Knowledge Refactoring for Inductive Program Synthesis

Sebastijan Dumancic, [1] **Tias Guns,** [1] **Andrew Cropper** [2]
[1] KU Leuven, Belgium
[2] Oxford University, United Kingdom
sebastijan.dumancic@cs.kuleuven.be, tias.guns@kuleuven.be, andrew.cropper@cs.ox.ac.uk

Abstract

Humans constantly restructure knowledge to use it more efficiently. Our goal is to give a machine learning system similar abilities so that it can learn more efficiently. We introduce the *knowledge refactoring* problem, where the goal is to restructure a learner's knowledge base to reduce its size and to minimise redundancy in it. We focus on inductive logic programming, where the knowledge base is a logic program. We introduce `Knorf`, a system which solves the refactoring problem using constraint optimisation. A key feature of `Knorf` is that, rather than simply removing knowledge, it also introduces new knowledge through *predicate invention*. We evaluate our approach on two domains: building Lego structures and real-world string transformations. Our experiments show that learning from refactored knowledge can improve predictive accuracies fourfold and reduce learning times by half.

Introduction

According to Rumelhart and Norman (1976), humans exhibit three modes of learning. *Learning by accretion* is an everyday kind of learning which merely increments a person's knowledge base with new facts. *Learning by tuning* involves changes in the categories people use for interpreting new information. For instance, the process of tuning specialises a child's interpretation of the word 'doggie' from all animals to dogs only. *Restructuring* devises new memory structures and organisation of already stored knowledge, which in turn allows for better accessibility of the acquired knowledge. This restructuring ability is the most significant mode and is what separates well-performing individuals from others (Karmiloff-Smith 1992; Stern 2005).

The key to effective restructuring is finding appropriate abstractions. As a running example, consider building Lego structures. Figure 1 (top) shows two structures built using only two types of bricks: short ▯ and long ▭. Building the structures using only these two types of bricks is complex and requires 29 and 18 bricks respectively. However, as Figure 1 (bottom) shows, by introducing new types of bricks through restructuring, such as pillar and horizontal bricks of various lengths, we can build the same structures using only 7 and 11 pieces respectively. In other words, by finding suitable abstractions, we can make the structures, and potentially future structures, easier and faster to build.

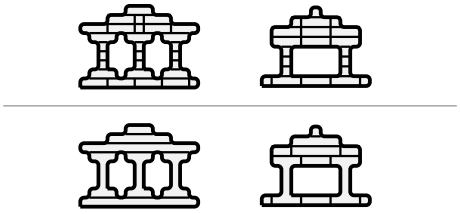


Figure 1: Complex Lego arcades (top, built with only two distinct Lego bricks: ▯ and ▭) become easier to build after new, useful types of bricks are introduced (bottom). The complexity is measured as the number of pieces needed for the construction.

While the importance of abstraction in AI is well-known (Saitta and Zucker 2013), the majority of learning AI agents merely accumulate knowledge, which can be detrimental to learning performance (Srinivasan, King, and Bain 2003; Cropper 2020). In other words, as knowledge is a form of inductive bias in machine learning (Mitchell 1997), increasing the amount of knowledge increases the hypothesis space and consequently makes finding the target hypothesis more difficult. The challenge is, therefore, to choose a learner's inductive bias (knowledge) so that the hypothesis space is large enough to contain the target hypothesis, yet small enough to be efficiently searched.

This paper aims to tackle the inductive bias problem by (i) reducing the size of the knowledge base, and (ii) restructuring it to make it easier to learn from. Rather than only adding or removing knowledge (De Raedt et al. 2008; Lin et al. 2014; Cropper 2020), we argue that the human-like ability to *restructure knowledge* can provide a better inductive bias to a learner and thus improve performance. We call this problem *knowledge refactoring*. The idea is similar to program refactoring, where the goal of a programmer is to identify a good set of support functions to make a program more compact and reusable.

To restructure knowledge, we must explicitly store it. This requirement eliminates non-symbolic learning approaches which dissipate knowledge in the parameters of a model. We, therefore, use symbolic learning approaches, specifically

inductive program synthesis (Shapiro 1982), which learns programs from input-output examples. We focus on inductive logic programming (ILP) (Muggleton and De Raedt 1994), which represents background knowledge (BK) as a logic program and which has strong foundations in knowledge representation.

Our specific contributions are:

- We introduce the *knowledge refactoring* problem: revising a learner's knowledge base (a logic program) to reduce its size and minimise redundancy. Our key idea is to automatically identify useful substructures via predicate invention. The challenge lies in efficiently identifying substructures that lead to smaller programs. We tackle this challenge by casting the problem of knowledge refactoring as a constraint optimisation problem over a large set of candidate invented predicates.
- We introduce `Knorf`[1], a system that refactors knowledge bases by searching for new, reusable pieces of knowledge. A key feature of `Knorf` is that, rather than simply removing knowledge, it also introduces new knowledge through *predicate invention* (Stahl 1993).
- We evaluate our approach on two domains: building Lego structures and real-word string transformations. Our experiments show that learning from refactored knowledge can substantially improve the predictive accuracies of an ILP system and reduce learning times.

