \leftarrow \textit{gt(A,B), gt(B,C), gt(A,C)}$$

Assuming a standard definition for the *gt/2* relation, this rule is implication reducible because *gt/2* is transitive, i.e. *gt(A,B), gt(B,C)* implies *gt(A,C)*.

Because an implication reducible rule contains a redundant literal, an implication reducible rule cannot be in an optimal hypothesis. However, a specialisation of an implication reducible rule can be in an optimal hypothesis. For instance, consider the rule:

$$r_1 = h \leftarrow member(X,L),\ member(Y,L)$$

In this rule, *member(X,L)* implies *member(Y,L)* and vice-versa, so one of the literals is redundant. However, we could still specialise this rule as:

$$r_2 = h \leftarrow member(X,L),\ member(Y,L),\ gt(Y,X)$$

The rule $r_1$ is not logically equivalent to $r_2$ and $r_2$ may appear in an optimal hypothesis.

We identify implication reducible rules where we can prune all its specialisations. The idea is to identify a redundant *captured* literal, which is a literal implied by other literals and where all of its variables appear elsewhere in the rule. For instance, consider the rule:

$$h \leftarrow succ(A,B),\ succ(B,C),\ gt(C,A),\ gt(C,D)$$

In this rule, the literal *gt(C,A)* is captured because its variables all appear elsewhere in the rule. By contrast, the literal *gt(C,D)* is not captured because the variable $D$ does not appear elsewhere in the rule.

We define a captured literal:

**Definition 9 (Captured literal).** Let $r$ be a rule, $l \in body(r)$, and $vars(l) \subseteq vars(body(r) \setminus \{l\})$. Then $l$ is *r-captured*.

If a literal is captured in a rule then it is captured in its specialisations:

**Lemma 1.** Let $r_1$ be a rule, $r_2 \subseteq r_1$, $l \in body(r_2)$, and $l$ be $r_2$-captured. Then $l$ is $r_1$-captured.

Proof. Follows from Definition 1 as the sub-rule relation preserves variable occurrence. □

We define an implication reducible rule:

**Definition 10 (Implication reducible).** Let $r$ be a rule, $B$ be BK, $l \in body(r)$ be $r$-captured, and $B \models r \leftrightarrow r \setminus \{l\}$. Then $r$ is *implication reducible*.

Some specialisations of an implication reducible rule are implication reducible:

**Proposition 2 (Implication reducible specialisations).** Let $B$ be BK, $r_1$ be an implication reducible rule, and $r_1 \subseteq r_2$. Then $r_2$ is implication reducible.

Proof. Since $r_1$ is implication reducible then, by definition 10, there exists $l \in body(r_1)$ s.t. $l$ is $r_1$-captured and $B \models r_1 \leftrightarrow (r_1 \setminus \{l\})$. Since $l$ is $r_1$-captured then, by Lemma 1, it is also $r_2$-captured. Because $l$ is $r_2 - captured$, every variable in $l$ occurs elsewhere in $r_2$ so removing $l$ does not remove any variables from $r_2$. Let $C = body(r_2) \setminus body(r_1)$. Then $B \models (r_1 \cup C) \leftrightarrow ((r_1 \cup C) \setminus \{l\})$. Finally, since $r_2 = (r_1 \cup C)$ then $B \models r_2 \leftrightarrow (r_2 \setminus \{l\})$, so $r_2$ is implication reducible. □

Certain hypotheses that contain a sub-hypothesis with implication reducible rules are not optimal:

**Proposition 3 (Implication reducible soundness).** Let $B$ be BK, $h_1$ be a hypothesis, $h_2 \subseteq h_1$, $r_1$ be a basic rule in $h_1$, $r_2 \in h_2$, $r_2 \subseteq r_1$, and $r_2$ be implication reducible with respect to $B$. Then $h_1$ is not optimal.

Proof. By Proposition 2, $r_1$ is also reducible implying that there exists a rule $r_3 \subseteq r_1$ such that (i) $B \models r_1 \leftrightarrow r_3$ and (ii) $|r_3| < |r_1|$. Let $h_3 = (h_1 \setminus \{r_1\}) \cup \{r_3\}$. Then $cost_{E,B}(h_3) < cost_{E,B}(h_1)$, i.e. $h_1$ is not optimal. □

### 3.6 Recall Reducible Rules

A *recall reducible* rule contains a redundant literal determined by the number of ground instances of a literal deducible from the BK (Cropper and Dumancic 2022). For example, consider the rule:

$$h \leftarrow succ(A,B),\ succ(A,C)$$

This rule contains a redundant literal (*succ(A,B)* or *succ(A,C)*) because the second argument of *succ/2* is functionally dependent on the first argument. In other words, if *succ(A,B)* and *succ(A,C)* are true then *B=C*.

As a second example, consider the rule:

$$h \leftarrow add(A,B,C),\ add(A,B,D)$$

This rule contains a redundant literal (*add(A,B,C)* or *add(A,B,D)*) because the third argument of *add/3* is functionally dependent on the first two arguments. In other words, if *add(A,B,C)* and *add(A,B,D)* are true then *C=D*.

Our implication reducible definition (Definition 10) requires literals to be captured, so is insufficient for functional dependencies. For instance, in the above *succ/2* example, neither literal is captured. We therefore introduce the notion of a *recall reducible rule* to capture this type of redundancy. Specifically, our definition captures the observation that redundancy occurs when multiple literals with the same predicate are instantiated with identical constants. In these cases, there exists another rule that contains fewer literals in the body and logically implies the original rule. Our formal definition is:

**Definition 11 (Recall reducible).** Let $B$ be BK, $r_1$ be a rule, $r_1 \preceq_\theta r_2$, $|r_1| > |r_2|$, and $B \models r_1 \leftrightarrow r_2$. Then $r_1$ is recall reducible.

We show how recall reducible applies to a rule:

**Example 1.** The following example illustrates that $r_1$ (as defined below) is recall reducible. Observe that $r_1 \preceq_\theta r_2$ as $r_1\theta = r_2$, $|r_1| > |r_2|$, and $B \models r_1 \rightarrow r_2$ follows from $r_1 \preceq_\theta r_2$. Furthermore, $B \models r_2 \rightarrow r_1$ follows from *succ/2* being injective.

$$B = \{succ(1,2),\ succ(2,3),\ \ldots\}$$
$$r_1 = h \leftarrow succ(X,Y),\ succ(X,Z)$$
$$r_2 = h \leftarrow succ(U,V)$$
$$\theta = \{X \mapsto U, Y \mapsto V, Z \mapsto V\}$$

**Example 2.** The following example illustrates that $r_1$ (as defined below) is recall reducible. Observe that $r_1 \preceq_\theta r_2$ as $r_1\theta = r_2$, $|r_1| > |r_2|$, and $B \models r_1 \rightarrow r_2$ follows from $r_1 \preceq_\theta r_2$. Furthermore, $B \models r_2 \rightarrow r_1$ follows from *edge/2* being a bijective mapping from $\{a, b, c\}$ to itself.

$$B = \{edge(a,b),\ edge(b,c),\ edge(c,a)\ \}$$
$$r_1 = h \leftarrow edge(A,B),\ edge(B,C),\ edge(C,D),\ edge(D,E)$$
$$r_2 = h \leftarrow edge(A,B),\ edge(B,C),\ edge(C,A)$$
$$\theta = \{D \mapsto A, E \mapsto B\}$$

We show that a specialisation of a recall reducible rule is also recall reducible:

**Proposition 4 (Recall specialisation).** Let $B$ be BK, $r_1$ be a recall reducible rule, and $r_1 \subseteq r_2$. Then $r_2$ is recall reducible.

Proof (Sketch). See Appendix B for the full proof. Essentially, $r_1$ contains more literals with the same predicate symbol than the background knowledge supports with distinct values, i.e. some literals must be redundant. If we add additional literals to the body of $r_1$, regardless of their predicate symbol, the resulting rule, $r_2$ will still contain the redundancies of $r_1$. □

A hypothesis with a recall reducible rule cannot be optimal:

**Proposition 5 (Recall reducible soundness).** Let $B$ be BK, $h_1$ be a hypothesis, $h_2 \subseteq h_1$, $r_1$ be a basic rule in $h_1$, $r_2 \in h_2$, $r_2 \subseteq r_1$, and $r_2$ be recall reducible with respect to $B$. Then $h_1$ is not optimal.

Proof. By Proposition 4, $r_1$ is recall reducible as $r_2 \subseteq r_1$. Thus, there exists a substitution $\theta$, a rule $r_3 = r_1\theta$, and hypothesis $h_3 = (h_1 \setminus \{r_1\}) \cup \{r_3\}$ such that (i) $B \models r_1 \leftrightarrow r_3$ and (ii) $|r_3| < |r_1|$. It follows that $cost_{E,B}(h_3) < cost_{E,B}(h_1)$, i.e. $h_1$ is not optimal. □

Appendix C details the difference between implication and recall reducible rules.

### 3.7 Singleton Reducible Rules

A *singleton reducible* rule contains a literal that is always true. For instance, consider the rule:

$$f(A) \leftarrow length(A,B)$$

If $A$ is of type `list` then *length(A,B)* is always true because every list has a length. In other words, because the variable $A$ of length is total and the variable $B$ only appears once in this rule, the literal *length(A,B)* is redundant because it has no discriminatory power.

We can identify total literals by separating its variables into two sets $S_1$ and $S_2$ and observing that for any instantiation of variables in $S_1$ there exists an instantiation of the variables in $S_2$ such that the literal is true. For instance, consider the rule *divides(A,B,C)* ← *A/B = C*. Separate the variables in the sets $S_1 = \{B\}$ and $S_2 = \{A, C\}$. If we instantiate $B$ with 0 then *(A,0,C)* is false for any instantiation of $A$ and $C$. Thus, we say that *divides(A,B,C)* is partial in $S_1$. If instead we chose $S_1 = \{C\}$ and $S_2 = \{A, B\}$, then for every instantiation of $C$ there is an instantiation of $A$ and $B$ such that *divides(A,B,C)* is true. Thus, we say that *divides(A,B,C)* is total in $S_1$.

In this section, we assume a simple typing framework, which we describe in Appendix A, and that all literals are well typed. We first define a literal instance:

**Definition 12 (Instance).** Let $l$ be a well-typed literal, $S \subseteq vars(l)$, and $\theta$ be a well-typed substitution with respect to $l$ that maps each variable in $S$ to a constant. Then $l\theta$ is an instance of $l$ over $S$. The set of $S$ instances of $l$ is denoted $inst(S, l)$.

We define a *total literal*:

**Definition 13 (Total literal).** Let $B$ be BK, $l$ a literal, and $S \subseteq vars(l)$. Then $l$ is *total* in $S$ if for every $l' \in inst(S, l)$, $B \models l'$; otherwise, $l$ is partial in $S$.

In addition to checking whether a given set $S$ of variable argument positions is total in a given literal $l$, we need to check whether the positions which are not in $S$ only occur in $l$; otherwise, the removal of the literal could change the semantics of the rule containing $l$. For instance, consider the rule *p(A,B)* ← *add(A,B,C), mul(A,B,C)*. Observe that *p(A,B)* is true when $A = B = 0$ or $A = B = 2$. Furthermore, no matter how we choose $S$, *add(A,B,C)* is total in $S$. If we remove *add(A,B,C)* from the above rule, the resulting rule accepts any pair of numbers.

As this example illustrates, we need to take care when removing total literals. We define the precise requirements:

**Definition 14 (Singleton reducible literal).** Let $r$ be a rule, $l$ be a body literal of $r$, $S \subseteq vars(l)$ such that $l$ is total in $S$ and each $V \in vars(l) \setminus S$ occurs precisely once in $r$. Then $l$ is *singleton reducible*. By $tot(l)$ we denote $vars(l) \setminus S$.

If a literal is singleton reducible then it does not affect whether the rule is entailed by the BK:

**Proposition 6.** Let $B$ be BK, $r$ be a rule, and $l$ be a singleton reducible literal in $r$. Then $B \models r \leftrightarrow r \setminus \{l\}$.

Proof. Let $r = (h \leftarrow b, l)$ and $r' = (h \leftarrow b)$. For $B \models r' \rightarrow r$, it follows since $r'$ subsumes $r$. For $B \models r \rightarrow r'$, let $\sigma$ be any grounding substitution for the variables of $r'$ and assume $B \models b\sigma$. Since $l$ is singleton reducible in $r$, there exists $S \subseteq vars(l)$ such that $l$ is total in $S$ and each variable in $vars(l) \setminus S$ occurs precisely once in $r$ and thus only in $l$. Extend $\sigma$ to a grounding substitution $\theta$ for all variables of $r$ such that $B \models l\theta$, where the extra

variables only occur in $l$. Such a substitution is constructable by Definition 13 as $l$ is total in $S$. Thus $B \models (b \wedge l)\theta$, and using $r$ we obtain $B \models h\theta$, hence $B \models h\sigma$. Therefore $B \models r \rightarrow r'$. Hence $B \models r \leftrightarrow r'$. □

A *singleton reducible rule* is a rule with a singleton reducible literal. The specialisations of a singleton reducible rule are also singleton reducible if the reducible literal remains total:

**Proposition 7 (Singleton specialisation).** Let $B$ be BK, $r_1$ be a rule, $l$ be a singleton reducible literal in $r_1$, $r_1 \subseteq r_2$, and variables in $tot(l)$ are singleton in $r_2$. Then $r_2$ is a singleton reducible rule.

Proof. Since $r_1 \subseteq r_2$ then $l \in body(r_2)$. The totality of $l$ only depends on $B$, so $l$ is total in $r_2$. By assumption, the variables in $tot(l)$ are singleton in $r_2$. Therefore, by Definition 14, $l$ is singleton reducible in $r_2$. □

We now show a soundness result similar to those introduced in previous sections:

**Proposition 8 (Singleton reducible soundness).** Let $B$ be BK, $h_1$ be a hypothesis, $h_2 \subseteq h_1$, $r_1$ be a basic rule in $h_1$, $r_2 \in h_2$, $r_2 \subseteq r_1$, $r_2$ be singleton reducible with respect to $B$ and a literal $l \in body(r)$, and variables in $tot(l)$ are singleton in $r_1$. Then $h_1$ is not optimal.

Proof. By Proposition 7, $r_1$ is also singleton reducible. Let $h_3 = (h_1 \setminus \{r_1\}) \cup \{r_1 \setminus \{l\}\}$. By Proposition 6, $B \models r \rightarrow r \setminus \{l\}$. Since $r_1$ is basic in $h_1$, it holds that for all $r' \in h_1 \setminus r_1$, $head(r_1) \notin body(r')$. Hence, $B \models (h_1 \setminus \{r_1\}) \leftrightarrow (h_3 \setminus \{r_1 \setminus \{l\}\})$. Furthermore, the singleton reducibility of $r_1$ implies that $h_1$ and $h_3$ entail exactly the same examples, i.e. $fp_{E,B}(h_1) = fp_{E,B}(h_3)$ and $fn_{E,B}(h_1) = fn_{E,B}(h_3)$. But $cost_{E,B}(h_3) < cost_{E,B}(h_1)$. Therefore, $h_1$ is not optimal. □

## 4 SHRINKER

The previous section outlines four types of pointless rules that we can deduce from BK. We now describe SHRINKER, which automatically finds such pointless rules to remove from the hypothesis space. Algorithm 1 shows the high-level algorithm. This algorithm takes as input BK in the form of a Datalog program. In the rest of this section, we assume, for simplicity, that the BK has been materialised into a set of facts (Hu et al. 2018).

**Algorithm 1** SHRINKER

```
1 def shrinker(bk, max_size, max_vars, batch_size, timeout):
2   pointless_rules = {}
3   pointless_rules += find_unsat_impli(bk, max_size, max_vars, batch_size, timeout)
4   pointless_rules += find_recall_reducible(bk)
5   pointless_rules += find_singleton_reducible(bk)
6   return pointless_rules
```

SHRINKER has three components. The first finds unsatisfiable and implication reducible rules (line 3). The second finds recall reducible rules (line 4). The third finds singleton reducible rules (line 5). We describe these components in turn. We then describe how SHRINKER uses the pointless rules to shrink the hypothesis space of a constraint-based ILP system. The output of the algorithm is independent of the order of the three steps.

