Scalable Knowledge Refactoring Using Constrained Optimisation

Minghao Liu[1], **David M. Cerna**[2], **Filipe Gouveia**[1], **Andrew Cropper**[1]

[1]University of Oxford, United Kingdom
[2]Czech Academy of Sciences Institute of Computer Science (CAS ICS), Prague, Czechia
minghao.liu@cs.ox.ac.uk, dcerna@cs.cas.cz, filipe.gouveia@cs.ox.ac.uk, andrew.cropper@cs.ox.ac.uk

Abstract

Knowledge refactoring compresses logic programs by replacing them with new rules. Current approaches struggle to scale to large programs. To overcome this limitation, we introduce a constrained optimisation refactoring approach. Our first key idea is to encode the problem with decision variables based on literals rather than rules. Our second key idea is to focus on linear invented rules. Our empirical results on multiple domains show that our approach can refactor programs quicker and with more compression than the previous state-of-the-art approach, sometimes by 60%.

Code — https://github.com/minghao-liu/MaxRefactor

1 Introduction

Knowledge refactoring is a key component of human intelligence (Rumelhart and Norman 1976). The goal is to compress a knowledge base, such as a logic program, by introducing new rules and applying replacements. Knowledge refactoring is also important in AI, such as for policy reuse in planning (Bonet, Drexler, and Geffner 2024) and to improve performance in program synthesis (Ellis et al. 2018; Dumančić, Guns, and Cropper 2021).

To illustrate knowledge refactoring, consider the logic program:

$$\mathcal{P}_1 = \left\{ \begin{array}{l} g(A) \leftarrow p(A), q(A,B), r(B), s(A,B) \\ g(A) \leftarrow p(A), q(A,B), r(B), t(A,B) \\ g(A) \leftarrow p(B), q(B,C), r(C), w(A,B) \\ g(A) \leftarrow p(A), q(B,A), r(A), z(A,B) \end{array} \right\}$$

This program has 4 rules, each with 5 literals. Thus, the size of this program is 20 (literals).

We could add a new aux_1 rule to refactor $\mathcal{P}_1$ as:

$$\mathcal{P}_2 = \left\{ \begin{array}{l} aux_1(A,B) \leftarrow p(A), q(A,B), r(B) \\ g(A) \leftarrow aux_1(A,B), s(A,B) \\ g(A) \leftarrow aux_1(A,B), t(A,B) \\ g(A) \leftarrow aux_1(B,C), w(A,B) \\ g(A) \leftarrow p(A), q(B,A), r(A), z(A,B) \end{array} \right\}$$

The size of $\mathcal{P}_2$ is smaller than $\mathcal{P}_1$ (18 vs 20 literals) yet syntactically equivalent to $\mathcal{P}_1$ after unfolding[1] (Tamaki and Sato 1984).

A limitation of current refactoring approaches is scalability (Ellis et al. 2018; Cao et al. 2023; Bowers et al. 2023). For instance, KNORF (Dumančić, Guns, and Cropper 2021) frames the refactoring problem as a constrained optimisation problem (COP) (Rossi, Van Beek, and Walsh 2006). KNORF builds a set of invented rules by enumerating all subsets of the rules in an input program. KNORF then uses a COP solver to find a subset of the invented rules that maximally compresses the input program. However, because it enumerates all subsets, KNORF struggles to scale to programs with large rules and to programs with many rules.

To overcome this scalability limitation, we introduce a novel refactoring approach. Our first key contribution is a new COP formulation. Instead of enumerating all subsets of rules, we use decision variables to determine whether a literal is used in an invented rule. Our new formulation has two key advantages. First, the number of decision variables required is exponentially reduced. Second, an invented rule can use any combination of literals in the input program, rather than a strict subset of an input rule.

To illustrate the benefit of our first contribution, consider the input program $\mathcal{P}_1$. The invented rule aux_1 cannot refactor the last rule as any instance of aux_1 cannot cover the literals *p(A), q(B,A), r(A)* simultaneously. However, we could invent the rule aux_2 to refactor the program as:

$$\mathcal{P}_3 = \left\{ \begin{array}{l} aux_2(A,B,C) \leftarrow p(A), q(B,C), r(C) \\ g(A) \leftarrow aux_2(A,A,B), s(A,B) \\ g(A) \leftarrow aux_2(A,A,B), t(A,B) \\ g(A) \leftarrow aux_2(B,B,C), w(A,B) \\ g(A) \leftarrow aux_2(A,B,A), z(A,B) \end{array} \right\}$$

The body of aux_2 is not a subset of any rule in $\mathcal{P}_1$, so KNORF could not invent aux_2. By contrast, because we allow an invented rule to use any literal, we can invent aux_2. The program $\mathcal{P}_3$ is smaller than $\mathcal{P}_2$ (16 vs 18 literals) yet is syntactically equivalent to $\mathcal{P}_1$ after unfolding. As this example shows, our first contribution allows our approach to find better (smaller) refactorings.