Related Work

Redundancy elimination. Reducing redundancy is useful in many areas of AI, such as to improve SAT efficiency (Heule et al. 2015). In machine learning, irrelevant and redundant knowledge is detrimental to learning performance (Srinivasan, Muggleton, and King 1995; Srinivasan, King, and Bain 2003; Cropper and Tourret 2020). Much work focuses on removing redundant literals or clauses from a logical theory (Plotkin 1971). Theory *minimisation* approaches try to find a minimum equivalent formula to a given input propositional formula (Hemaspaandra and Schnoor 2011) and also introduce new formulas. By contrast, we focus on first-order (Horn) logic. *Forgetting* approaches (Cropper 2020) try to remove clauses from the knowledge base to improve learning performance. Our work is different because we (i) restructure knowledge, and (ii) introduce new knowledge through predicate invention.

Theory refinement. Theory *refinement* (Wrobel 1996) aims to improve the quality of a theory. Theory *revision* approaches (De Raedt 1992; Adé, Malfait, and De Raedt 1994; Richards and Mooney 1995) revise a program so that it entails missing answers or does not entail incorrect answers. Theory *compression* (De Raedt et al. 2008) approaches select a subset of clauses such that the performance is minimally affected with respect to certain examples. By contrast, our approach does not consider examples: we only consider the knowledge base. Theory *restructuring* changes the structure of a logic program to optimise its execution or its readability (Wrobel 1996). For instance, FENDER (Sommer 1995) restructures a theory with intra- and inter-construction operators (Muggleton 1995). The authors claim that their approach leads to a theory that is deeper, more modular, and possibly easier to understand and maintain. By contrast, our goal is to restructure a theory by reducing the number of unnecessary predicate symbols in it and by introducing new ones. Moreover, we formulate the refactoring problem as a constraint optimisation problem (COP).

[1]Code available at https://github.com/sebdumancic/knorf_aaai21

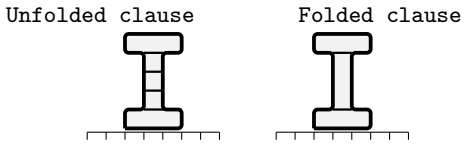


Figure 2: Construction of the pillar (left) can be simplified by *abstracting* (or *folding* in LP) the procedure for constructing the vertical piece in the middle (right).

Predicate invention. `Knorf` supports *predicate invention* (Stahl 1993), the automatic introduction of new auxiliary predicates. In contrast to the existing approaches which invent predicates before (Cropper; Hocquette and Muggleton 2020) or during (Muggleton, Lin, and Tamaddoni-Nezhad 2015) learning, `Knorf` invents them after learning through refactoring. Three approaches are especially relevant to us. Alps (Dumančić et al. 2019) invents predicates by compressing a knowledge base formed of facts. By contrast, `Knorf` considers definite clauses with more than one literal. EC (Dechter et al. 2013) learns programs such that they are compressible but does not revise previously invented abstractions, while `Knorf` does. EC^2 (Ellis et al. 2018), building upon EC, locally searches for small changes to a functional program to increase an optimisation function. Our approach differs because (i) we work in a purely logical setting, (ii) we preserve the semantics of the original program, and (iii) we solve the refactoring problem as a COP.

Problem Description

To introduce the knowledge refactoring problem, we first provide essential preliminaries on logic programming (LP) (Sterling and Shapiro 1986), after which we show how knowledge refactoring can aid inductive program synthesis.

Logic Programming

A definite logic program is a set of definite clauses of the form `head :- cond`$_1$`, ..., cond`$_N$`.` A clause states that head is true if all conditions are true. The head and conditions are **atoms** or their negations (jointly called literals), i.e., structured terms that represent relations between objects. In the Lego example, `place(`▢`,Po,E,E′)` is an atom, consisting of a predicate `place/4`, which *places a brick of the type* ▢ *at a position* `Po` *in a world with the state* `E`*, resulting in a new state* `E′`. Assume a one-dimensional world with Lego pieces and a cursor indicating the current position (Figure 2). The clause:

```
pillar(X,Y,E,E′) :-
    place(▬,X,E,E1), right(X,Z),
    place(▢,Z,E1,E2), place(▢,Z,E2,E3),
```

```
    place(□,Z,E3,E4), left(Z,Y),
    place(▭,X,E4,E′).
```

provides instructions for constructing a pillar at the position X: place a horizontal brick on position X, move the cursor to the position right of X, put three bricks on top of each other, move the cursor back to the starting position and place another horizontal brick (Figure 2, left).

A key concept in LP is (un)folding (Tamaki and Sato 1984). Intuitively, given a set of clauses S, the $fold$(P,S) operation replaces every occurrence of the body of clause c ∈ S, up to variable renaming, in program P with its head. For instance, folding the clause 1 with the clause:

```
2| ver(X,E,E′) :-
     place(□,X,E,E1), place(□,X,E1,E2),
     place(□,X,E2,E′).
```

results in the clause:

```
3| pillar(X,Y,E,E′) :-
     place(▭,X,E,E1), right(X,Z),
     ver(Z,E1,E2), left(Z,Y),
     place(▭,X,E2,E′).
```

The $unfold$(P) operation essentially inlines all functions: for every clause c in program P which defines a predicate that is used in the body of another clause, it replaces every occurrence of the head of c in P with its body. For instance, unfolding the clause 3 with the clause 2 results in the clause 1. We assume that every inlined clause is removed from the program after unfolding.