### 4.1 Finding Unsatisfiable and Implication Reducible Rules

To find unsatisfiable and implication reducible rules, SHRINKER builds small rule templates and uses the BK to check whether any instances are unsatisfiable or implication reducible. A *template* is a set of second-order literals. For instance, the set *{P(A,B), Q(B,A)}* is a template where $P$ and $Q$ are second-order variables and $A$ and $B$ are

first-order variables. An *instance* of a template is a grounding of the second-order variables, such as *{succ(A,B), succ(B,A)}*. SHRINKER considers batches of templates. As there can be many possible templates, SHRINKER uses a timeout to find as many unsatisfiable or implication reducible rules as possible in the time limit.

Algorithm 2 shows the algorithm for finding unsatisfiable and implication reducible rules. The algorithm takes the inputs: background knowledge (*BK*), a maximum number of literals in a rule (*max_size*), a maximum number of unique variables in a rule (*max_vars*), the number of templates to check in each iteration (*batch_size*), and a time limit (*timeout*). The algorithm returns a set of pointless rules.

**Algorithm 2** Find unsatisfiable and reducible rules.

```
1 def find_unsat_impli(bk, max_size, max_vars, batch_size, timeout):
2   templates = build_templates(bk, max_size, max_vars)
3   start_time = get_time()
4   pointless_rules = {}
5   while (get_time() - start_time) < timeout:
6     check_templates = select_templates(templates, batch_size)
7     templates -= check_templates
8     pointless_rules += find_pointless_rules(bk, check_templates)
9   return pointless_rules
```

Algorithm 2 works as follows. The function *build_templates* (line 2) builds all possible templates with at most *max_size* literals and at most *max_vars* variables and returns them in ascending order of size (number of literals). Lines 5-8 show the loop to test templates on the BK to find unsatisfiable and implication reducible rules. The function *select_templates* (line 6) selects *batch_size* untested templates to test. The function *find_pointless_rules* (line 8) is key. It takes as input BK and templates and returns the discovered pointless rules.

We implement the *find_pointless_rules* function in a bottom-up way using ASP. Specifically, we search for a counter-example for each rule to show that it is not unsatisfiable or not implication reducible. For instance, to show that any rule with *p(A,B), p(B,C), p(A,C)* in the body is unsatisfiable, we need to check there is no counter-example where the literals *p(A,C), p(B,C), p(A,C)* all simultaneously hold in the BK. Likewise, to show that any rule with the literals *p(A), q(A)* in the body is implication reducible, we need to check there is no counter-example where either *p(A)* holds and *q(A)* does not, or vice-versa.

We use ASP to perform these checks. We first build an ASP encoding $\mathcal{P}$ which includes all the BK. In the following, we use `typewriter typeface` to denote ASP code. For each relation $p$ with arity $a$ in the BK, we add these rules to $\mathcal{P}$:

`pred(p,a).`
`holds(p,(`$X_1, X_2, \ldots, X_a$`)):- p(`$X_1, X_2, \ldots, X_a$`).`

We use the given templates to build ASP rules to add to $\mathcal{P}$ to find pointless rules. We describe each type of rule in turn.

*4.1.1 Unsatisfiable Rules.* Given the template *{P(A,B), Q(B,A)}*, we add these ASP rules to $\mathcal{P}$:

```
sat_ab_ba(P,Q):- holds(P,(A,B)), holds(Q,(B,A)).
unsat_ab_ba(P,Q):- pred(P,2), pred(Q,2), not sat_ab_ba(P,Q).
```

The atom *sat_ab_ba(P,Q)* is true for the predicate symbols $P$ and $Q$ if the literals *P(A,B)* and *Q(B,A)* are both true for any values $A$ and $B$. This atom can be seen as a counter-example to say that this template with these

predicate symbols is satisfiable. The atom *unsat_ab_ba(P,Q)* is true for the predicate symbols $P$ and $Q$ if there is no satisfiable counter-example. These ASP rules allow us to deduce asymmetry for binary relations.

Given the template *{P(A,B), Q(B,C), R(A,C)}*, we add these ASP rules to $\mathcal{P}$:

```
sat_ab_bc_ac(P,Q,R):- holds(P,(A,B)), holds(Q,(B,C)), holds(R,(A,C)).
unsat_ab_bc_ac(P,Q,R):- pred(P,2), pred(Q,2), pred(R,2), not sat_ab_bc_ac(P,Q,R).
```

These rules allow us deduce antitransitivity for binary relations.

*4.1.2 Implication Reducible Rules.* Given the template *{P(A), Q(A)}*, we add these ASP rules to $\mathcal{P}$:

```
aux_a_a(P,Q):- holds(P,(A,)), not holds(Q,(A,)).
implies_a_a(P,Q):- pred(P,1), pred(Q,1), P != Q, not aux_a_a(P,Q).
```

The atom *aux_a_a(P,Q)* is true for the predicate symbols $P$ and $Q$ if there is any literal *P(A)* that is true where the literal *Q(A)* is not true. This atom can be seen as a counter-example to say that *P(A)* does not imply *Q(A)*. The atom *implies_a_a(P,Q)* is true for the predicate symbols $P$ and $Q$ if there is no implication counter-example.

Given the template *{P(A,B), Q(B,C), R(A,C)}*, we add these ASP rules to $\mathcal{P}$:

```
aux_ab_bc_ac(P,Q,R):- holds(P,(A,B)), holds(Q,(B,C)), not holds(R,(A,C)).
implies_ab_bc_ac(P,Q,R):- pred(P,2), pred(Q,2), pred(R,2), not aux_ab_bc_ac(P,Q,R).
```

*4.1.3 Deducing Unsatisfiable and Implication Reducible Rules.* We use the ASP solver CLINGO (Gebser et al. 2019) to find a model of $\mathcal{P}$ and thus find unsatisfiable and implication reducible rules. Specifically, we use the multi-shot solving feature of CLINGO (Gebser et al. 2019) to test batches of templates incrementally without regrounding BK. SHRINKER represents a pointless rule as an atom. For instance, if the template *{succ(A,B), succ(B,A)}* is unsatisfiable then SHRINKER represents it as the atom *unsat_ab_ba(succ,succ)*. Algorithm 2 returns a set of such atoms, denoting pointless rules.

## 4.2 Finding Recall Reducible Rules

To find recall reducible rules, we use an approach inspired by Struyf and Blockeel (2003) for ordering literals in a rule to improve query efficiency. See Section 2.8 for more detail on this work. At a high level, for any background relation, we determine the maximum number of answer substitutions (ground instantiations) for any subset of its arguments. We use standard recall notation (S. Muggleton 1995) to denote input/ground *(+)* and output/non-ground *(-)* arguments. To illustrate this notation, consider the facts:

```
p(1,2). p(2,1). p(3,1).
q(p1,a,b). q(p2,b,c). q(p3,a,b). q(p4,b,c).
```

The recall for *p(-,-)* is 3 because there are only 3 *p/2* facts. The recall for *p(+,-)* is 1 because, for any ground first argument, there is at most 1 ground second argument. The recall for *q(+,-,-)* is also 1 because, for any first argument, there is at most one pair formed of arguments 2 and 3. The recall for *q(-,+,+)* is 2 because for any pair of arguments formed of arguments 2 and 3 there are at most 2 non-ground arguments in position 1.

Algorithm 3 shows our algorithm for finding recall reducible rules. The algorithm takes as input BK and returns a set of atoms denoting the recall of a relation with respect to a subset of its arguments. The algorithm works as follows. It loops over every atom in the BK (line 4) and every subset of the arguments of an atom (line 9), which corresponds to all recall combinations. For instance, for the atom *succ(A,B)*, using the recall notation, the subsets are *succ(-,-)*, *succ(+,-)*, and *succ(-,+)*. We ignore the case when all the arguments are input/ground *(succ(+,+))* because the recall is one. For each subset, we create a key formed of the input/ground arguments and a value formed of the output/non-ground arguments. We then add the keys and values to a hashmap that is specific

for the predicate symbol and its argument subset. Finally, for each relation and subset of its arguments, we use this hashmap to calculate the maximum answer substitutions (recall). The algorithm returns a set of atoms of the form *recall_pred_input_output_count*, where *pred* is a predicate symbol, *input* is a sequence of input variables, *output* is a sequence of output variables, and *count* is the corresponding recall value. For instance, for *p(+,-)* with recall 1, the atom is *recall_p_a_b_1*. For *p(-,+)* with recall 1, the atom is *recall_p_b_a_1*. For *q(-,+,+)* with recall 2, the atom is *recall_q_bc_a_2*.

Algorithm 3 is exponential in the maximum arity of a predicate symbol in the background knowledge because it iterates over the powerset of argument positions (line 9). In practice, the maximum arity is small, usually <4.

**Algorithm 3** Find recall reducible rules

```
1 def find_recall_redundant(bk):
2   counts = {}
3   preds = {}
4   for atom in bk:
5     pred = get_pred(atom)
6     args = get_args(atom)
7     arity = get_arity(atom)
8     preds += (pred, arity)
9     for arg_subset in powerset({1,...,arity}):
10        key = ''
11        value = ''
12        for i in range(arity):
13          if i in arg_subset:
14            key += args[i]
15          else:
16            value += args[i]
17        counts[pred][arg_subset][key].add(value)
18
19  recalls = {}
20  for pred, arity in preds:
21    for arg_subset in powerset({1,...,arity}):
22      keys = counts[pred][arg_subset]
23      recall = max(len(counts[pred][arg_subset][key]) for key in keys)
24      recalls[(pred, arg_subset)] = recall
25  return recalls
```

### 4.3 Finding Singleton Reducible Rules

We find singleton reducible rules using ASP. As a reminder, a singleton reducible rule contains a literal that is always true. For instance, consider the rule:

$$f(A) \leftarrow length(A,B)$$

Assuming sensible semantics for *length/2*, the literal *length(A,B)* is always true because every list has a length. In other words, because the variable $B$ only appears once in this rule and $A$ is true for all lists, the literal *length(A,B)* is redundant as it has no discriminatory power.

We assume that the background relations are simply typed (see Appendix A for an elaboration on types). We use ASP to determine whether a relation is partial or total and use that information to deduce singleton reducible rules. We describe our ASP encoding $\mathcal{P}$. We add the BK to $\mathcal{P}$. For each relation $p$ with arity $a$ in the BK, we add this rule to $\mathcal{P}$:

`holds(p,(`$X_1,X_2,\ldots,X_a$`)):- p(`$X_1,X_2,\ldots,X_a$`).`

For each relation $p$ with arity $a$ and each index $i \in \{1,\ldots,a\}$ with type $t$ we add this fact to $\mathcal{P}$:

```
type(p,i,t).
```

For each relation $p$ with arity $a$ and for every index $i = 1 \ldots a$, we determine whether a constant symbol $C_i$ holds at $i$ for $p$ by adding this rule to $\mathcal{P}$:

`cholds(p,i,`$C_i$`):- holds(p,(`$C_1$`,`$C_i$`,...,`$C_a$`)).`

We determine the constants that hold for a type $T$ (the domain of $T$) by adding this rule to $\mathcal{P}$:

```
domain(T,C):- cholds(P,I,C), type(P,I,T).
```

For a predicate symbol $p$ with arity $a > 1$, we determine whether two constant symbols $C_i$ and $C_j$ hold at indices $i$ and $j$ respectively for $i = 1 \ldots j-1, j \ldots a$ by adding this rule to $\mathcal{P}$:

`holds2(p,i,j,`$C_i$`,`$C_j$`):- cholds(p,i,`$C_i$`), cholds(P,j,`$C_j$`).`

We add similar rules for all values $1 \leq i_1 < i_2 < \cdots < i_k < a$, such as this rule:

`holds3(p,i,j,k,`$C_i$`,`$C_j$`,`$C_k$`):- cholds(p,i,`$C_i$`), cholds(p,j,`$C_j$`), cholds(p,k,`$C_k$`).`

We deduce whether an argument position $I$ is partial with the rule:

```
partial(P,I):- type(P,I,T), domain(T,X), not cholds(P,I,X).
```

We add similar rules for multiple arguments. For instance, we deduce whether two arguments are partial with the rule:

```
partial2(P,I,J):- I<J, type(P,I,T_I), type(P,J,T_J), domain(T_I,C_I), domain(T_J,C_J),
                  not holds2(P,I,J,C_I,C_J).
```

Finally, we deduce whether arguments are total by whether they are not partial. We use an ASP solver to find a model of $\mathcal{P}$ and thus find total literals. Algorithm 2 then returns a set of atoms denoting the total arguments of background literals. For instance, for the relation *length/2*, where the first argument is total, SHRINKER returns the atom *total_length_b*. For the relation *add/3*, where any pair of variables is total, SHRINKER returns three atoms *total_add_ab*, *total_add_ac*, *total_add_bc*.

### 4.4 SHRINKER Correctness and Complexity

We show that SHRINKER is correct in that it solves the hypothesis space reduction problem (Definition 7):

**Proposition 9 (SHRINKER correctness).** Let $I = (E, B, \mathcal{H})$ be an ILP problem, $h \in \mathcal{H}$ be an optimal hypothesis for $I$, and $\mathcal{H}'$ be the hypothesis space resulting from applying SHRINKER to $\mathcal{H}$ given $B$. Then $h \in \mathcal{H}'$.

PROOF. Propositions 1, 3, 5, & 8 imply that the pruning implemented by SHRINKER does not prune optimal hypotheses. Thus, if $h \in \mathcal{H}$ then $h \in \mathcal{H}'$. □

The time complexity of SHRINKER (Algorithm 1) is dominated by the call to Algorithm 2 (finding unsatisfiable and implication reducible rules). For Algorithm 2, let $p$ be number of distinct predicate symbols in $BK$, $a$ be the maximum arity of any predicate symbol in $BK$, $v = max\_vars$, and $m = max\_size$. Then, ignoring the time limit ($timeout$), the worst-case complexity of Algorithm 2 is bounded by $O((pv^a)^m)$. In other words, the complexity grows exponentially in both the maximum arity ($a$) and the maximum template size ($m$).

### 4.5 POPPER

The output of SHRINKER (Algorithm 1) is a set of pointless rules, specifically a set of atoms representing pointless rules. Any ILP system could use these rules to prune the hypothesis space, such as Aleph's rule pruning mechanism (Srinivasan 2001). We use these rules to shrink the hypothesis space of the constraint-based ILP system POPPER (Cropper and Morel 2021), which can learn optimal and recursive hypotheses. There are many POPPER variants, including ones that learn from noisy data (Hocquette, Niskanen, Järvisalo, et al. 2024), learn higher-order programs (Purgal et al. 2022), and learn from probabilistic data (Hillerström and Burghouts 2024). Although SHRINKER will shrink the hypothesis space of all the variants, we describe the simplest version because it is sufficient to explain how we combine it with SHRINKER.

POPPER takes as input BK, training examples, and a maximum hypothesis size and learns hypotheses as definite programs. POPPER searches for an optimal hypothesis which entails all the positive examples, no negative ones, and has minimal size. POPPER uses a generate, test, combine, and constrain loop to find an optimal hypothesis. POPPER starts with an initial ASP encoding $\mathcal{P}$. The encoding $\mathcal{P}$ can be viewed as a *generator* because each model (answer set) of $\mathcal{P}$ represents a hypothesis. The encoding $\mathcal{P}$ uses head (*hlit*/3) and body (*blit/3*) literals to represent hypotheses. The first argument of each literal is the rule id, the second is the predicate symbol, and the third is the literal variables, where *0* represents *A*, *1* represents *B*, etc. For instance, consider the rule:

$$second(A,B) \leftarrow tail(A,C), head(C,B)$$

POPPER represents this rule as the set:

$$\{hlit(0,second,(0,1)), blit(0,tail,(0,2)), blit(0,head,(2,1))\}$$

The encoding $\mathcal{P}$ contains choice rules for *head/3* and *body/3* literals:

```
{hlit(Rule,Pred,Vars)}:- rule(Rule), vars(Vars,Arity), head_pred(Pred,Arity).
{blit(Rule,Pred,Vars)}:- rule(Rule), vars(Vars,Arity), body_pred(Pred,Arity).
```

In these rules, *rule(Rule)* denotes rule indices, *vars(Vars,Arity)* denotes possible variable tuples, and the literals *head_pred(Pred,Arity)* and *body_pred(Pred,Arity)* denote predicate symbols and arities that may appear in the head or body of a rule respectively.