Knowledge Refactoring Problem

Consider our running example of constructing arcade structures (Figure 1). The predicates in the logic program have different roles:

- *Primitive* predicates represent user-provided primitive knowledge that cannot be further decomposed, e.g. `place/4`, `right/2` and `left/2`.
- *Task* predicates define solutions to tasks we want to solve or have solved, e.g. arcade structures.
- *Support* predicates represent useful abstractions, e.g. `ver/3`. They help us better structure a program but can be unfolded from a program without changing its semantics with respect to the task predicates.

Our refactoring problem takes as input space of possible support predicates. We restrict support clauses by their size. The size of a clause c, $size$(c), is the number of literals in c (including the head atom). We define the support clause space:

Definition 1 (**Support clause space**). A clause c is in the support clause space $\mathcal{S}$ of a program P if (i) the head predicate of c does not appear in P, and (ii) the predicates in the body of c are in P or other support clauses.

When we refactor a program, we want to preserve the semantics of the original program with respect to complex predicates. To do so, we reason about the restricted consequences of a program:

Definition 2 (**Restricted consequences**). Let T be a set of predicate symbols and P be a logic program. The consequences of P restricted to T is $M_T(P) = \{a | a \in atoms(P), P \models a$, the predicate symbol of a is in $T\}$.

We also want to reduce the size of the original program. The function $size(P)$ denotes the total number of literals in the logic program P. We define the *knowledge refactoring* problem:

Definition 3 (**Knowledge refactoring**). Let P be a logic program, T be a set of task predicate symbols, and $\mathcal{S}$ be a space of support clauses. Then the refactoring problem is to find $P' \subseteq fold(unfold(P), \mathcal{S})$ such that (i) $M_T(P') = M_T(P)$, and (ii) $size(P') < size(P)$.

This definition provides conditions for refactoring: it should yield support clauses that, once folded into a program, (i) preserve the semantics of the original program, and (ii) lead to a smaller program. Refactoring, therefore, produces a lossless compression of the unfolded program, with respect to the task predicates. Importantly, it leaves the construction of the support clauses open as there are many valid ways to do so. We detail this aspect when discussing the implementation of the system.

Benefit of Refactoring

To show the potential benefits of refactoring, imagine an ILP system that enumerates all programs in the hypothesis space, a common approach when inducing functional programs (Balog et al. 2017; Ellis et al. 2018). Ignoring first-order variables for simplicity, the size of the hypothesis space is at most $\binom{p^l}{m}$ where p is the number of predicate symbols allowed in a hypothesis, l is the maximal number of literals in the body of a clause in a hypothesis, and m is the maximum number of clauses in a hypothesis. According to the Blumer bound (Blumer et al. 1987), given two hypothesis spaces of different sizes, and assuming that the target hypothesis is in both spaces, searching the smaller space will result in fewer mistakes compared to the larger one. This result implies that we can improve the performance of an ILP system by reducing either the number of predicate symbols p or the size of the target program n. By refactoring we can reduce (i) p by removing redundant predicate symbols and also by limiting the number of predicate symbols allowed in the BK, and (ii) m and l by restructuring the BK so that we can express the target hypothesis (program) using fewer, or shorter, clauses. We argue that refactoring is especially important in a *lifelong learning* setting where a system continuously learns thousands of new concepts, i.e. where p can be very large.

`Knorf`: A Knowledge Refactoring System

Syntactic Refactoring

The refactoring problem (Definition 3) requires that the refactored program is (in a restricted form) semantically equivalent to the original program. However, checking this requirement is intractable in practice because we need to check that two programs produce the same output for every possible input, which could be infinite. To make the problem tractable, `Knorf` uses a weaker criterion of *syntactic equivalence*:

Definition 4 (**Syntactical equivalence**). A program P is syntactically equivalent to the program P' if $unfold(P) = unfold(P')$.

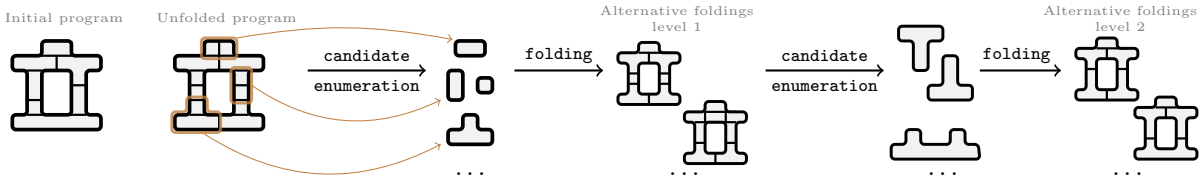


Figure 3: To identify candidate support clauses, Knorf first unfolds the original program. Then, Knorf constructs candidates from subparts of the unfolded program. To obtain more complex support clauses, Knorf first constructs alternative foldings of the given program, using the previously constructed candidates, and repeats the same procedure.

Note that two syntactically equivalent programs are necessarily semantically equivalent, while the opposite does not hold.

As Knorf searches for a syntactically equivalent refactoring, it constructs the support clause space by extracting subsets of body literals from the unfolded original program. We now introduce concepts necessary to construct the support clause space.

Definition 5 (Connected clause). A clause C is *connected* if it cannot be partitioned into two non-empty clauses C_1 and C_2 such that the variables in C_1 are disjoint from C_2, i.e, $\nexists C_1, C_2 \subseteq C$ such that $C_1 \neq C_2, \text{vars}(C_1) \cap \text{vars}(C_2) = \emptyset$.

For instance, the clause `h(X,Y) :- p(X,Y), q(Z)` is not connected because the variables can be partitioned in two disjoined subsets, {`X,Y`} and {`Z`}.

Definition 6 (Clausal power-set). A *clausal power-set* of the clause c, $\mathcal{P}(c)$, is a power-set of the literals in the body of c, excluding the empty set.

For instance, a clausal power-set of the clause `h(X,Y) :- a(X,Y),b(Y,Z),c(Z).` is {{`a(X,Y)`}, {`b(Y,Z)`}, {`c(Z)`},{`a(X,Y),b(Y,Z)`}, {`a(X,Y),c(Z)`}, {`b(Y,Z),c(Z)`}, {`a(X,Y),b(Y,Z),c(Z)`}}.

Definition 7. (Connected clausal power-set) A *connected clausal power-set* of the clause c, $\mathcal{C}(c)$, is a maximal subset of $\mathcal{P}(c)$ such at every $s \in \mathcal{C}(c)$ is connected: $\mathcal{C}(c) = \{e \in \mathcal{P}(c) \mid \text{connected}(c)\}$. In other words, only those subsets of literals of c that are connected.

Continuing on the previous example, the connected clausal power-set removes {`a(X,Y),c(Z)`} from the clausal power-set because the variables {`X,Y`} are disjoint from {`Z`}.

With these concepts in place, we define the space of support clauses.

Definition 8 (Space of support clauses). A clause is in the support clause space $\mathcal{S}^i_j$ of a program P when (i) it has at least i and at most j literals in the body, (ii) the set of literals in the body is in $\bigcup_{c \in P} \mathcal{C}(c)$ (up to variable renaming), and (iii) the head predicate symbol is unique and does not appear in P.

Knorf solves the refactoring problem by transforming it to a *constraint optimisation problem* (COP) (Rossi, van Beek, and Walsh 2006), where the goal is to find an optimal set of support clauses. Given (i) a set of *decision variables*, (ii) a problem description in terms of *constraints*, and (iii) an *objective function*, a COP solver finds an assignment to decision variables that satisfies all specified constraints and maximises or minimises the objective function[2].

Knorf minimises both size and redundancy:

Definition 9 (Syntactic redundancy). A logic program T has *syntactic redundancy* if there exist two clauses $c_1, c_2 \in T$ such that $c_1 \neq c_2$, $u_1 \in \mathcal{C}(c_1)$, $u_2 \in \mathcal{C}(c_2)$, $size(u_1) > 1$, $size(u_2) > 1$, and u_1 and u_2 are the same up to the variable renaming.

In other words, two clauses have a common subset of body literals. Though minimising program size should imply the removal of redundancy, we notice empirically that minimising both better guides the search towards good solutions, i.e. within a certain time limit.

Decision Variables: Support Clauses

Knorf solves the refactoring problem as a subset selection problem over the space of support clauses (Definition 8). In principle, the support clause space $\mathcal{S}^i_j$ is infinite, even with the upper-bounded length of clauses, as any number of support predicates can be introduced. To avoid this issue, we introduce an incremental procedure to construct $\mathcal{S}^i_j$ with clauses of fixed length. The procedure repeatedly applies two steps, **candidate extraction** and **folding**, starting from the unfolded program `P`. The unfolded program contains only *primitives* (in Figure 3 left, this results in clauses placing only ▭ and ▬ pieces).

The **candidate extraction** step constructs support clauses from the connected power-sets of the clauses from the unfolded program `P`, $\bigcup_{c \in \texttt{P}} \mathcal{C}(c)$, with at least i and at most j literals. More specifically, Knorf turns each element of $\bigcup_{c \in \texttt{P}} \mathcal{C}(c)$ into a support clause by creating a new predicate symbol in the head. These support clauses are expressed in terms of primitive predicates. In the Lego example, taking $i = 1$ and $j = 2$ would result in some of the candidates illustrated in Figure 3, middle.

The **folding** step folds the extracted support clauses into the (unfolded) program. This step essentially rewrites the program such that the bodies of its clauses are made of support predicates (a single clause can have multiple foldings). In the Lego example, this results in the simplified construction of the pillar structure (Figure 3, middle). To obtain more complex support clauses, Knorf repeats the same two steps but starts from the folded program.

[2]We use the CP-SAT solver (Perron and Furnon 2019).

Knorf repeats these two steps until each clause in the program has only one body literal. The result of this procedure is a **hierarchy** of support clauses, each one building on simpler support clauses. We refer to these steps as *levels of refactoring*; the folding the unfolded program yields *level one* refactoring, folding again yields *level two* refactoring, and so on.

Each enumerated support clause candidate k is associated with a Boolean variable $\mathtt{sc}_\mathtt{k}$ indicating whether the support clause is selected. Each folding of the clause i is associated with a Boolean variable $\mathtt{f}^\mathtt{i}_n$ indicating that a particular folding is selected as a part of the refactored program.

Pruning Support Clauses The incremental candidate enumeration procedure described above still results in many candidates because each clause can be expressed in many ways, given a set of support clauses. We further prune the support clause space by (i) eliminating clauses with singleton variables, and (ii) removing clauses that cannot reduce the program size.

We remove support clauses with *singleton variables*, i.e. clauses with a variable that only appears once. For instance, the clause:

```
4| sup(X,E) :-
     place(□,X,E,E1), place(□,X,E1,E').
```

is removed because $\mathtt{E}'$ appears only once. Adding $\mathtt{E}'$ as the last argument in the head would make the clause valid. As we focus on inductive program synthesis problems, ignoring singleton clauses is not sacrificing expressivity because singleton variables are essentially variables that are never used.