In the *generate stage*, POPPER uses an ASP solver to find a model of $\mathcal{P}$ for a fixed hypothesis size (enforced via a cardinality constraint on the number of head and body literals). If no model is found, POPPER increases the hypothesis size and loops again. If a model exists, POPPER converts it to a hypothesis $h$.

In the *test stage*, POPPER uses Prolog to test $h$ on the training examples and BK. If $h$ entails at least one positive example and no negative examples then POPPER saves $h$ as a *promising hypothesis*.

In the *combine stage*, POPPER searches for a combination (a union) of promising hypotheses that entails all the positive examples and has minimal size. POPPER formulates the search as a combinatorial optimisation problem, specifically as an ASP optimisation problem (Cropper and Hocquette 2023a). If a combination exists, POPPER saves it as the best hypothesis so far and updates the maximum hypothesis size.

In the *constrain stage*, POPPER uses $h$ to build constraints. POPPER adds these constraints to $\mathcal{P}$ to prune models and thus prune the hypothesis space. For instance, if $h$ does not entail any positive example, POPPER adds a constraint to prune its specialisations as they are guaranteed not to entail any positive example. For instance, the following constraint prunes all specialisations (supersets) of the rule $second(A,B) \leftarrow tail(A,C), head(C,B)$:

```
:- hlit(R,second,(0,1)), blit(R,tail,(0,2)), blit(R,head,(2,1)).
```

Popper repeats this loop until it exhausts the models of $\mathcal{P}$ or exceeds a user-defined timeout. Popper then returns the best hypothesis found.

### 4.6 shrinker + Popper

We use shrinker to shrink the hypothesis space of Popper before Popper searches for a hypothesis, i.e. before Popper starts its loop. To do so, we allow Popper to reason about the unsatisfiable, implication reducible, recall reducible, and singleton reducible rules found by shrinker. We provide examples.

*4.6.1 Unsatisfiable Rules.* If shrinker finds an unsatisfiable rule of the form *P(A,B), Q(B,A)* then we add all the *unsat_ab_ba(P,Q)* atoms and the following constraint to Popper:

```
:- unsat_ab_ba(P,Q), blit(Rule,P,(A,B)), blit(Rule,Q,(B,A)).
```

For instance, if `unsat_ab_ba(succ,succ)` holds, this constraint prunes all models (and thus hypotheses) that contain the literals *blit(Rule,succ,(A,B))* and *blit(Rule,succ,(B,A))*, including all variable substitutions for *A*, *B*, and *Rule*. For instance, for a learning task with an arbitrary head literal *h*, the constraint prunes the rule:

$$h \leftarrow \textit{succ(A,B), succ(B,C), succ(C,B)}$$

If shrinker finds an unsatisfiable rule of the form *P(A,B), Q(B,C), R(A,C)* (antitransitivity) then we add all the *unsat_ab_bc_ac(P,Q,R)* atoms found by shrinker and the following constraint to Popper:

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

*4.6.2 Implication Reducible Rules.* If shrinker finds an implication reducible rule of the form *P(A), Q(A)* then we add all the *implies_a_a(P,Q)* atoms and the following constraint to Popper:

```
:- implies_a_a(P,Q), blit(Rule,P,(A,)), blit(Rule,Q,(A,)).
```

If shrinker finds an implication reducible rule of the form *P(A,B), Q(B,C), R(A,C)* then we add the following constraint and all *implies_ab_bc_ac(P,Q,R)* atoms found by shrinker.

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

*4.6.3 Recall Reducible Rules.* We use ASP aggregate constraints to succinctly express recall constraints. For instance, if shrinker discovers that *succ/2* is functional (where for any ground first argument there is at most 1 ground second argument), it represents this information as the atom *recall_succ_a_b_1*. Given this atom, we add this constraint to Popper:

```
:- blit(Rule,succ,(V0,_)), #count{V1:blit(Rule,succ,(V0,V1))} > 1.
```

This constraint prunes all models (and thus hypotheses) containing the literals *blit(Rule, succ, (A,B))* and *blit(Rule, succ, (A,C))* where *B!=C*, i.e. this constraint prunes all rules with the body literals *succ(A,B)* and *succ(A,C)*, where *B!=C*. This constraint applies to all variable substitutions for *A* and *B* and all rules *Rule*. For instance, for a learning task with an arbitrary head literal *h*, the constraint prunes the rule:

$$h \leftarrow \textit{succ(B,C), succ(B,D)}$$

If shrinker discovers that *succ/2* is injective (for any ground second argument there is exactly 1 ground first argument), it represents this information as the atom *recall_succ_b_a_1*. Given this atom, we add this constraint to Popper:

```
:- blit(Rule,succ,(_,V1)), #count{V0: blit(Rule,succ,(V0,V1))} > 1.
```

For instance, the constraint prunes the rule:

$$h \leftarrow succ(A,B),\ succ(C,B)$$

If SHRINKER returns the recall atom *recall_q_bc_a_2*, we add this constraint to POPPER:

```
:- blit(Rule,q,(_,V1,V2)), #count{V0: blit(Rule,q,(V0,V1,V2))} > 2.
```

*4.6.4 Singleton Reducible Rules.* To prune singleton reducible rules, we add this ASP rule to the generate encoding of POPPER:

```
singleton(Rule,V):- rule(Rule), var(V),
    #count{P,Vars : blit(Rule,P,Vars), vars(Vars,_), member(V,Vars)} == 1.
```

In this rule, *var(V)* denotes a variable (*V*) allowed in a rule and *member(V,Vars)* is true when the variable *V* is in the tuple of variables *Vars*. This ASP rule allows us to identify variables that only appear once in a rule. We use this rule to build ASP constraints to prune singleton reducible rules. For instance, for the relation *length/2*, where the first argument is total, SHRINKER returns the atom *total_length_b*. We therefore add this constraint to POPPER:

```
:- blit(Rule,length,(V0,V1)), singleton(Rule,V1).
```

For the relation *add/3*, where any pair of variables is total, SHRINKER returns three atoms *total_add_ab*, *total_add_ac*, *total_add_bc*. We therefore add three constraints to POPPER:

```
:- blit(Rule,add,(V0,V1,V2)), singleton(Rule,V1), singleton(Rule,V2).
:- blit(Rule,add,(V0,V1,V2)), singleton(Rule,V0), singleton(Rule,V2).
:- blit(Rule,add,(V0,V1,V2)), singleton(Rule,V0), singleton(Rule,V1).
```

## 5 Experiments

The purpose of SHRINKER is to shrink a hypothesis space without pruning optimal hypotheses. Proposition 9 shows that SHRINKER does not prune optimal hypotheses. However, it is unclear in practice what impact SHRINKER has on learning performance. Therefore, our experiments[1] aim to answer the question:

**Q1** Can SHRINKER reduce learning times?

To answer **Q1** we compare the learning times of POPPER with and without SHRINKER.

SHRINKER removes unsatisfiable, implication reducible, recall reducible, and singleton reducible rules from the hypothesis space. To understand the impact of each type of rule, our experiments aim to answer the question:

**Q2** Can removing each type of pointless rule reduce learning times?

To answer **Q2** we compare the learning times of POPPER when using SHRINKER but only removing a single type of pointless rule from the hypothesis space.

SHRINKER takes a timeout as a parameter. To understand the impact of this parameter, our experiments aim to answer the question:

**Q3** Can more SHRINKER time reduce learning times?

To answer **Q3**, we compare the impact of SHRINKER when using 10 and 100 second timeouts.

[1] All code and experimental data are archived at https://doi.org/10.5281/zenodo.19068412

## 5.1 Experimental Setup

*5.1.1 Generalisation Error.* Each learning task contains training and testing examples. We use the training examples to train the ILP system to learn a hypothesis. We test a hypothesis on the testing examples to see how they generalise to unseen data. Given a hypothesis $h$, background knowledge $B$, and a set of test examples $E$, we define the following. A true positive is a positive example in $E$ entailed by $h \cup B$. A true negative is a negative example in $E$ not entailed by $h \cup B$. We denote the number of true positives and true negatives as $tp(h)$ and $tn(h)$ respectively. We measure generalisation error in terms of *balanced predictive accuracy*. This measure handles imbalanced data by evaluating the average performance across both positive and negative classes:

$$balancedaccuracy(h) = \frac{1}{2}\left(\frac{tp(h)}{tp(h) + fn(h)} + \frac{tn(h)}{tn(h) + fp(h)}\right)$$

Balanced accuracy is equivalent to standard accuracy when the hypothesis performs equally well on both classes or when the data is balanced.

*5.1.2 Settings.* We use SHRINKER with the settings *max_size=3*, *max_vars=6*, *batch_size=1000*, and *timeout=10*, except when we use a timeout of 100 seconds to answer **Q3**. We allow POPPER to learn rules with at most 6 variables and at most 10 body literals. We use an AWS m6a.16xlarge instance to run experiments where each learning task uses a single core.

*5.1.3 Method.* POPPER is an anytime approach. We use the best hypothesis found by POPPER given a maximum learning time of 60 minutes. Learning time is the time POPPER needs to learn and prove optimality of a hypothesis and terminate. We repeat all the experiments 10 times. We measure mean balanced accuracy and mean learning time. We round times over one second to the nearest second. We plot and report 95% confidence intervals. We use a Wilcoxon Signed-Rank Test to determine the statistical significance of any differences in the results. Any subsequent reference to a significance test refers to this test.

*5.1.4 Datasets.* We use several datasets:

**1D-ARC.** This dataset (Xu et al. 2024) contains visual reasoning tasks inspired by the abstract reasoning corpus (Chollet 2019).

**Alzheimer.** These real-world tasks (King et al. 1995) involve learning rules describing four properties desirable for drug design against Alzheimer's disease.

**IGGP.** In inductive general game playing (IGGP) (Cropper, Evans, et al. 2020), the task is to induce rules from game traces from the general game-playing competition (Genesereth and Björnsson 2013).

**IMDB.** We use a real-world dataset which contains relations between movies, actors, and directors (Mihalkova et al. 2007).

**List functions.** The list functions dataset (Rule et al. 2024) is designed to evaluate human and machine concept learning ability. The goal of each task is to identify a function that maps input lists to output lists, where list elements are natural numbers. We use a relational encoding (Hocquette and Cropper 2024).

**Trains.** The goal is to find a hypothesis that distinguishes east and west trains (Larson and Michalski 1977).

**Zendo.** Zendo is a multiplayer game where players must discover a secret rule by building structures (Cropper and Hocquette 2023b).

## 5.2 Experimental Results

*5.2.1 Q1. Can SHRINKER Reduce Learning Times?* Figure 1 shows the improvements in learning times with SHRINKER. A significance test confirms ($p < 0.05$) that SHRINKER decreases learning time on 178/419 (42.5%) tasks and increases learning time on 18/419 (4.3%) tasks. There is no significant difference on the other tasks. The mean decrease in learning time is 20.1 ± 3.3 minutes, corresponding to a 81.1% decrease and a 5.3x speedup.

The mean increase in learning time is 1.2 ± 0.4 seconds, corresponding to a 60.5% increase and a 1.6x slowdown. These are minimum improvements because Popper without shrinker often times out after 60 minutes. With a longer timeout, we would likely see greater improvements. The full results showing the times for every learning task are in Appendix D.

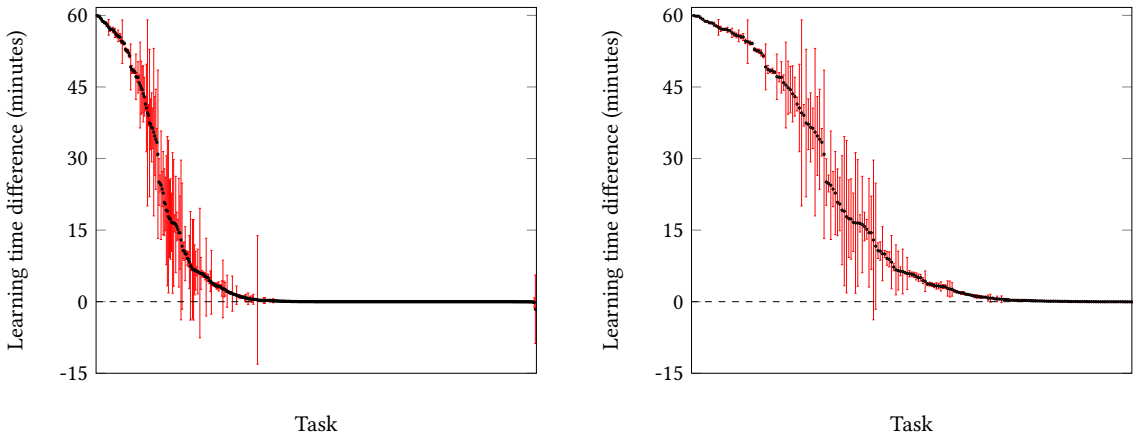


Fig. 1. Learning time improvements when using shrinker. The left figure shows all tasks and the right figure shows tasks where the two approaches significantly ($p < 0.05$) differ. The tasks are ordered by the improvement.

shrinker can drastically reduce learning times. For instance, for the *list-function-043* task, where the task is to learn how to prepend a sequence of numbers to a list, shrinker reduces the learning time from 60 ± 0 minutes to 6 ± 0 minutes, a 90.2% decrease and 10.2x speedup. For the *iggp-duikoshi-next_control* task, shrinker reduces the learning time from 60 ± 0 minutes to 2 ± 0 seconds, a 99.9% reduction and 1800x speedup.

To further explore the potential of shrinker to reduce learning times, we ran a separate experiment where we increased the maximum learning time to 24 hours for a subset of tasks. With this greater timeout, for the *iggp-duikoshi-next_control* task, shrinker reduces the learning time from 6.5 ± 1 hours to 1 ± 0 seconds. For the *iggp-horseshoe-terminal* task, shrinker reduces the learning time from 10 ± 0 hours to 33 ± 3 seconds.

To illustrate why shrinker works, consider the *iggp-scissors_paper_stone-next_score* task. The goal of this task is to learn the rules of the game *rock, paper, scissors* from observations of gameplay. For this task, shrinker discovers that the background relation *succ/2* is irreflexive, antitransitive, antitriangular, and asymmetric, which are expressed as unsatisfiable rules. shrinker also discovers implication reducible rules, such as that the literals *succ(A,B), int_0(A), int_1(B)* are implication reducible. shrinker also discovers that the *succ/2* relation is injective and functional, both expressed as recall reducible rules. shrinker discovers these given only 10 seconds of preprocessing time, yet the resulting constraints reduce the learning time of Popper from 353 ± 46 seconds to 51 ± 1 seconds, an 85.5% reduction and 6.9x speedup.

shrinker increases the learning time on 17/419 tasks. The reason is the overhead of reasoning about the constraints discovered by shrinker, i.e. the overhead of Clingo processing the constraints. For these tasks, the mean increase in learning time is only 1.2 ± 0.4 seconds.

A significance test confirms ($p < 0.05$) that shrinker increases predictive accuracy on 15/419 (3.6%) tasks and decreases accuracy on 24/419 (5.7%) tasks. There is no significant difference on the other tasks. The main reason for any accuracy increase is that without shrinker Popper sometimes does not find a good hypothesis in the time limit. By contrast, as shrinker prunes the hypothesis space, there are fewer hypotheses for Popper

to consider, so it sometimes finds a better hypothesis quicker. Furthermore, as SHRINKER is optimally sound (Proposition 9), it is guaranteed to lead to a hypothesis space that is a subset of the original one yet still contains an optimal hypothesis.