We also remove support clauses that cannot reduce the size of the program. For instance, let c be a support clause and $usage(\mathtt{c}, \mathcal{T})$ be the number of clauses in the program T which can be folded with c. This means that in the best case, we can replace $usage(\mathtt{c,\ T}) \times (size(\mathtt{c}) - 1)$ literals in the program (i.e., every occurrence of the body of c) by $usage(\mathtt{c,\ T}) \times 1$ uses of the head of the support clause c and the addition of a clause c to the theory T. Hence, if it is the case that this inequality holds:

$$usage(\mathtt{c,\ T}) \times (size(\mathtt{c}) - 1) \leq usage(\mathtt{c,\ T}) + size(\mathtt{c})$$

Then we know that the use of this candidate support clause will never lead to a reduction in the program size (our overall goal). We remove candidate support clauses that violate this inequality.

Constraints: Valid Refactoring

Each clause in the unfolded program has multiple possible foldings, grouped in different levels due to the support clause generation process. The refactored program should replace the original clauses with one of the possible foldings. Hence, Knorf enforces a constraint stating that at least one of the foldings of the clause i should be formed by the chosen support clauses.

We group the foldings of the clause i per level and add an additional level indicator $\mathtt{l}^\mathtt{i}_d$. For reasons that will become obvious later, the level indicator $\mathtt{l}^i_d$ is a Boolean variable which is set to true if the selected folding of the clause i comes from the level d. This results in constraints of the form:

$$\bigvee_{d=1}^{\text{max levels}} \left(\mathtt{l}^i_d \wedge \left(\bigvee_n \mathtt{f}^i_n \right) \right).$$

Knorf forces that one level of refactoring is chosen for each clause by imposing the following constraint:

$$\sum_{d=1}^{\text{max levels}} \mathtt{l}^i_d = 1.$$

This level variable will be part of the objective, where higher levels are typically better.

To decide whether a folding $\mathtt{f}^\mathtt{i}_n$ can be constructed, the solver needs to know which support clauses are needed for that particular folding. For instance, to construct the top folding at the level 1 in Figure 3, we need the following pieces: □, ▭, ⊓, and ⊓ (assume that the selection of these support clauses is indicate with the variables $\mathtt{sc}_p$, $\mathtt{sc}_r$ and $\mathtt{sc}_q$). To ensure this connection, Knorf enforces the constraint stating that the folding $\mathtt{f}^\mathtt{i}_n$ can be constructed only if all the necessary pieces are selected as a part of the solution:

$$\mathtt{f}^\mathtt{i}_n \Leftrightarrow (\mathtt{sc}_p \wedge \mathtt{sc}_r \wedge \mathtt{sc}_q).$$

Finally, candidates extracted from level L depend on the candidates from level $L-1$ (i.e., the bodies of support clauses from level L are composed from the predicates introduced by the support clauses at level $L-1$). Knorf imposes the constraint directly materialising this dependency – if the support clause $\mathtt{sc}_k$ is selected as a part of the solution, then all support clauses defining the predicates in the body of the clause $\mathtt{sc}_k$ (assume $\mathtt{sc}_l$ and $\mathtt{sc}_m$) also have to be a part of the solution

$$\mathtt{sc}_k \Rightarrow (\mathtt{sc}_l \wedge \mathtt{sc}_m).$$

For instance, one needs ▭ and □ bricks to make ⊓.

Objective: Size and Redundancy

Knorf searches for the smallest refactored program, syntactically equivalent to the original program, with the least redundancy. The size of the refactored program equals the number of literals in it, i.e., the size of the *selected* foldings and support clauses. To guide a COP solver towards a small refactored program, Knorf minimises the following objective function

$$\underbrace{\sum_{\mathtt{cl} \in \mathtt{T}} \sum_{d=1}^{\mathrm{L}} \sum_{n=1}^{F^d(\mathtt{cl})} \mathtt{w}_{\mathtt{f}^\mathtt{cl}_n} * \mathtt{f}^\mathtt{cl}_n * \mathtt{l}^\mathtt{cl}_d}_{\text{size of selected foldings}} + \underbrace{\sum_{\mathtt{sc} \in \mathcal{S}^i_j} \mathtt{w}_\mathtt{sc} * \mathtt{sc}}_{\text{size of selected support clauses}}$$

where L is the maximal number of level,$F^d(\mathtt{cl})$ is the number of foldings of the clause cl at level d, and $\mathtt{w}_\mathtt{cl}$ is the size of clause cl.

The level indicators introduced in Section are important to measure the program size correctly. Assume that the selected folding of a certain (unfolded) clause is at level 3. As any folding at level 3 is constructed from support predicates introduced at level 2, at least one folding at level 2 is possible to construct. But any folding from level 2, if the selected one is at level 3, should not contribute to the size of the refactored

program. The level indicators ensure that only the selected folding contributes to the program size by multiplying the size of folding from lower levels by 0.

To minimise the redundancy between clauses, `Knorf` keeps track of all foldings that share literals in the body. We then introduce a new Boolean variable (e.g., $\mathtt{r}_i$) indicating whether more than one folding (e.g., corresponding to variables $\mathtt{f}_n^{\mathtt{i}}$ and $\mathtt{f}_m^k$) with such redundancy can be constructed

$$\mathtt{r}_i \Leftrightarrow \left(\mathtt{f}_n^{\mathtt{i}} + \mathtt{f}_m^{\mathtt{k}} > 1\right).$$

`Knorf` introduces such constraint for all found redundancies and adds the sum over `r` variables to the objective function.

Experiments

We argue that an ILP system can learn better from refactored BK. Our experiments, therefore, aim to answer the question:

Q: Can an ILP system learn *better* with refactored BK?

By better, we ask whether it can solve more tasks, learn with higher predictive accuracies, or learn in less time. To answer this question, we compare the performance of state-of-the-art ILP system Metagol (Cropper and Muggleton 2016) with and without refactored BK.

Lifelong learning. To evaluate the usefulness of refactoring, we focus on a lifelong learning scenario in which a learner continuously learns to perform new tasks by continuously adding programs to its BK. This allows us to evaluate the benefit of refactoring over BKs with various sizes. To generate the BK in this setting, we use Playgol (Cropper)[3], an ILP system that generates BK automatically. Playgol learns in two phases. In the first **play** phase, Playgol solves randomly generated tasks that are similar to the user-provided target tasks. In the second **build** phase, Playgol solves the user-provided tasks, using the solutions to the play tasks as BK. We refer to the play tasks as *background tasks* and generate BKs with n background tasks, $n \in \{200, 400, \ldots, 4000\}$

Systems. We evaluate Metagol when learning to solve user-provided tasks from (i) the generated BK (**No refactoring**), (ii) the BK after refactoring, i.e. after `Knorf` has refactored it (**Refactoring**), and (iii) the BK refactored with a simple form of refactoring that replaces every redundancy in a program with a new predicate symbol and represents the redundancy with an additional clause (**No redundancy**).

Experiment setting. To build the support clause space, we set the minimum and maximum length of support clauses to 2 and 3 respectively. We impose no limit on the number of layers. When solving the COP, we impose a timeout of 90 minutes. If refactoring takes longer, we stop the search and take the best solution found so far. We additionally impose a constraint that the refactored BK cannot have more predicates than the original BK. We give Metagol a learning timeout of 60 seconds per task. We repeat each experiment 10 times and plot the means and 95% confidence intervals. All experiments are run on a CPU with 3.20 GHz and 16 Gb RAM. We have allowed CP-SAT so use 8 parallel threads.

[3]The original work performs simple deduplication of clauses. To fully verify the usefulness of refactoring, we have disabled this step.

Experiment 1 - Lego

Our first experiment is on learning to build Lego structures in a controlled environment (Cropper 2020).

Materials We consider a Lego world with a base dimension of 6×1 on which bricks can be stacked. We only consider 1×1 bricks of a single colour. A training example is an atom $f(s1, s2)$, where f is the target predicate and $s1$ and $s2$ are initial and final states respectively. A state describes a Lego structure as a list of integers. The value k at index i denotes that there are k bricks stacked at position i. The goal is to learn a program to build the Lego structure from a blank Lego board (a list of zeros). We generate training examples by generating random final states. The learner can move along the board using the actions *left* and *right*; can place a Lego brick using the action *place_brick*; and can use the fluents *at_left* and *at_right* and their negations to determine whether it is at the leftmost or rightmost board position.

Method The background tasks were generated with a Lego board of size 2 to 4. We randomly generate 1000 target tasks for a Lego board of size 6. We measure the percentage of tasks solved (tasks where the Metagol learns a program) and learning times (total time need to solve all target tasks).

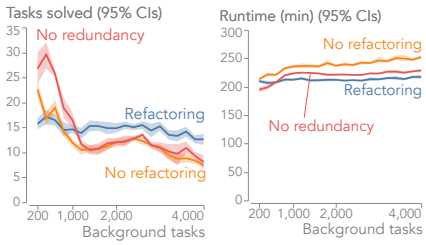
Results The results (Figure 4a) show that refactoring helps Metagol to maintain performance when confronted with large BK. With refactored BK, Metagol's performance decreases less with the increase of background tasks. The results also show that refactoring slightly degrades the ability to solve tasks when BK is small. The likely explanation is that smaller BK has less chance for redundancy and, thus, refactoring is eliminating predicates that Metagol could use to solve tasks. When the BK is large (≥ 1000 background tasks), refactoring improves the ability to solve tasks. These results appear to corroborate existing results (Cropper 2020) which show that simple forgetting can improve learning performance but only when learning from lots of BK. Figure 4b also shows that refactoring reduces learning times by approximately 20%. Interestingly, refactoring by replacing redundancies (No redundancy) consistently reduces total learning times, but does not improve performance for a large BK.

Figure 6a shows that refactoring drastically reduces the size of the BK. Both the number of literals and the number of predicates in the refactored BK are only a fraction of their number in the original BK. This suggests that much of the raw BK obtained by Playgol can be represented using a shared set of apriori unknown support clauses.

Experiment 2 - String Transformations

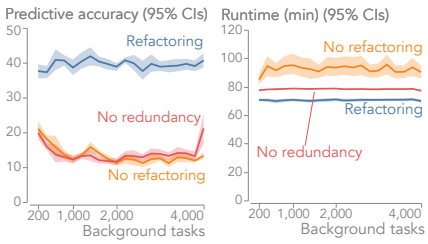
Our second experiment is on *real-world* string transformations.