The main reason for the decrease in accuracy is a POPPER implementation issue. For the *iggp-tiger_vs_dogs-goal* task, POPPER without SHRINKER learns this rule:

```
goal(V0,V1,V2):- score_0(V2), agent_d(V1), pos_3(V5), mark_b(V3), pos_3(V4),
                 true_cell(V0,V4,V5,V3).
```

This rule has 58% ± 0 accuracy. POPPER with SHRINKER cannot learn this rule. The reason is that SHRINKER discovers that *pos_3(A)* has a maximum recall of 1, i.e. it is defined for a single value (3). Therefore, SHRINKER says that this rule is recall reducible and prunes it from the hypothesis space. A logically equivalent non-recall reducible rule is:

```
goal(V0,V1,V2):- score_0(V2), agent_d(V1), mark_b(V3), pos_3(V4), true_cell(V0,V4,V4,V3).
```

In this rule, the predicate symbol *pos_3* only appears once but the variable $V4$ appears twice in the literal *true_cell(V0,V4,V4,V3)*. POPPER cannot learn this rule because it prohibits a variable from appearing twice in a literal (due to legacy reasons). Therefore, POPPER without SHRINKER uses multiple variables to learn a logically equivalent rule.

Overall, the results in this section show that the answer to **Q1** is yes, SHRINKER can drastically reduce learning times.

*5.2.2 Q2. Can Removing Each Type of Pointless Rule Reduce Learning Times?* Figure 2 shows the learning times when only removing unsatisfiable rules. Doing so decreases learning time on 38/419 (9.1%) tasks and increases learning time on 21/419 (5.0%) tasks. There is no significant difference on the other tasks. The mean decrease in learning time is 2.0 ± 1.0 minutes, corresponding to a 27.4% decrease and a 1.4x speedup. The mean increase in learning time is 2.8 ± 3.2 seconds, corresponding to a 46.0% increase and a 1.5x slowdown.

Figure 3 shows the learning times when only removing implication reducible rules. Doing so decreases learning time on 154/419 (36.8%) tasks and increases learning time on 10/419 (2.4%) tasks. There is no significant difference on the other tasks. The mean decrease in learning time is 19.2 ± 3.3 minutes, corresponding to a 73.1% decrease and a 3.7x speedup. The mean increase in learning time is 9.4 ± 18.5 seconds, corresponding to a 110.1% increase and a 2.1x slowdown.

Figure 4 shows the learning times when only removing recall reducible rules. Doing so decreases learning time on 70/419 (16.7%) tasks and increases learning time on 9/419 (2.1%) tasks. There is no significant difference on the other tasks. The mean decrease in learning time is 3.5 ± 1.8 minutes, corresponding to a 38.8% decrease and a 1.6x speedup. The mean increase in learning time is 34.1 ± 68.7 seconds, corresponding to a 104.6% increase and a 2.0x slowdown. The reason for the increase in learning time is because, due to the aforementioned *variable appearing twice in a literal* issue, POPPER with SHRINKER sometimes needs to learn a longer rule, so it takes more time.

Figure 5 shows the learning times when only removing singleton rules. Doing so decreases learning time on 66/419 (15.8%) tasks and increases learning time on 11/419 (2.6%) tasks. There is no significant difference on the other tasks. The mean decrease in learning time is 2.7 ± 1.1 minutes, corresponding to a 31.7% decrease and a 1.5x speedup. The mean increase in learning time is 1.3 ± 1.6 minutes, corresponding to a 25.9% increase and a 1.3x slowdown.

Again, it is important to note these are minimum improvements, as POPPER without SHRINKER often times out after 60 minutes.

Overall, the results in this section show that the answer to **Q2** is yes, removing each type of pointless rule can reduce learning times but removing implication reducible rules has the most impact.

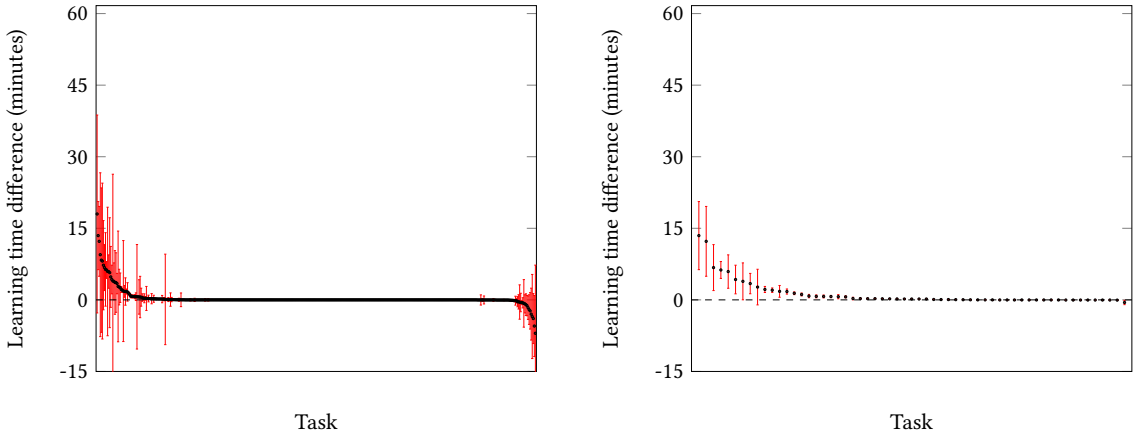


Fig. 2. Learning time improvements when using SHRINKER with only unsatisfiable rules. The left figure shows all tasks and the right figure shows tasks where the two approaches significantly ($p < 0.05$) differ. The tasks are ordered by the improvement.

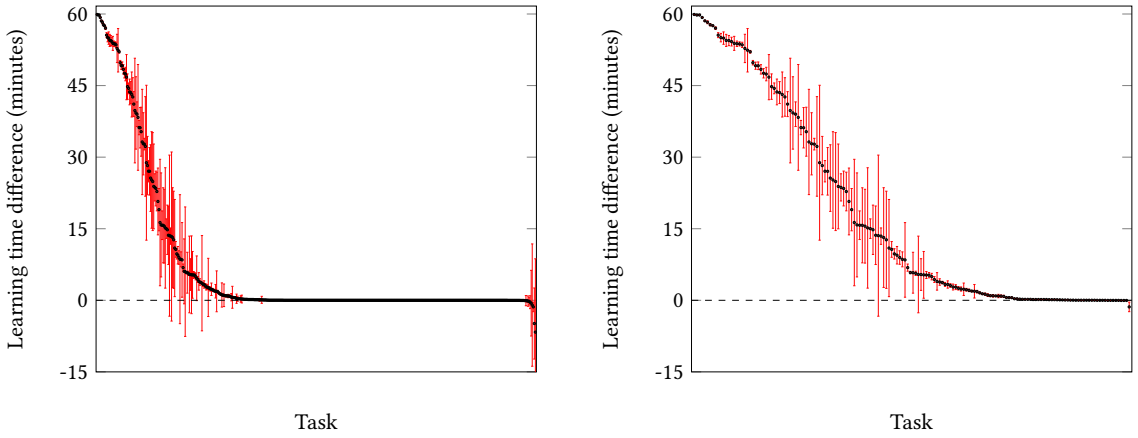


Fig. 3. Learning time improvements when using SHRINKER with only implication reducible rules. The left figure shows all tasks and the right figure shows tasks where the two approaches significantly ($p < 0.05$) differ. The tasks are ordered by the improvement.

*5.2.3 Q3. Can More SHRINKER Time Reduce Learning Times?* Figure 6 shows the learning times when using SHRINKER with 10 and 100 second timeouts. Significance tests confirm ($p < 0.05$) that a 100 second timeout decreases learning time on 26/419 (6.2%) tasks and increases learning time on 27/419 (6.4%) tasks. There is no significant difference on the other tasks. The mean decrease in learning time is 1.0 ± 0.9 minutes, corresponding to a 16.8% decrease and a 1.2x speedup. The mean increase in learning time is 4.0 ± 3.7 seconds, corresponding to a 36.9% increase and a 1.4x slowdown. Overall, the results in this section show that the answer to **Q3** is yes, more SHRINKER time can improve performance but not commensurably so, and that 10 seconds is usually sufficient.

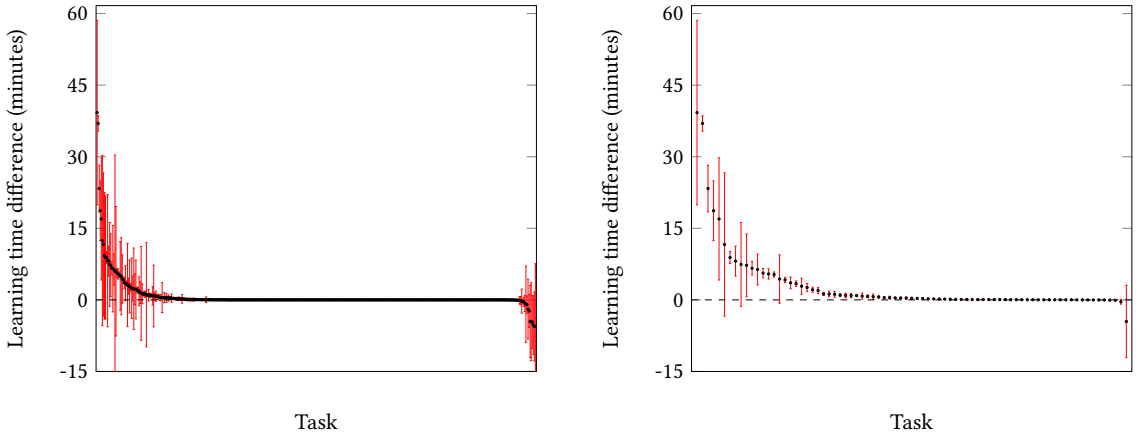


Fig. 4. Learning time improvements when using SHRINKER with only recall reducible rules. The left figure shows all tasks and the right figure shows tasks where the two approaches significantly ($p < 0.05$) differ. The tasks are ordered by the improvement.

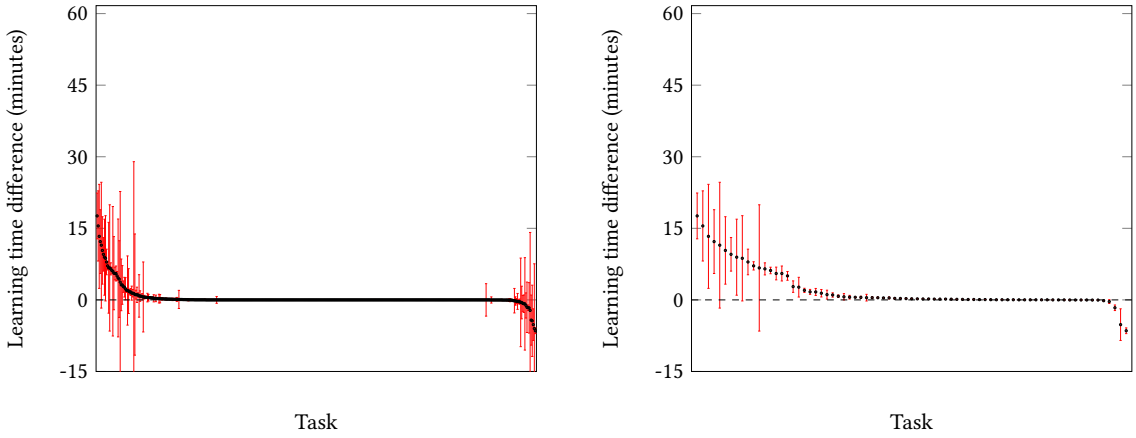


Fig. 5. Learning time improvements when using SHRINKER with only singleton reducible rules. The left figure shows all tasks and the right figure shows tasks where the two approaches significantly ($p < 0.05$) differ. The tasks are ordered by the improvement.

## 6 Conclusions and Limitations

We introduced SHRINKER, an approach to automatically shrink the hypothesis space of an ILP system by preprocessing the background knowledge to find pointless rules. SHRINKER finds four types of pointless rules (unsatisfiable, implication reducible, recall reducible, and singleton reducible). We show that any hypothesis with a pointless rule is non-optimal so can be soundly removed from the hypothesis space (Propositions 1, 3, 5, and 8). Our experiments on multiple domains show that SHRINKER consistently reduces the learning times of an ILP system. For instance, given only 10 seconds of preprocessing time, SHRINKER can reduce learning times from over 10 hours to 2 seconds.

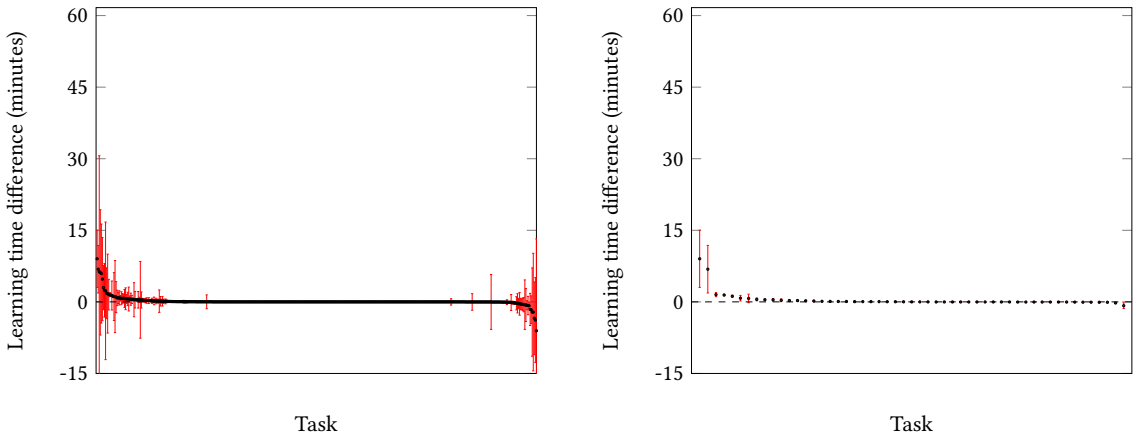


Fig. 6. Learning time improvements when using SHRINKER with 100 vs 10 second timeout. The left figure shows all tasks and the right figure shows tasks where the two approaches significantly ($p < 0.05$) differ. The tasks are ordered by the improvement.

## Limitations

*Finite BK.* Our shrinking idea is sufficiently general to handle definite programs as BK. However, because our bottom-up implementation uses ASP, we require a finite grounding of the BK. This restriction means that our implementation cannot handle BK with an infinite grounding, such as when reasoning about continuous values. Future work should address this limitation, such as by using top-down methods to discover pointless rules.

*Closed-world assumption.* We adopt a closed-world assumption (CWA) to discover pointless rules from given BK. For instance, we assume that *odd(2)* does not hold if not given as BK. As almost all ILP systems adopt a CWA, this limitation only applies if our approach is used with a system that does not make the CWA, such as recent rule mining algorithms (Zeman et al. 2021).

*Monotonic ILP.* Our theoretical results only hold for monotonic ILP, where a hypothesis is a definite program. How to extend all of our results to non-monotonic ILP (Cerna and Cropper 2024), such as learning programs with negation-as-failure (Clark 1977), is unclear. The problem is that some pointless rules are no longer clearly pointless. For instance, suppose a rule $r$ with the head literal *aux* is unsatisfiable (Def 8). Then an optimal hypothesis could still contain the body literal *not aux*, where *not* denotes negation-as-failure. Extending the results to non-monotonic ILP is future work.

*Noisy BK.* We assume that the BK is noiseless, i.e. if a fact is true in the BK then it is meant to be true. Handling noisy BK is an open challenge (Cropper and Dumancic 2022) that is beyond the scope of this paper.

*Efficiency.* SHRINKER brute-force builds templates to find unsatisfiable and implication reducible rules. However, we think that certain rules have a more significant impact on shrinking the hypothesis space. Future work should, therefore, explore ways of ordering the templates to quickly find the most impactful rules.

## Acknowledgments

Andrew Cropper and Filipe Gouveia were supported by the EPSRC fellowship (*EP/V040340/1*). David M. Cerna was supported by the Czech Science Foundation Grant 22-06414L and Cost Action CA20111 EuroProofNet.