Materials We use 130 string transformation tasks from (Cropper). Each task has 10 examples. An example is an atom $f(x, y)$ where f is the task name and x and y are input and output strings respectively. The goal is to learn to map the inputs to the outputs, such as to map the full name of a person (input) to its initials (output), e.g. *'Alan Turing'* $\mapsto$ *'A.T.'*. We provide as BK the binary predicates *mk_uppercase*,



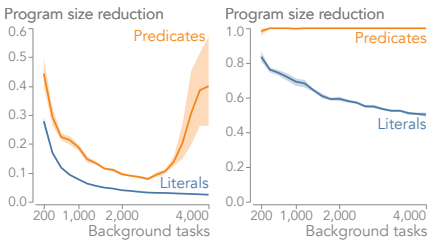
(a) (b)

Figure 4: With refactoring, Metagol solves more Lego tasks and does so in less than time than without refactoring.



(a) (b)

Figure 5: With refactoring, Metagol solves more string transformation tasks and does so in less than time than without refactoring.



(a) Lego (b) Strings

Figure 6: Refactoring reduces (shown as $^{\text{refactored}}/_{\text{original}}$) the number of predicates and literals in the program

mk_lowercase, *skip*, *copy*, *write*, and the unary predicates *is_letter*, *is_uppercase*, *is_space*, *is_number*.

Method We follow the procedure described in (Cropper) to obtain the play tasks and thus BK. For each of the 130 tasks, we sample uniformly without replacement 5 examples as training examples and use the remaining 5 as test examples. We measure predictive accuracy and learning times (total time needed to solve all target tasks).

Results The results (Figure 5a) show that refactoring drastically improves predictive accuracies. When learning from unrefactored BK and BK with redundancies removed, Metagol's performance quickly deteriorates because Metagol's search space increases exponentially in the size of the BK. By contrast, when given refactored BK, Metagol has higher predictive accuracy in all cases, eventually four times higher than without refactored BK. Moreover, the results show that refactoring reduces learning times by a third. Interestingly, the gain in performance does not come from the reduced number of predicates, as both refactored and unrefactored programs have an equal number of predicates, though overall program size decreases (Figure 6b). Rather, the performance gains (both in accuracy and speed) come from **better structured** knowledge.

Solver Behaviour Figure 7 shows the reduction of the BK size during a typical refactoring process for three different BK sizes. Regardless of the size, the solver can find a good solution (within 10% of the final size) within a minute. It takes between 2-21 min to reach a solution within 1% of the final size, depending on the number of background tasks. Though the solver finds the best solution within an hour, for most of the runs it continues searching for a better solution until timeout. This indicates two things: (1) we could have obtained equally good solutions with a more restrictive timeout, and (2) the encoding of a problem could be improved as the solver currently spends most of the time finding small improvements.

Conclusion

The main claim of this work is that the structure of an agent's knowledge can significantly influence its learning abilities: more knowledge results in larger hypothesis spaces

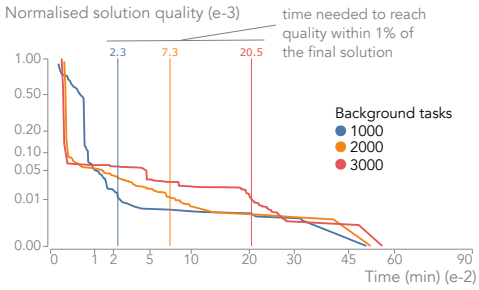


Figure 7: Despite the timeout of 90 min, the solver can quickly find a good solution and finds the best solution within an hour. Note that axes follow power scale.

and makes learning more difficult. Focusing on inductive logic programming, we introduced a problem of knowledge refactoring – rewriting an agent's knowledge base, expressed as a logic program, by removing the redundancies and minimising its size. We also introduced `Knorf`, a system that performs automatic knowledge refactoring by formulating it as a constraint optimisation procedure. We evaluated the proposed approach on two inductive program synthesis domains: building Lego structures and real-world string transformations. Our experimental results show that learning from the refactored knowledge base results can increase predictive accuracies fourfold and reduced learning times substantially.

Limitations and Future Work We have used one ILP system that already performs predicate invention. It would be interesting to see how effective refactoring is when a system does not perform predicate invention. We have focused on eliminating redundancy to improve the performance of an ILP system. However, there are many other properties that we may want to optimise, such as modularity or readability. Finally, we have not tackled the question of *when to refactor?* Refactoring too often is likely to have a negative effect on learning times. We will investigate the strategies for detecting the need for refactoring in future work.

Acknowledgments

S.D. is funded by the Research Foundation Flanders (FWO).

References

Adé, H.; Malfait, B.; and De Raedt, L. 1994. RUTH: an ILP theory revision system. In Raś, Z. W.; and Zemankova, M., eds., *Methodologies for Intelligent Systems*.

Balog, M.; Gaunt, A. L.; Brockschmidt, M.; Nowozin, S.; and Tarlow, D. 2017. DeepCoder: Learning to Write Programs. In *ICLR*. OpenReview.net.

Blumer, A.; Ehrenfeucht, A.; Haussler, D.; and Warmuth, M. K. 1987. Occam's Razor. *Inform. Proc. Lett.* 24: 377–380.

Cropper, A. ???? Playgol: Learning Programs Through Play. In *(IJCAI-19)*, 6074–6080.