## References


J. Ahlgren and S. Y. Yuen. 2013. "Efficient program synthesis using constraint satisfaction in inductive logic programming." *J. Machine Learning Res.*, 14, 1, 3649–3682.

A. Bembenek, M. Greenberg, and S. Chong. Jan. 2023. "From SMT to ASP: Solver-Based Approaches to Solving Datalog Synthesis-as-Rule-Selection Problems." *Proc. ACM Program. Lang.*, 7, POPL, Article 7, (Jan. 2023), 33 pages. doi:10.1145/3571200.

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

W. Bridewell and L. Todorovski. 2007. "Learning declarative bias." In: *International Conference on Inductive Logic Programming*. Springer, 63–77.

D. M. Cerna and A. Cropper. 2024. "Generalisation through Negation and Predicate Invention." In: *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*. Ed. by M. J. Wooldridge, J. G. Dy, and S. Natarajan. AAAI Press, 10467–10475. doi:10.1609/AAAI.V38I9.28915.

F. Chollet. 2019. "On the Measure of Intelligence." *CoRR*.

K. L. Clark. 1977. "Negation as Failure." In: *Logic and Data Bases, Symposium on Logic and Data Bases*. New York, 293–322.

D. Corapi, A. Russo, and E. Lupu. 2011. "Inductive Logic Programming in Answer Set Programming." In: *ILP 2011*.

A. Cropper. 2019. "Playgol: Learning Programs Through Play." In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, 6074–6080.

A. Cropper and S. Dumancic. 2022. "Inductive Logic Programming At 30: A New Introduction." *J. Artif. Intell. Res.*, 74, 765–850. doi:10.1613/jair.1.13507.

A. Cropper, R. Evans, and M. Law. 2020. "Inductive general game playing." *Mach. Learn.*, 109, 7, 1393–1434. doi:10.1007/s10994-019-05843-w.

A. Cropper and C. Hocquette. 2023a. "Learning Logic Programs by Combining Programs." In: *ECAI 2023 - 26th European Conference on Artificial Intelligence*. Vol. 372. IOS Press, 501–508. doi:10.3233/FAIA230309.

A. Cropper and C. Hocquette. 2023b. "Learning Logic Programs by Discovering Where Not to Search." In: *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023*. AAAI Press, 6289–6296. doi:10.1609/AAAI.V37I5.25774.

A. Cropper and R. Morel. 2021. "Learning programs by learning from failures." *Mach. Learn.*, 110, 4, 801–856. doi:10.1007/s10994-020-05934-z.

A. Cropper and S. Tourret. 2020. "Logical reduction of metarules." *Mach. Learn.*, 109, 7, 1323–1369. https://doi.org/10.1007/s10994-019-05834-x.

W. Dai and S. Muggleton. 2021. "Abductive Knowledge Induction from Raw Data." In: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, 1845–1851. doi:10.24963/ijcai.2021/254.

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

L. De Raedt. 2008. *Logical and relational learning*. ISBN: 978-3-540-20040-6. doi:10.1007/978-3-540-68856-3.

S. Dumancic, T. Guns, and A. Cropper. 2021. "Knowledge Refactoring for Inductive Program Synthesis." In: *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021*, 7271–7278. https://ojs.aaai.org/index.php/AAAI/article/view/16893.

S. Dumančić, T. Guns, W. Meert, and H. Blockeel. 2019. "Learning Relational Representations with Auto-encoding Logic Programs." In: *IJCAI 2019*, 6081–6087.

N. Eén and A. Biere. 2005. "Effective Preprocessing in SAT Through Variable and Clause Elimination." In: *SAT 2005*. Vol. 3569, 61–75. doi:10.1007/11499107_5.

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

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

R. Evans, J. Hernández-Orallo, J. Welbl, P. Kohli, and M. J. Sergot. 2021. "Making sense of sensory input." *Artif. Intell.*, 293, 103438. doi:10.1016/j.artint.2020.103438.

N. A. Fonseca, V. S. Costa, F. M. A. Silva, and R. Camacho. 2004. "On Avoiding Redundancy in Inductive Logic Programming." In: *Inductive Logic Programming, 14th International Conference, ILP 2004, Porto, Portugal, September 6-8, 2004, Proceedings* (Lecture Notes in Computer Science). Vol. 3194. Springer, 132–146. doi:10.1007/978-3-540-30109-7_13.

J. Fürnkranz and T. Kliegr. 2015. "A Brief Overview of Rule Learning." In: *RuleML 2015* (Lecture Notes in Computer Science). Vol. 9202. Springer, 54–69. doi:10.1007/978-3-319-21542-6_4.

L. Galárraga, C. Teflioudi, K. Hose, and F. M. Suchanek. 2015. "Fast rule mining in ontological knowledge bases with AMIE+." *VLDB J.*, 24, 6, 707–730.

M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. 2012. *Answer Set Solving in Practice*.

M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. 2019. "Multi-shot ASP solving with clingo." *Theory Pract. Log. Program.*, 19, 1, 27–82. doi:10.1017/S1471068418000054.

M. R. Genesereth and Y. Björnsson. 2013. "The International General Game Playing Competition." *AI Magazine*, 34, 2, 107–111. http://www.aaai.org/ojs/index.php/aimagazine/article/view/2475.

G. Gottlob and C. G. Fermüller. 1993. "Removing Redundancy from a Clause." *Artif. Intell.*, 61, 2, 263–289. doi:10.1016/0004-3702(93)90069-N.
S. Gulwani, O. Polozov, R. Singh, et al.. 2017. "Program synthesis." *Foundations and Trends® in Programming Languages*, 4, 1-2, 1–119.
F. Hillerström and G. J. Burghouts. 2024. "Towards Probabilistic Inductive Logic Programming with Neurosymbolic Inference and Relaxation." *Theory Pract. Log. Program.*, 24, 4, 628–643. doi:10.1017/S1471068424000371.
C. Hocquette and A. Cropper. 2024. "Relational decomposition for program synthesis." *CoRR*, abs/2408.12212. arXiv: 2408.12212. doi:10.48550/ARXIV.2408.12212.
C. Hocquette and S. H. Muggleton. 2020. "Complete Bottom-Up Predicate Invention in Meta-Interpretive Learning." In: *IJCAI 2020*, 2312–2318. doi:10.24963/ijcai.2020/320.
C. Hocquette, A. Niskanen, M. Järvisalo, and A. Cropper. 2024. "Learning MDL Logic Programs from Noisy Data." In: *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI*. AAAI Press, 10553–10561. doi:10.1609/AAAI.V38I9.28925.
C. Hocquette, A. Niskanen, R. Morel, M. Järvisalo, and A. Cropper. 2024. "Learning Big Logical Rules by Joining Small Rules." In: *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, Jeju, South Korea, August 3-9, 2024*. ijcai.org, 3430–3438. https://www.ijcai.org/proceedings/2024/380.
K. Hoder, Z. Khasidashvili, K. Korovin, and A. Voronkov. 2012. "Preprocessing techniques for first-order clausification." In: *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*. Ed. by G. Cabodi and S. Singh. IEEE, 44–51. https://ieeexplore.ieee.org/document/6462554/.
P. Hu, B. Motik, and I. Horrocks. 2018. "Optimised Maintenance of Datalog Materialisations." In: *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. Ed. by S. A. McIlraith and K. Q. Weinberger. AAAI Press, 1871–1879. doi:10.1609/AAAI.V32I1.11554.
K. Inoue, A. Doncescu, and H. Nabeshima. 2013. "Completing causal networks by meta-level abduction." *Mach. Learn.*, 91, 2, 239–277.
W. Joyner. 1976. "Resolution Strategies as Decision Procedures." *J. ACM*, 23, 3, 398–417. doi:10.1145/321958.321960.
T. Kaminski, T. Eiter, and K. Inoue. 2019. "Meta-Interpretive Learning Using HEX-Programs." In: *IJCAI 2019*, 6186–6190. doi:10.24963/ijcai.2019/860.
Z. Khasidashvili and K. Korovin. 2016. "Predicate Elimination for Preprocessing in First-Order Theorem Proving." In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings* (Lecture Notes in Computer Science). Ed. by N. Creignou and D. L. Berre. Vol. 9710. Springer, 361–372. doi:10.1007/978-3-319-40970-2_22.
R. D. King, M. J. E. Sternberg, and A. Srinivasan. 1995. "Relating Chemical Activity to Structure: An Examination of ILP Successes." *New Gener. Comput.*, 13, 3&4, 411–433. doi:10.1007/BF03037232.
O. Kullmann. 1999. "On a Generalization of Extended Resolution." *Discret. Appl. Math.*, 96-97, 149–176. doi:10.1016/S0166-218X(99)00037-2.
J. Larson and R. S. Michalski. 1977. "Inductive inference of VL decision rules." *SIGART Newsletter*, 63, 38–44.
M. Law, A. Russo, and K. Broda. 2014. "Inductive Learning of Answer Set Programs." In: *JELIA 2014*.
J. W. Lloyd. 2012. *Foundations of logic programming*. Springer Science & Business Media.
E. McCreath and A. Sharma. 1995. "Extraction of Meta-Knowledge to Restrict the Hypothesis Space for ILP Systems." In: *Eighth Australian Joint Conference on Artificial Intelligence*, 75–82.
L. Mihalkova, T. N. Huynh, and R. J. Mooney. 2007. "Mapping and Revising Markov Logic Networks for Transfer Learning." In: *AAAI 2007*, 608–614.
S. Muggleton. 1991. "Inductive Logic Programming." *New Generation Computing*, 8, 4, 295–318.
S. Muggleton. 1995. "Inverse Entailment and Progol." *New Generation Comput.*, 13, 3&4, 245–286.
S. H. Muggleton, D. Lin, and A. Tamaddoni-Nezhad. 2015. "Meta-interpretive learning of higher-order dyadic Datalog: predicate invention revisited." *Mach. Learn.*, 100, 1, 49–73.
J. Picado, A. Termehchy, A. Fern, and S. Pathak. 2017. "Towards Automatically Setting Language Bias in Relational Learning." In: *Proceedings of the 1st Workshop on Data Management for End-to-End Machine Learning, DEEM@SIGMOD 2017, Chicago, IL, USA, May 14, 2017*, 3:1–3:4. doi:10.1145/3076246.3076249.
G. Plotkin. Aug. 1971. "Automatic Methods of Inductive Inference." Ph.D. Dissertation. Edinburgh University, (Aug. 1971).
S. J. Purgal, D. M. Cerna, and C. Kaliszyk. 2022. "Learning Higher-Order Logic Programs From Failures." In: *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*. Ed. by L. D. Raedt. ijcai.org, 2726–2733. doi:10.24963/IJCAI.2022/378.
J. R. Quinlan. 1990. "Learning Logical Definitions from Relations." *Mach. Learn.*, 5, 239–266.
L. D. Raedt and J. Ramon. 2004. "Condensed Representations for Inductive Logic Programming." In: *Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004), Whistler, Canada, June 2-5, 2004*. Ed. by D. Dubois, C. A. Welty, and M. Williams. AAAI Press, 438–446. http://www.aaai.org/Library/KR/2004/kr04-046.php.
O. Ray. 2009. "Nonmonotonic abductive inductive learning." *J. Applied Logic*, 7, 3, 329–340.
R. Reiter. 1977. "On Closed World Data Bases." In: *Logic and Data Bases, Symposium on Logic and Data Bases*, 55–76.

J. S. Rule, S. T. Piantadosi, A. Cropper, K. Ellis, M. Nye, and J. B. Tenenbaum. 2024. "Symbolic metaprogram search improves learning efficiency and explains rule learning in humans." *Nature Communications*, 15, 1, 6847. ISBN: 2041-1723. doi:10.1038/s41467-024-50966-x.

I. Savnik and P. A. Flach. 1993. "Bottom-up induction of functional dependencies from relations." In: *Proceedings of the AAAI-93 Workshop on Knowledge Discovery in Databases*, 174–185.

P. Schüller and M. Benz. 2018. "Best-effort inductive logic programming via fine-grained cost-based hypothesis generation - The inspire system at the inductive logic programming competition." *Mach. Learn.*, 107, 7, 1141–1169. doi:10.1007/s10994-018-5708-2.

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

A. Srinivasan and R. Kothari. 2005. "A Study of Applying Dimensionality Reduction to Restrict the Size of a Hypothesis Space." In: *Inductive Logic Programming, 15th International Conference, ILP 2005, Bonn, Germany, August 10-13, 2005, Proceedings*. Vol. 3625, 348–365.

J. Struyf and H. Blockeel. 2003. "Query Optimization in Inductive Logic Programming by Reordering Literals." In: *Inductive Logic Programming: 13th International Conference, ILP 2003, Szeged, Hungary, September 29-October 1, 2003, Proceedings* (Lecture Notes in Computer Science). Ed. by T. Horváth. Vol. 2835. Springer, 329–346. doi:10.1007/978-3-540-39917-9_22.

P. Vukmirovic, J. Blanchette, and M. J. H. Heule. 2023. "SAT-Inspired Eliminations for Superposition." *ACM Trans. Comput. Log.*, 24, 1, 7:1–7:25. doi:10.1145/3565366.

Y. Xu, W. Li, P. Vaezipoor, S. Sanner, and E. B. Khalil. 2024. "LLMs and the Abstraction and Reasoning Corpus: Successes, Failures, and the Importance of Object-based Representations." *Trans. Mach. Learn. Res.*, 2024. https://openreview.net/forum?id=E8m8oySvPJ.

V. Zeman, T. Kliegr, and V. Svátek. 2021. "RDFRules: Making RDF rule mining easier and even more efficient." *Semantic Web*, 12, 4, 569–602. doi:10.3233/SW-200413.

Q. Zeng, J. M. Patel, and D. Page. 2014. "QuickFOIL: Scalable Inductive Logic Programming." *VLDB*, 197–208.

## A Typing

We assume a simple typing mechanism with the properties described below.

**Definition 15 (Types).** Let $B$ be background knowledge. Then $types(B)$ is a set of unary predicate symbols defined in $B$ with the following properties: For all $ty_1, ty_2 \in types(B)$ and constants $c$

- $c$ is of type $ty_1 \in types(B)$ if and only if $B \models ty_1(c)$, and
- $ty_1(c)$ and $ty_2(c)$ if and only if $ty_1 = ty_2$.

For the rest of the paper we only consider background knowledge $B$ with non-empty $types(B)$.

**Definition 16 (Position types).** Let $B$ be background knowledge, $p$ be a predicate symbol with arity $a$, and $1 \leq i \leq a$. Then $arg\_type_B(p, i) \in types(B)$ denotes the type of the $i^{th}$ argument of $p$ in $B$.

For the rest of the paper we only consider predicate symbols whose arguments are typed with respect to the given background knowledge.

**Definition 17 (Well-typed literal).** Let $B$ be background knowledge and $l = p(t_1, \ldots, t_a)$ a literal. Then $l$ is *well-typed over* $B$ if

- for all $1 \leq i \leq a$ where $arg\_type(p, i) = ty$ and $t_i$ is a constant, $B \models ty(t_i)$ and
- for all $1 \leq i, j \leq a$ where $t_i$ and $t_j$ are variables, $t_i = t_j$ iff $arg\_type(p, i) = arg\_type(p, j)$.

Using well-typed literals we can construct well typed-substitutions.

**Definition 18 (Well-typed substitution).** Let $B$ be background knowledge, $l$ a literal, and $\sigma$ a substitution with domain $S \subseteq var(l)$. Then $\sigma$ is a *well-typed substitution with respect to* $l$ if $l\sigma$ is well-typed with respect to $B$.

In addition to assuming that all literals we consider are well-typed, we also assume the following consistency statement, i.e., non-well-typed literals do not entail from the background knowledge.

**Statement 1 (Consistency).** Let $B$ be background knowledge, $p$ a predicate symbol with arity $a$, $c_1, \ldots, c_a$ constants, and for some $1 \leq i \leq a$, $arg\_type(p, i) = ty$ and $B \not\models ty(c_i)$. Then $B \not\models p(c_1, \ldots, c_a)$.