Cropper, A. 2020. Forgetting to Learn Logic Programs. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020*, 3676–3683. AAAI Press.

Cropper, A.; and Muggleton, S. H. 2016. Metagol System. https://github.com/metagol/metagol. URL https://github.com/metagol/metagol. Accessed: October 2019.

Cropper, A.; and Tourret, S. 2020. Logical reduction of metarules. *Machine Learning* 109(7): 1323–1369.

De Raedt, L. 1992. *Interactive Theory Revision: An Inductive Logic Programming Approach*. Academic Press Ltd. ISBN 0122107306.

De Raedt, L.; Kersting, K.; Kimmig, A.; Revoredo, K.; and Toivonen, H. 2008. Compressing probabilistic Prolog programs. *Machine learning* 70(2): 151–168.

Dechter, E.; Malmaud, J.; Adams, R.; and Tenenbaum, J. 2013. Bootstrap Learning via Modular Concept Discovery. In *IJCAI*.

Dumančić, S.; Guns, T.; Meert, W.; and Blockeel, H. 2019. Learning relational representations with auto-encoding logic programs. In *IJCAI 2019*, 6081–6087.

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

Hemaspaandra, E.; and Schnoor, H. 2011. Minimization for Generalized Boolean Formulas. In Walsh, T., ed., *IJCAI 2011*, 566–571. IJCAI/AAAI.

Heule, M.; Järvisalo, M.; Lonsing, F.; Seidl, M.; and Biere, A. 2015. Clause Elimination for SAT and QSAT. *J. Artif. Intell. Res.* 53: 127–168.

Hocquette, C.; and Muggleton, S. H. 2020. Complete Bottom-Up Predicate Invention in Meta-Interpretive Learning. In Bessiere, C., ed., *IJCAI 2020*, 2312–2318. ijcai.org.

Karmiloff-Smith, A. 1992. *Beyond modularity : a developmental perspective on cognitive science / Annette Karmiloff-Smith*. MIT Press Cambridge, Mass. ISBN 0262111691 0262611147.

Lin, D.; Dechter, E.; Ellis, K.; Tenenbaum, J. B.; and Muggleton, S. 2014. Bias reformulation for one-shot function induction. In *ECAI 2014*, 525–530.

Mitchell, T. M. 1997. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill.

Muggleton, S. 1995. Inverse Entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming* 13(3-4): 245–286.

Muggleton, S.; and De Raedt, L. 1994. Inductive Logic Programming: Theory and Methods. *JOURNAL OF LOGIC PROGRAMMING* 19(20): 629–679.

Muggleton, S. H.; Lin, D.; and Tamaddoni-Nezhad, A. 2015. Meta-interpretive learning of higher-order dyadic Datalog: predicate invention revisited. *Machine Learning* 100(1): 49–73.

Perron, L.; and Furnon, V. 2019. OR-Tools. URL https://developers.google.com/optimization/. Accessed: October 2019.

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

Richards, B. L.; and Mooney, R. J. 1995. Automated Refinement of First-Order Horn-Clause Domain Theories. *Machine Learning* 19(2): 95–131.

Rossi, F.; van Beek, P.; and Walsh, T. 2006. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc.

Rumelhart, D. E.; Norman, D. A.; and California Univ., L. J. C. f. H. I. P. 1976. *Accretion, Tuning and Restructuring [microform] : Three Modes of Learning. Report No. 7602 / David E. Rumelhart and Donald A. Norman*. Distributed by ERIC Clearinghouse [Washington, D.C.]. URL https://eric.ed.gov/?id=ED134902.

Saitta, L.; and Zucker, J.-D. 2013. *Abstraction in artificial intelligence and complex systems*. Springer.

Shapiro, E. Y. 1982. *Algorithmic Program Debugging*. Ph.D. thesis, USA. AAI8221751.

Sommer, E. 1995. FENDER: An Approach to Theory Restructuring (Extended Abstract). In *ECML 1995*, volume 912 of *Lecture Notes in Computer Science*, 356–359. Springer.

Srinivasan, A.; King, R. D.; and Bain, M. 2003. An Empirical Study of the Use of Relevance Information in Inductive Logic Programming. *J. Mach. Learn. Res.* 4: 369–383.

Srinivasan, A.; Muggleton, S. H.; and King, R. D. 1995. Comparing the use of background knowledge by inductive logic programming systems. In *Proceedings of the 5th International Workshop on Inductive Logic Programming*, 199–230.

Stahl, I. 1993. Predicate Invention in ILP – an Overview. In *Proceedings of the 6th European Conference on Machine Learning*, ECML 1993, 311–322.

Sterling, L.; and Shapiro, E. 1986. *The Art of Prolog*. Cambridge, MA: MIT Press.

Stern, E. 2005. Knowledge restructuring as a powerful mechanism of cognitive development: How to lay an early foundation for conceptual understanding in formal domains. In Tomlinson, P.; Dockrell, J.; and Winne, P., eds., *Pedagogy - Learning for Teaching*, 155–170. British Psychological Society.

Tamaki, H.; and Sato, T. 1984. Unfold/Fold Transformation of Logic Programs. In *Proceedings of the Second International Logic Programming Conference, Uppsala University, Uppsala, Sweden, July 2-6, 1984*, 127–138. Uppsala University.

Wrobel, S. 1996. First-order theory refinement. In *Advances in Inductive Logic Programming*, 14–33.