Essentially, consistency states that what entails from the background knowledge is type consistent.

**Definition 19 (Well-typed rule).** Let $B$ be background knowledge and $r$ a rule. Then $r$ is *well-typed over* $B$ if for all literals $p(t_1, \ldots, t_a)$ and $q(s_1, \ldots, s_b)$ in $r$, the following holds: for all $1 \leq i \leq a$ and $1 \leq j \leq b$ where $t_i$ and $s_j$ are variables, $t_i = s_j$ iff $arg\_type(p, i) = arg\_type(q, j)$. A hypothesis is well-typed over $B$ if all of its rules are.

We assume all rules and hypotheses used in this paper are well-typed. Observe that the typing mechanisms outlined above can be enforced by adding the appropriate type literals to the body of a rule, thus no additional type checking infrastructure is needed. Note, we do not consider such type literals are part of the body and thus they do not count towards the size of a rule.

## B Recall Reducible

Our proof of Proposition 4 requires an alternative formulation of *recall reducible* (Definition 22) and then proving that the alternative formulation is equivalent to recall reducible ((Definition 11). This alternative formulation allows us to convert the substitution implicit in $\theta$-subsumption into equality literals. These equality literals are than used to test logical equivalence. The previously mentioned construction requires comparing argument tuples of a set of literals:

**Definition 20 (Arguments).** Let $l = p(X_1, \ldots, X_n)$ be a literal. Then $args(l) = (X_1, \ldots, X_n)$, i.e. a tuple of variables containing the arguments of $l$ in the order they occur.

Our alternative definition of recall reducible requires identifying mutually distinct argument tuples.

**Definition 21 (All different).** Let $S$ be a set of tuples of variables such that for all $t_1, t_2 \in S$, $|t_1| = |t_2|$. Then $alldiff(S) = \{neq(t_1, t_2) \mid (t_1, t_2 \in S) \wedge (t_1 \neq t_2)\}$ where for $t_1 = (X_1, \ldots, X_n)$ and $t_2 = (Y_1, \ldots, Y_n)$, we define the binary predicate $neq$ as follows: $neq(t_1, t_2) \equiv (X_1 \neq Y_1 \vee \cdots \vee X_n \neq Y_n)$.

We can interpret this type of redundancy as adaptation of the *pigeonhole principle* to rule learning. The following definition captures this adaptation:

**Definition 22 (Pigeonholed rule).** Let $B$ be background knowledge, $r$ be a rule, $b \subseteq body(r)$ where every literal in $b$ has the same predicate symbol, $s = \{args(l) \mid l \in b\}$, and $B \not\models r \leftrightarrow (r \cup \{alldiff(s)\})$. Then $r$ is pigeonholed.

Essentially, Definition 22 states the following: if a rule $r$ has $n + 1$ literals with the predicate symbol $p$, but can only instantiate the argument tuples of the literals in $k < n$ ways, then some of the literal occurrences are not necessary. The following examples illustrate this construction:

**Example 3.** Consider the following example and Definition 22

$$
\begin{aligned}
B =& \{edge(a, b), edge(b, c), edge(c, a)\} \\
r =& h \leftarrow edge(X, Y), edge(Y, Z), edge(Z, W), edge(W, E) \\
b =& \{edge(X, Y), edge(Y, Z), edge(Z, W), edge(W, E)\} \\
s =& \{(X, Y), (Y, Z), (Z, W), (W, E)\} \\
alldiff(s) =& \{neq((X, Y), (Y, Z)), neq((X, Y), (Z, W)), \\
& \quad neq((X, Y), (W, E)), neq((Y, Z), (Z, W)), \\
& \quad neq((Y, Z), (W, E)), neq((Z, W), (W, E))\}
\end{aligned}
$$

Observe that $B \not\models r \leftrightarrow (r \cup \{alldiff(s)\})$ because $B$ defines a 3-cycle.

**Example 4.** Consider the following example and Definition 22

$$\begin{aligned} B &= \{p(a,b,c), p(b,a,c), p(c,b,a)\} \\ r &= h \leftarrow p(A,B,C), p(A,Y,Z). \\ b &= \{p(A,B,C), p(A,Y,Z)\} \\ s &= \{(A,B,C),(A,Y,Z)\} \\ \mathit{alldiff}(s) &= \{\mathit{neq}((A,B,C),(A,Y,Z))\} \end{aligned}$$

Observe that $B \not\models r \leftrightarrow (r \cup \{\mathit{alldiff}(s)\})$ because $B$ only contains one instance of $p/3$ per choice of first argument.

As we will show in Theorem 1, *recall reducible* (Definition 11) captures our adaptation of the *pigeonhole principle*. Consider two rules $r_1$ and $r_2$. By Definition, $r_1$ is *recall reducible* if $r_1 \preceq_\theta r_2$, $|r_2| < |r_1|$ and the two rules are logically equivalent. Observe that this realises the inequalities introduced in Definition 22 through the implicit substitution of $\theta$-subsumption.

As mentioned above, recall reducible and Pigeonholed are equivalent concepts. While pigeonholed formalises our adaptation of the *pigeonhole principle* through the introduction of inequalities, recall reducible uses the implicit substitution of $\theta$-subsumption. The proof of Theorem 1 provides the constructions allowing one to switch between the two concepts.

**Theorem 1.** Let $B$ be background knowledge and $r$ a rule. Then $r$ is recall reducible iff $r$ is pigeonholed.

Proof.
$\underline{\Longrightarrow}$: By Definition 11, there exists $\theta$ such that $r\theta = r_2$. Furthermore, given that $|r| > |r_2|$, there must exists literals $l_1, l_2 \in \mathit{body}(r_1)$ such that $l_1 \neq l_2$ and $l_1\theta = l_2\theta$. Let $b$ be the subset of $\mathit{body}(r)$ containing all literals with the same predicate symbol as $l_1$ and $l_2$. Observe that

$$B \models (r \leftrightarrow r_2) \rightarrow \neg(r \leftrightarrow r \cup \mathit{alldiff}(\{\mathit{args}(l) \mid l \in b\})),$$

holds because the construction of $r_2$ unifies at least two literals in $b$. Thus, we derive that $B \models \neg(r \leftrightarrow r \cup \mathit{alldiff}(\{\mathit{args}(l) \mid l \in b\}))$, i.e. $B \not\models r \leftrightarrow r \cup \mathit{alldiff}(\{\mathit{args}(l) \mid l \in b\})$. Thus, $r$ is pigeonholed.

$\underline{\Longleftarrow}$: By Definition 22, there exists $b \subseteq \mathit{body}(r)$ such that the literals in $b$ have the same predicate symbol and $B \not\models r \leftrightarrow (r \cup \{\mathit{alldiff}(s)\})$, where $s = \{\mathit{args}(l) \mid l \in b\}$. This implies that there exists $c \subset b$ and a mapping $f : c \rightarrow (c \setminus b)$ such that

$$B \models r \leftrightarrow (r \cup \{\mathit{args}(p) = \mathit{args}(f(p)) \mid p \in c\})$$

Now consider some $p \in c$. From $\mathit{args}(p) = (X_1, \cdots, X_n)$ and $\mathit{args}(f(p)) = (Y_1, \cdots, Y_n)$, we construct a substitution $\theta = \{X_i \mapsto Y_i \mid 1 \leq i \leq n\}$. Let $r_2 = r\theta$. Observe that $r \preceq_\theta r_2$, $|r| > |r_2|$, and $B \models r \leftrightarrow r_2$. Thus, r is recall reducible. □

Given a recall reducible rule, specialisations of the rule are not necessarily recall reducible. However, rules containing the recall reducible rule as a subrule are recall reducible. This observation is not a simple corollary of the definition of recall reducible as proving it requires invoking Theorem 1 and performing most of the construction using pigeonholed rules. The full argument is presented below:

**Proposition 5 (Recall specialisation).** Let $B$ be BK, $r_1$ be *recall reducible*, and $r_1 \subseteq r_2$. Then $r_2$ is *recall reducible*.

Proof. By Theorem 1, we know $r_1$ is also pigeonholed, thus there exists $b \subseteq \mathit{body}(r_1)$ such that all literals in $b$ have the same symbol and $B \not\models r_1 \leftrightarrow (r_1 \cup \{\mathit{alldiff}(s)\})$, where $s = \{\mathit{args}(l) \mid l \in b\}$. As in the proof of Theorem 1, there exists $c \subset b$ and a mapping $f : c \rightarrow (c \setminus b)$ such that $B \models r_1 \leftrightarrow (r_1 \cup d_p)$, where

$$d_p = \{X_i = Y_i \mid \mathit{args}(p) = (X_1, \cdots, X_n) \ \wedge \ \mathit{args}(f(p)) = (Y_1, \cdots, Y_n) \ \wedge \ 1 \leq i \leq n\}.$$

Furthermore, $B \models (r_1 \leftrightarrow (r_1 \cup d_p)) \rightarrow ((r_1 \cup e) \leftrightarrow (r_1 \cup d_p \cup e))$, where $r_2 = r_1 \cup e$. Observe that $d_p$ can be replaced by a substitution $\theta = \{X \mapsto Y \mid X = Y \in d_p\}$. Doing so results in the following: $B \models (r_1 \leftrightarrow r_1\theta) \rightarrow (r_2 \leftrightarrow r_2\theta)$. Observe that $r_2 \preceq r_2\theta$, $|r_2| > |r_2\theta|$, and given that $B \models (r_1 \leftrightarrow r_1\theta)$, we can derive that $B \models (r_2 \leftrightarrow r_2\theta)$. Thus, $r_2$ is recall reducible. □

## C Implication vs Recall Reducible

The following example illustrates how *implication* and *recall* reducible rules differ.

**Example 5.** We use the following BK with standard semantics:

$$B = \{nat/1,\ odd/1,\ prime/1,\ succ/2,\ leq/2\ \}$$

We consider these rules:

$$r_1 = h \leftarrow leq(B,C),\ succ(A,B),\ succ(A,C)$$
$$r_2 = h \leftarrow nat(A),\ succ(A,B)$$
$$r_3 = h \leftarrow succ(A,B),\ succ(A,C),\ odd(B),\ prime(C)$$

We illustrate the relationships exemplified in the follow diagram

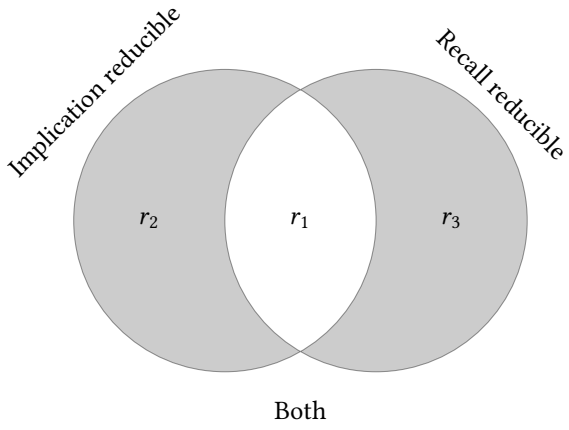


Fig. 7. The relationship between implication and recall reducible with respect to the rules introduced in Example 5

**$r_1$** : Observe that for $r_4 = r_1\{C \mapsto B\}$ and $r_5 = r_1 \setminus \{leq(B,C)\}$

$$r_4 : h \text{ :- } leq(B,B), succ(A,B).$$
$$r_5 : h \text{ :- } succ(A,B), succ(A,C).$$

both $B \models r_1 \leftrightarrow r_4$ and $B \models r_1 \leftrightarrow r_5$ hold. Furthermore $r_1 \preceq_\theta r_4$ and $|r_1| > |r_4|$ implying that $r_1$ is *recall reducible* (Definition 11). For $r_5$, observe that $leq(B,C)$ is $r_1$-captured (Definition 9), thus implying that $r$ is *Implication reducible* (Definition 10) as presented in Figure 7.

**$r_2$** : Observe that there does not exists a substitution $\theta$ such that $B \models r_2 \leftrightarrow r_2\theta$ and $|r_2| > |r_2\theta|$. However, $r_6 = r_2 \setminus \{nat(A)\}$ results in

$$r_6 : h \text{ :- } succ(A,B).$$

Observe that $B \models r_2 \leftrightarrow r_6$ as successor only applies to natural numbers. Thus we can deduce that $r_2$ is *Implication Reducible* (Definition 10), but not *recall reducible* (Definition 11) as presented in Figure 7.

**$r_3$ :** Observe that no matter which literal we remove from $r_3$ the meaning of the rule changes. For example,
- if we remove $succ(A, C)$, then $B$ need not be prime,
- if we remove $succ(A, B)$, then $C$ need not be odd, i.e. 2 is prime,
- if we remove $odd(B)$, then $B$ and $C$ need not be odd, i.e. 2 is prime, and
- if we remove $prime(C)$, then $B$ and $C$ may be divisible by 3.

Thus, $r_3$ cannot be *Implication Reducible* (Definition 10). However, applying $\{B \mapsto C\}$ to $r_3$ results in the following rule:

$$r_7 : h \text{ :- } succ(A, C), odd(C), prime(C).$$

Observe that $|r_3| > |r_7|$ and $B \models r_3 \leftrightarrow r_7$, thus implying that $r_3$ is *recall reducible* (Definition 11) as presented in Figure 7.

## D Full results Popper vs shrinker results.

Table 1 shows the full experimental results using the method described in Section 5.1.3. The Popper column denotes the learning times of Popper without shrinker. The shrinker column denotes the learning times of Popper with shrinker. The saving column shows the saving in learning time.

Table 1. Full Popper vs shrinker learning times.

| Domain | Task | Popper | shrinker | Saving | Speedup |
|---|---|---|---|---|---|
| 1d | 1d_denoising_1c | 1252 ± 332 | 253 ± 100 | 999 ± 366 | 4.95× |
| 1d | 1d_denoising_mc | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| 1d | 1d_fill | 18 ± 17 | 4 ± 3 | 14 ± 18 | 4.50× |
| 1d | 1d_flip | 3600 ± 0 | 2136 ± 681 | 1464 ± 681 | 1.69× |
| 1d | 1d_hollow | 3600 ± 0 | 1353 ± 926 | 2246 ± 926 | 2.66× |
| 1d | 1d_mirror | 928 ± 1000 | 153 ± 144 | 775 ± 1003 | 6.07× |
| 1d | 1d_move_1p | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| 1d | 1d_move_2p | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| 1d | 1d_move_2p_dp | 3350 ± 377 | 659 ± 303 | 2690 ± 334 | 5.08× |
| 1d | 1d_move_3p | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| 1d | 1d_move_dp | 3399 ± 454 | 2998 ± 919 | 400 ± 631 | 1.13× |
| 1d | 1d_padded_fill | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| 1d | 1d_pcopy_1c | 3600 ± 0 | 2732 ± 683 | 867 ± 683 | 1.32× |
| 1d | 1d_pcopy_mc | 3549 ± 114 | 728 ± 189 | 2820 ± 196 | 4.88× |
| 1d | 1d_recolor_cmp | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| 1d | 1d_recolor_cnt | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| 1d | 1d_recolor_oe | 3241 ± 812 | 3240 ± 812 | 0 ± 0 | 1.00× |
| 1d | 1d_scale_dp | 212 ± 112 | 57 ± 28 | 155 ± 84 | 3.72× |
| alzheimer | acetyl | 3600 ± 0 | 1467 ± 1053 | 2132 ± 1053 | 2.45× |
| alzheimer | amine | 523 ± 353 | 618 ± 451 | -95 ± 430 | 0.85× |
| alzheimer | mem | 979 ± 313 | 375 ± 399 | 603 ± 340 | 2.61× |
| alzheimer | toxic | 1089 ± 762 | 1067 ± 784 | 22 ± 807 | 1.02× |
| iggp | alquerque-goal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | alquerque-legal_move | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |

| Domain | Task | Popper | shrinker | Saving | Speedup |
|---|---|---|---|---|---|
| iggp | alquerque-next_cell | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | alquerque-next_control | 2904 ± 287 | 76 ± 3 | 2827 ± 284 | 38.21× |
| iggp | alquerque-next_score | 1088 ± 112 | 104 ± 4 | 984 ± 108 | 10.46× |
| iggp | alquerque-next_step | 1 ± 0 | 2 ± 0 | 0 ± 0 | 0.50× |
| iggp | alquerque-terminal | 34 ± 2 | 6 ± 0 | 28 ± 2 | 5.67× |
| iggp | asylum-goal | 3 ± 0 | 4 ± 1 | -1 ± 1 | 0.75× |
| iggp | asylum-legal_place | 3600 ± 0 | 2114 ± 105 | 1485 ± 105 | 1.70× |
| iggp | asylum-next_color | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | asylum-next_control | 214 ± 25 | 26 ± 1 | 187 ± 25 | 8.23× |
| iggp | asylum-next_location | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | asylum-next_step | 3 ± 0 | 3 ± 0 | 0 ± 0 | 1.00× |
| iggp | asylum-next_strength | 3600 ± 0 | 1256 ± 135 | 2343 ± 135 | 2.87× |
| iggp | asylum-terminal | 1 ± 0 | 2 ± 0 | 0 ± 0 | 0.50× |
| iggp | battle_of_numbers-goal | 3600 ± 0 | 272 ± 43 | 3327 ± 43 | 13.24× |
| iggp | battle_of_numbers-legal_move | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | battle_of_numbers-next_capture | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | battle_of_numbers-next_cell | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | battle_of_numbers-next_control | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | battle_of_numbers-next_step | 1 ± 0 | 2 ± 0 | 0 ± 0 | 0.50× |
| iggp | battle_of_numbers-terminal | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | breakthrough-goal | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | breakthrough-next_cell | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | breakthrough-next_control | 3600 ± 0 | 986 ± 352 | 2613 ± 352 | 3.65× |
| iggp | breakthrough-terminal | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | buttons_and_lights-goal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | buttons_and_lights-legal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | buttons_and_lights-terminal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | centipede-goal | 16 ± 1 | 1 ± 0 | 15 ± 1 | 16.00× |
| iggp | centipede-legal | 129 ± 16 | 3 ± 0 | 125 ± 16 | 43.00× |
| iggp | centipede-next_at | 3600 ± 0 | 450 ± 26 | 3149 ± 26 | 8.00× |
| iggp | centipede-next_dir | 557 ± 86 | 12 ± 1 | 544 ± 85 | 46.42× |
| iggp | centipede-terminal | 3600 ± 0 | 257 ± 72 | 3342 ± 72 | 14.01× |
| iggp | checkers-goal | 3 ± 0 | 5 ± 0 | -1 ± 0 | 0.60× |
| iggp | checkers-next_capturecount | 3600 ± 0 | 3356 ± 400 | 244 ± 400 | 1.07× |
| iggp | checkers-next_control | 3600 ± 0 | 845 ± 97 | 2754 ± 97 | 4.26× |
| iggp | checkers-next_location | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | checkers-next_step | 3 ± 0 | 3 ± 0 | 0 ± 0 | 1.00× |
| iggp | checkers-terminal | 1749 ± 521 | 383 ± 57 | 1365 ± 522 | 4.57× |
| iggp | coins-goal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | coins-legal_jump | 6 ± 0 | 3 ± 0 | 2 ± 0 | 2.00× |
| iggp | coins-next_cell | 57 ± 4 | 24 ± 0 | 32 ± 3 | 2.38× |
| iggp | coins-next_step | 5 ± 0 | 1 ± 0 | 4 ± 0 | 5.00× |
| iggp | coins-terminal | 20 ± 3 | 2 ± 0 | 17 ± 3 | 10.00× |
| iggp | connect4team-goal | 3600 ± 0 | 133 ± 6 | 3466 ± 6 | 27.07× |
| iggp | connect4team-legal_drop | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |

| Domain | Task | Popper | shrinker | Saving | Speedup |
|---|---|---|---|---|---|
| iggp | connect4team-next_cell | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | connect4team-next_control | 1435 ± 220 | 19 ± 1 | 1416 ± 221 | 75.53× |
| iggp | connect4team-terminal | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | dont_touch-goal | 273 ± 37 | 9 ± 0 | 264 ± 36 | 30.33× |
| iggp | dont_touch-legal_mark | 15 ± 0 | 8 ± 0 | 7 ± 1 | 1.88× |
| iggp | dont_touch-next_cell | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | dont_touch-next_control | 3600 ± 0 | 11 ± 1 | 3588 ± 1 | 327.27× |
| iggp | dont_touch-terminal | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | duikoshi-goal | 88 ± 6 | 4 ± 0 | 84 ± 6 | 22.00× |
| iggp | duikoshi-legal_mark | 3600 ± 0 | 95 ± 9 | 3504 ± 9 | 37.89× |
| iggp | duikoshi-next_cell | 3600 ± 0 | 715 ± 16 | 2884 ± 16 | 5.03× |
| iggp | duikoshi-next_control | 3600 ± 0 | 2 ± 0 | 3597 ± 0 | 1800.00× |
| iggp | duikoshi-terminal | 3600 ± 0 | 168 ± 32 | 3431 ± 32 | 21.43× |
| iggp | eight_puzzle-goal | 66 ± 3 | 9 ± 0 | 57 ± 3 | 7.33× |
| iggp | eight_puzzle-legal_move | 349 ± 27 | 3 ± 0 | 345 ± 27 | 116.33× |
| iggp | eight_puzzle-next_cell | 3600 ± 0 | 695 ± 20 | 2904 ± 20 | 5.18× |
| iggp | eight_puzzle-next_step | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | eight_puzzle-terminal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | farming-goal | 3 ± 0 | 4 ± 1 | -1 ± 1 | 0.75× |
| iggp | farming-legal_arson_col | 615 ± 92 | 83 ± 8 | 531 ± 94 | 7.41× |
| iggp | farming-legal_arson_row | 677 ± 47 | 78 ± 11 | 599 ± 44 | 8.68× |
| iggp | farming-legal_harvest_col | 31 ± 2 | 14 ± 2 | 16 ± 3 | 2.21× |
| iggp | farming-legal_harvest_row | 28 ± 2 | 15 ± 2 | 12 ± 3 | 1.87× |
| iggp | farming-legal_plow_col | 28 ± 3 | 12 ± 3 | 16 ± 5 | 2.33× |
| iggp | farming-legal_plow_row | 28 ± 2 | 13 ± 4 | 14 ± 5 | 2.15× |
| iggp | farming-legal_sow_col | 9 ± 2 | 8 ± 0 | 1 ± 2 | 1.12× |
| iggp | farming-legal_sow_row | 9 ± 2 | 8 ± 1 | 0 ± 3 | 1.12× |
| iggp | farming-legal_water_col | 28 ± 3 | 15 ± 2 | 13 ± 3 | 1.87× |
| iggp | farming-legal_water_row | 29 ± 2 | 12 ± 2 | 16 ± 3 | 2.42× |
| iggp | farming-next_control | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | farming-next_has_arson | 446 ± 46 | 11 ± 0 | 435 ± 46 | 40.55× |
| iggp | farming-next_plowed | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | farming-next_ripe | 137 ± 6 | 21 ± 1 | 116 ± 8 | 6.52× |
| iggp | farming-next_score | 3435 ± 290 | 166 ± 19 | 3268 ± 273 | 20.69× |
| iggp | farming-next_season | 3600 ± 0 | 3467 ± 200 | 132 ± 200 | 1.04× |
| iggp | farming-next_sown | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | farming-next_step | 3 ± 0 | 4 ± 0 | 0 ± 0 | 0.75× |
| iggp | farming-next_year_first_player | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | farming-next_year_second_player | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | farming-terminal | 1 ± 0 | 2 ± 0 | 0 ± 0 | 0.50× |
| iggp | firesheep-goal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | firesheep-legal_burn | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | firesheep-legal_force_noop | 3600 ± 0 | 9 ± 0 | 3590 ± 0 | 400.00× |
| iggp | firesheep-legal_freeze | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | firesheep-legal_kill | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |

| Domain | Task | Popper | shrinker | Saving | Speedup |
|---|---|---|---|---|---|
| iggp | firesheep-next_at | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | firesheep-next_burning | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | firesheep-next_forced | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | firesheep-next_frozen | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | firesheep-next_grass | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | firesheep-next_grass_last_turn | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | firesheep-next_has_force_noop | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | firesheep-next_has_kill | 3600 ± 0 | 268 ± 23 | 3331 ± 23 | 13.43× |
| iggp | firesheep-next_score | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | firesheep-next_sheep | 3600 ± 0 | 1595 ± 172 | 2004 ± 172 | 2.26× |
| iggp | firesheep-terminal | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | fizzbuzz-goal | 235 ± 25 | 14 ± 1 | 221 ± 24 | 16.79× |
| iggp | fizzbuzz-legal_say | 3600 ± 0 | 934 ± 292 | 2665 ± 292 | 3.85× |
| iggp | fizzbuzz-next_count | 1 ± 0 | 2 ± 0 | -1 ± 0 | 0.50× |
| iggp | fizzbuzz-next_success | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | fizzbuzz-terminal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | forager2-goal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | forager2-legal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | forager2-next_at | 3600 ± 0 | 2450 ± 684 | 1149 ± 684 | 1.47× |
| iggp | forager2-next_score | 3600 ± 0 | 3240 ± 814 | 359 ± 814 | 1.11× |
| iggp | forager2-next_time | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | forager2-terminal | 3240 ± 814 | 2171 ± 733 | 1069 ± 958 | 1.49× |
| iggp | freeforall-goal | 2 ± 0 | 1 ± 0 | 0 ± 0 | 2.00× |
| iggp | freeforall-next_capture | 3600 ± 0 | 346 ± 25 | 3253 ± 25 | 10.40× |
| iggp | freeforall-next_cell | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | freeforall-next_control | 1 ± 0 | 3 ± 0 | -2 ± 0 | 0.33× |
| iggp | freeforall-next_step | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | freeforall-terminal | 1 ± 0 | 2 ± 0 | -1 ± 0 | 0.50× |
| iggp | frogs_and_toads-legal_jump | 159 ± 4 | 161 ± 8 | -2 ± 6 | 0.99× |
| iggp | frogs_and_toads-legal_move | 155 ± 5 | 159 ± 5 | -3 ± 7 | 0.97× |
| iggp | frogs_and_toads-next_cell | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | frogs_and_toads-next_correctFrogs | 285 ± 57 | 92 ± 7 | 192 ± 57 | 3.10× |
| iggp | frogs_and_toads-next_correctToads | 527 ± 48 | 145 ± 13 | 381 ± 54 | 3.63× |
| iggp | frogs_and_toads-next_step | 2 ± 0 | 2 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_attrition-goal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_attrition-legal | 2 ± 0 | 1 ± 0 | 1 ± 0 | 2.00× |
| iggp | gt_attrition-next_claim_made_by | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_attrition-next_control | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_attrition-next_score | 2278 ± 317 | 47 ± 1 | 2230 ± 318 | 48.47× |
| iggp | gt_attrition-terminal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_centipede-goal | 13 ± 1 | 1 ± 0 | 11 ± 1 | 13.00× |
| iggp | gt_centipede-legal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_centipede-next_blackPayoff | 7 ± 1 | 2 ± 0 | 4 ± 1 | 3.50× |
| iggp | gt_centipede-next_control | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_centipede-next_whitePayoff | 2 ± 0 | 1 ± 0 | 1 ± 0 | 2.00× |

| Domain | Task | Popper | shrinker | Saving | Speedup |
|---|---|---|---|---|---|
| iggp | gt_centipede-terminal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_chicken-goal | 5 ± 0 | 5 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_chicken-legal | 125 ± 12 | 34 ± 3 | 91 ± 10 | 3.68× |
| iggp | gt_chicken-next_blackScore | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_chicken-next_rounds | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_chicken-next_whiteScore | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_chicken-terminal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_prisoner-goal | 7 ± 0 | 5 ± 0 | 1 ± 0 | 1.40× |
| iggp | gt_prisoner-legal | 145 ± 22 | 36 ± 5 | 108 ± 18 | 4.03× |
| iggp | gt_prisoner-next_blackScore | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_prisoner-next_maxRounds | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_prisoner-next_rounds | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_prisoner-next_whiteScore | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_prisoner-terminal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_ultimatum-goal | 18 ± 1 | 14 ± 1 | 3 ± 0 | 1.29× |
| iggp | gt_ultimatum-legal_offer | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_ultimatum-next_blackScore | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_ultimatum-next_control | 332 ± 33 | 129 ± 9 | 203 ± 26 | 2.57× |
| iggp | gt_ultimatum-next_offered | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_ultimatum-next_rounds | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_ultimatum-next_whiteScore | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | gt_ultimatum-terminal | 2 ± 1 | 3 ± 0 | -1 ± 1 | 0.67× |
| iggp | hexforthree-goal | 3600 ± 0 | 427 ± 20 | 3172 ± 20 | 8.43× |
| iggp | hexforthree-legal_place | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | hexforthree-next_cell | 10 ± 0 | 10 ± 0 | 0 ± 0 | 1.00× |
| iggp | hexforthree-next_connected | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | hexforthree-next_control | 3600 ± 0 | 112 ± 26 | 3487 ± 26 | 32.14× |
| iggp | hexforthree-next_owner | 22 ± 0 | 16 ± 1 | 5 ± 1 | 1.38× |
| iggp | hexforthree-next_step | 3 ± 0 | 3 ± 0 | 0 ± 0 | 1.00× |
| iggp | hexforthree-terminal | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | horseshoe-goal | 304 ± 41 | 2 ± 0 | 301 ± 41 | 152.00× |
| iggp | horseshoe-legal_move | 3600 ± 0 | 191 ± 4 | 3408 ± 4 | 18.85× |
| iggp | horseshoe-next_cell | 3600 ± 0 | 76 ± 2 | 3523 ± 2 | 47.37× |
| iggp | horseshoe-next_control | 3600 ± 0 | 30 ± 0 | 3569 ± 0 | 120.00× |
| iggp | horseshoe-next_step | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | horseshoe-terminal | 3600 ± 0 | 47 ± 6 | 3552 ± 6 | 76.60× |
| iggp | hunter-goal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | hunter-legal_move | 3600 ± 0 | 2095 ± 292 | 1505 ± 292 | 1.72× |
| iggp | hunter-next_captures | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | hunter-next_cell | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | hunter-next_step | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | hunter-terminal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | knights_tour-goal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | knights_tour-legal_move | 3600 ± 0 | 1115 ± 85 | 2485 ± 85 | 3.23× |
| iggp | knights_tour-next_cell | 3600 ± 0 | 680 ± 50 | 2919 ± 50 | 5.29× |

| Domain | Task | Popper | shrinker | Saving | Speedup |
|---|---|---|---|---|---|
| iggp | knights_tour-next_moveCount | 2 ± 1 | 1 ± 0 | 0 ± 1 | 2.00× |
| iggp | knights_tour-terminal | 84 ± 26 | 94 ± 47 | -9 ± 57 | 0.89× |
| iggp | kono-goal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | kono-next_cell | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | kono-next_control | 2212 ± 255 | 32 ± 2 | 2180 ± 253 | 69.12× |
| iggp | kono-next_score | 2452 ± 439 | 260 ± 12 | 2192 ± 433 | 9.43× |
| iggp | kono-next_step | 1 ± 0 | 2 ± 0 | -1 ± 0 | 0.50× |
| iggp | kono-terminal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | leafy-goal | 3600 ± 0 | 509 ± 24 | 3090 ± 24 | 7.07× |
| iggp | leafy-legal_move | 3600 ± 0 | 73 ± 6 | 3526 ± 6 | 49.32× |
| iggp | leafy-next_isplayer | 3 ± 0 | 3 ± 0 | 0 ± 0 | 1.00× |
| iggp | leafy-next_leaf | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | leafy-terminal | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | lightboard-goal | 111 ± 15 | 1 ± 0 | 110 ± 15 | 111.00× |
| iggp | lightboard-legal_toggle | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | lightboard-next_on | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | lightboard-next_step | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | lightboard-terminal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | minimal_decay-legal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | minimal_decay-next_value | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | minimal_even-goal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | minimal_even-legal_choose | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | minimal_even-next_chosen | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | minimal_even-terminal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | multiplebuttonsandlights-goal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | multiplebuttonsandlights-legal_a | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | multiplebuttonsandlights-legal_b | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | multiplebuttonsandlights-legal_c | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | multiplebuttonsandlights-next_p | 3600 ± 0 | 236 ± 26 | 3363 ± 26 | 15.25× |
| iggp | multiplebuttonsandlights-next_q | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | multiplebuttonsandlights-next_step | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | multiplebuttonsandlights-terminal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | nineboardtictactoe-goal | 11 ± 0 | 3 ± 0 | 8 ± 0 | 3.67× |
| iggp | nineboardtictactoe-legal_play | 421 ± 33 | 84 ± 2 | 337 ± 33 | 5.01× |
| iggp | nineboardtictactoe-next_control | 3 ± 0 | 1 ± 0 | 1 ± 0 | 3.00× |
| iggp | nineboardtictactoe-next_currentboard | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | nineboardtictactoe-next_mark | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | nineboardtictactoe-terminal | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | pentago-goal | 3600 ± 0 | 93 ± 10 | 3506 ± 10 | 38.71× |
| iggp | pentago-legal_place | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | pentago-legal_rotate | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | pentago-next_cellholds | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | pentago-next_placecontrol | 3600 ± 0 | 219 ± 57 | 3380 ± 57 | 16.44× |
| iggp | pentago-next_rotatecontrol | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | pentago-terminal | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |

| Domain | Task | Popper | shrinker | Saving | Speedup |
|---|---|---|---|---|---|
| iggp | pilgrimage-goal | 2613 ± 245 | 34 ± 2 | 2578 ± 244 | 76.85× |
| iggp | pilgrimage-legal_move | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | pilgrimage-legal_raise | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | pilgrimage-next_builder | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | pilgrimage-next_cell | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | pilgrimage-next_control | 3600 ± 0 | 175 ± 9 | 3424 ± 9 | 20.57× |
| iggp | pilgrimage-next_moves | 386 ± 14 | 20 ± 1 | 366 ± 14 | 19.30× |
| iggp | pilgrimage-next_phase | 3600 ± 0 | 471 ± 38 | 3128 ± 38 | 7.64× |
| iggp | pilgrimage-next_pilgrim | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | platformjumpers-goal | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | platformjumpers-legal_col | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | platformjumpers-legal_row | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | platformjumpers-next_cell | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | platformjumpers-next_coled | 39 ± 2 | 38 ± 1 | 1 ± 2 | 1.03× |
| iggp | platformjumpers-next_control | 1327 ± 793 | 338 ± 39 | 989 ± 795 | 3.93× |
| iggp | platformjumpers-next_jumper | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | platformjumpers-next_rowed | 39 ± 2 | 36 ± 1 | 2 ± 3 | 1.08× |
| iggp | platformjumpers-terminal | 5 ± 0 | 8 ± 0 | -3 ± 0 | 0.62× |
| iggp | rainbow-goal | 11 ± 0 | 1 ± 0 | 10 ± 0 | 11.00× |
| iggp | rainbow-legal_mark | 321 ± 44 | 2 ± 0 | 318 ± 44 | 160.50× |
| iggp | rainbow-next_color | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | rainbow-terminal | 1252 ± 419 | 2 ± 0 | 1249 ± 419 | 626.00× |
| iggp | scissors_paper_stone-goal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | scissors_paper_stone-legal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | scissors_paper_stone-next_score | 353 ± 46 | 51 ± 1 | 302 ± 45 | 6.92× |
| iggp | scissors_paper_stone-next_step | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | scissors_paper_stone-terminal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | sheep_and_wolf-goal | 3 ± 0 | 2 ± 0 | 0 ± 0 | 1.50× |
| iggp | sheep_and_wolf-next_cell | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | sheep_and_wolf-next_control | 3600 ± 0 | 778 ± 58 | 2821 ± 58 | 4.63× |
| iggp | sheep_and_wolf-terminal | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | sokoban-goal | 4 ± 0 | 1 ± 0 | 3 ± 0 | 4.00× |
| iggp | sokoban-legal | 235 ± 17 | 12 ± 0 | 223 ± 17 | 19.58× |
| iggp | sokoban-next_at | 3600 ± 0 | 2372 ± 336 | 1227 ± 336 | 1.52× |
| iggp | sokoban-next_target | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | sokoban-terminal | 3600 ± 0 | 647 ± 288 | 2952 ± 288 | 5.56× |
| iggp | sudoku-goal | 7 ± 0 | 8 ± 0 | 0 ± 0 | 0.88× |
| iggp | sudoku-legal_mark | 3600 ± 0 | 442 ± 21 | 3158 ± 21 | 8.14× |
| iggp | sudoku-next_cell | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | sudoku-terminal | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | sukoshi-goal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | sukoshi-legal_mark | 38 ± 2 | 1 ± 0 | 36 ± 2 | 38.00× |
| iggp | sukoshi-next_cell | 472 ± 41 | 120 ± 3 | 351 ± 40 | 3.93× |
| iggp | sukoshi-terminal | 3058 ± 318 | 333 ± 419 | 2724 ± 538 | 9.18× |
| iggp | switches-legal | 96 ± 10 | 2 ± 0 | 94 ± 10 | 48.00× |

| Domain | Task | Popper | shrinker | Saving | Speedup |
|---|---|---|---|---|---|
| iggp | switches-next_at | 3600 ± 0 | 252 ± 29 | 3347 ± 29 | 14.29× |
| iggp | switches-next_open | 8 ± 1 | 1 ± 0 | 6 ± 1 | 8.00× |
| iggp | switches-next_switch | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | switches-next_target | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | tictactoe-goal | 58 ± 6 | 1 ± 0 | 57 ± 6 | 58.00× |
| iggp | tictactoe-legal_mark | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | tictactoe-next_cell | 3600 ± 0 | 325 ± 28 | 3274 ± 28 | 11.08× |
| iggp | tictactoe-next_control | 946 ± 179 | 1 ± 0 | 945 ± 179 | 946.00× |
| iggp | tictactoe-terminal | 3600 ± 0 | 183 ± 34 | 3416 ± 34 | 19.67× |
| iggp | tiger_vs_dogs-goal | 19 ± 1 | 3 ± 0 | 15 ± 1 | 6.33× |
| iggp | tiger_vs_dogs-legal_move | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | tiger_vs_dogs-next_cell | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | tiger_vs_dogs-next_control | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | tron-goal | 147 ± 17 | 10 ± 0 | 137 ± 17 | 14.70× |
| iggp | tron-legal | 934 ± 125 | 24 ± 2 | 909 ± 123 | 38.92× |
| iggp | tron-next_at | 3600 ± 0 | 697 ± 32 | 2902 ± 32 | 5.16× |
| iggp | tron-next_marked | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | tron-terminal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | ttcc4-goal | 3600 ± 0 | 175 ± 6 | 3424 ± 6 | 20.57× |
| iggp | ttcc4-legal_checkermove | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | ttcc4-legal_drop | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | ttcc4-legal_knightmove | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | ttcc4-legal_pawnmove | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | ttcc4-next_cell | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | ttcc4-next_control | 3 ± 0 | 3 ± 0 | 0 ± 0 | 1.00× |
| iggp | ttcc4-next_step | 2 ± 0 | 4 ± 0 | -2 ± 0 | 0.50× |
| iggp | ttcc4-terminal | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| iggp | untwisty_corridor-goal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | untwisty_corridor-legal | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | untwisty_corridor-next_step | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| iggp | walkabout-goal | 81 ± 13 | 3 ± 0 | 78 ± 13 | 27.00× |
| iggp | walkabout-legal | 646 ± 99 | 3 ± 0 | 642 ± 99 | 215.33× |
| iggp | walkabout-next_at | 3555 ± 99 | 106 ± 3 | 3449 ± 98 | 33.54× |
| iggp | walkabout-terminal | 7 ± 2 | 1 ± 0 | 6 ± 2 | 7.00× |
| imdb | imdb1 | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| imdb | imdb2 | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| imdb | imdb3 | 50 ± 31 | 34 ± 36 | 15 ± 36 | 1.47× |
| jr | 001 | 2 ± 1 | 2 ± 1 | 0 ± 1 | 1.00× |
| jr | 002 | 2 ± 0 | 1 ± 0 | 0 ± 0 | 2.00× |
| jr | 003 | 3 ± 1 | 2 ± 0 | 1 ± 1 | 1.50× |
| jr | 004 | 3 ± 1 | 2 ± 0 | 0 ± 1 | 1.50× |
| jr | 005 | 2 ± 1 | 1 ± 0 | 0 ± 1 | 2.00× |
| jr | 006 | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| jr | 007 | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| jr | 008 | 3 ± 1 | 1 ± 0 | 1 ± 1 | 3.00× |

| Domain | Task | Popper | shrinker | Saving | Speedup |
|---|---|---|---|---|---|
| jr | 009 | 3 ± 0 | 1 ± 0 | 1 ± 0 | 3.00× |
| jr | 010 | 3551 ± 109 | 1509 ± 617 | 2041 ± 628 | 2.35× |
| jr | 011 | 28 ± 10 | 10 ± 2 | 17 ± 8 | 2.80× |
| jr | 012 | 16 ± 4 | 7 ± 1 | 9 ± 4 | 2.29× |
| jr | 013 | 1057 ± 428 | 190 ± 121 | 867 ± 460 | 5.56× |
| jr | 014 | 532 ± 251 | 141 ± 45 | 391 ± 244 | 3.77× |
| jr | 015 | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| jr | 016 | 218 ± 35 | 21 ± 2 | 196 ± 33 | 10.38× |
| jr | 017 | 1087 ± 513 | 46 ± 9 | 1040 ± 505 | 23.63× |
| jr | 018 | 242 ± 23 | 26 ± 2 | 215 ± 22 | 9.31× |
| jr | 019 | 680 ± 115 | 44 ± 2 | 635 ± 115 | 15.45× |
| jr | 020 | 318 ± 124 | 61 ± 25 | 257 ± 128 | 5.21× |
| jr | 021 | 74 ± 22 | 9 ± 1 | 64 ± 21 | 8.22× |
| jr | 022 | 68 ± 18 | 9 ± 2 | 59 ± 17 | 7.56× |
| jr | 023 | 3600 ± 0 | 3148 ± 682 | 451 ± 682 | 1.14× |
| jr | 024 | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| jr | 025 | 40 ± 4 | 11 ± 1 | 28 ± 3 | 3.64× |
| jr | 026 | 485 ± 298 | 85 ± 59 | 400 ± 313 | 5.71× |
| jr | 027 | 3600 ± 0 | 3588 ± 26 | 11 ± 26 | 1.00× |
| jr | 028 | 3600 ± 0 | 3190 ± 625 | 409 ± 625 | 1.13× |
| jr | 029 | 1 ± 0 | 2 ± 1 | 0 ± 1 | 0.50× |
| jr | 030 | 97 ± 52 | 68 ± 26 | 29 ± 61 | 1.43× |
| jr | 031 | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| jr | 032 | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| jr | 033 | 1146 ± 604 | 174 ± 74 | 971 ± 606 | 6.59× |
| jr | 034 | 229 ± 26 | 40 ± 1 | 189 ± 24 | 5.72× |
| jr | 035 | 74 ± 17 | 11 ± 1 | 62 ± 16 | 6.73× |
| jr | 036 | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| jr | 037 | 2 ± 0 | 1 ± 0 | 1 ± 0 | 2.00× |
| jr | 038 | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| jr | 039 | 1977 ± 1067 | 123 ± 26 | 1854 ± 1057 | 16.07× |
| jr | 040 | 382 ± 202 | 27 ± 3 | 355 ± 200 | 14.15× |
| jr | 041 | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| jr | 042 | 227 ± 44 | 29 ± 6 | 198 ± 39 | 7.83× |
| jr | 043 | 3600 ± 0 | 352 ± 15 | 3247 ± 15 | 10.23× |
| jr | 044 | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| jr | 045 | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| jr | 046 | 10 ± 1 | 1 ± 0 | 9 ± 1 | 10.00× |
| jr | 047 | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| jr | 048 | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| jr | 049 | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| jr | 050 | 6 ± 0 | 1 ± 0 | 5 ± 0 | 6.00× |
| jr | 051 | 451 ± 296 | 72 ± 30 | 379 ± 281 | 6.26× |
| jr | 052 | 48 ± 33 | 13 ± 5 | 34 ± 34 | 3.69× |
| jr | 053 | 59 ± 8 | 15 ± 1 | 44 ± 8 | 3.93× |

| Domain | Task | POPPER | SHRINKER | Saving | Speedup |
|---|---|---|---|---|---|
| jr | 054 | 82 ± 22 | 12 ± 1 | 70 ± 20 | 6.83× |
| jr | 055 | 1548 ± 839 | 558 ± 304 | 990 ± 884 | 2.77× |
| jr | 056 | 801 ± 802 | 103 ± 41 | 698 ± 792 | 7.78× |
| jr | 057 | 271 ± 113 | 113 ± 66 | 158 ± 100 | 2.40× |
| jr | 058 | 3 ± 1 | 2 ± 1 | 0 ± 2 | 1.50× |
| jr | 059 | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| jr | 060 | 1090 ± 491 | 46 ± 4 | 1044 ± 487 | 23.70× |
| jr | 061 | 69 ± 44 | 23 ± 6 | 45 ± 49 | 3.00× |
| jr | 062 | 3 ± 1 | 2 ± 0 | 1 ± 1 | 1.50× |
| jr | 063 | 3600 ± 0 | 3439 ± 362 | 160 ± 362 | 1.05× |
| jr | 064 | 106 ± 51 | 69 ± 32 | 37 ± 71 | 1.54× |
| jr | 065 | 34 ± 7 | 7 ± 1 | 26 ± 6 | 4.86× |
| jr | 066 | 624 ± 355 | 126 ± 74 | 498 ± 335 | 4.95× |
| jr | 067 | 2797 ± 577 | 362 ± 101 | 2435 ± 549 | 7.73× |
| jr | 068 | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| jr | 069 | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| jr | 070 | 6 ± 2 | 2 ± 0 | 3 ± 2 | 3.00× |
| jr | 071 | 9 ± 3 | 4 ± 1 | 4 ± 4 | 2.25× |
| jr | 072 | 161 ± 130 | 80 ± 34 | 81 ± 115 | 2.01× |
| jr | 073 | 2 ± 1 | 1 ± 0 | 1 ± 2 | 2.00× |
| jr | 074 | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| jr | 075 | 81 ± 39 | 35 ± 18 | 45 ± 37 | 2.31× |
| jr | 076 | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| jr | 077 | 1 ± 0 | 1 ± 0 | 0 ± 0 | 1.00× |
| jr | 078 | 3600 ± 0 | 2461 ± 937 | 1138 ± 937 | 1.46× |
| jr | 079 | 3600 ± 0 | 1515 ± 494 | 2084 ± 494 | 2.38× |
| jr | 080 | 514 ± 524 | 205 ± 145 | 308 ± 488 | 2.51× |
| trains | trains1 | 3 ± 2 | 2 ± 1 | 1 ± 1 | 1.50× |
| trains | trains2 | 17 ± 10 | 2 ± 0 | 15 ± 10 | 8.50× |
| trains | trains3 | 191 ± 100 | 25 ± 27 | 166 ± 95 | 7.64× |
| trains | trains4 | 2523 ± 1239 | 149 ± 90 | 2374 ± 1170 | 16.93× |
| zendo | zendo1 | 41 ± 43 | 10 ± 5 | 31 ± 41 | 4.10× |
| zendo | zendo2 | 3600 ± 0 | 3600 ± 0 | 0 ± 0 | 1.00× |
| zendo | zendo3 | 3600 ± 0 | 3583 ± 36 | 16 ± 36 | 1.00× |
| zendo | zendo4 | 3600 ± 0 | 3499 ± 220 | 100 ± 220 | 1.03× |

Received 22 January 2026; accepted 18 March 2026