[1]Unfolding means replacing all aux_1 literals in the body with its definition literals and eliminating aux_1 from the program. A formal description is in Section 3.

Our second key contribution is to use *linear invented rules* where (i) the body literals may not occur in the input program, and (ii) the size may be larger than any rule in the input program.

To illustrate this idea, consider the program:

$$\mathcal{Q}_1 = \mathcal{P}_1 \cup \left\{ \begin{array}{l} g(A) \leftarrow p(A), p(B), q(A,B), r(B) \\ g(A) \leftarrow p(A), q(A,B), q(B,C), r(C) \end{array} \right\}$$

Although we can use aux_1 and aux_2 to refactor $\mathcal{Q}_1$, we can find a smaller refactoring by introducing a *linear invented rule* aux_3 to refactor $\mathcal{Q}_1$ as:

$$\mathcal{Q}_2 = \left\{ \begin{array}{l} aux_3(A,B,C,D,E,F,G) \leftarrow p(A), p(B), q(C,D), \\ \qquad\qquad\qquad\qquad\quad q(E,F), r(G) \\ g(A) \leftarrow aux_3(A,A,A,B,A,B,B), s(A,B) \\ g(A) \leftarrow aux_3(A,A,A,B,A,B,B), t(A,B) \\ g(A) \leftarrow aux_3(B,B,B,C,B,C,C), w(A,B) \\ g(A) \leftarrow aux_3(A,A,B,A,B,A,A), z(A,B) \\ g(A) \leftarrow aux_3(A,B,A,B,A,B,B) \\ g(A) \leftarrow aux_3(A,A,A,B,B,C,C) \end{array} \right\}$$

Note that aux_3 contains body literals with different variables (e.g., *q(E,F)*, *r(G)*) and more literals than any input rule. This refactoring reduces the size from 30 ($\mathcal{Q}_1$) to 22 ($\mathcal{Q}_2$).

In a linear invented rule, the variables occur linearly in the body literals. We prove (Theorem 2) that if a program can be refactored using any invented rule then it can be refactored to the same size (or smaller) using a linear invented rule. This contribution allows our approach to find smaller refactorings.

1.1 Novelty and Contributions

The three main novelties of this paper are (i) the theoretical proof of the complexity of the optimal knowledge refactoring problem, (ii) a concise and efficient encoding of the problem as a COP, and (iii) demonstrating the effectiveness of our approach on large-scale and real-world benchmarks.

Overall, our contributions are:

- We introduce the *optimal knowledge refactoring* problem, where the goal is to compress a logic program by inventing rules. We prove that the problem is $\mathcal{NP}$-hard.
- We introduce MAXREFACTOR, which solves the optimal knowledge refactoring problem by formulating it as a COP.
- We evaluate our approach on multiple benchmarks. Our results show that, compared to the state-of-the-art approach, MAXREFACTOR can improve the compression rate by 60%. Our results also show that MAXREFACTOR scales well to large and real-world programs.

2 Related Work

Knowledge refactoring. Knowledge refactoring is important in many areas of AI (Bonet, Drexler, and Geffner 2024), notably in program synthesis (Ellis et al. 2018; Bowers et al. 2023; Cao et al. 2023; Hocquette, Dumancic, and Cropper 2024). For instance, Dumančić, Guns, and Cropper (2021) show that learning from refactored knowledge can substantially improve predictive accuracies of an inductive logic programming system and reduce learning times because the knowledge is structured in a more reusable way and redundant knowledge is removed.

Model reformulation. There is much research on reformulating constraint satisfaction problem (CSP) models automatically (O'Sullivan 2010; Charlier et al. 2017; Vo 2020). The main categories include implied constraints generation (Charnley, Colton, and Miguel 2006; Bessiere, Coletta, and Petit 2007), symmetry and dominance breaking (Liberti 2012; Mears and de la Banda 2015), pre-computation (Cadoli and Mancini 2006), and cross-modeling language translation (Drake et al. 2002). We differ because we work with logic programs and invent rules.

Redundancy and compression. Knowledge refactoring is distinct from knowledge redundancy, which is useful in many areas of AI, such as in Boolean Satisfiability (Heule et al. 2015). For instance, Plotkin (1971) introduced a method to remove logically redundant literals and clauses from a logical theory. Similarly, theory compression (De Raedt et al. 2008) approaches select a subset of clauses such that performance is minimally affected with respect to a cost function. We differ from redundancy elimination and theory compression because we restructure knowledge by inventing rules.

Predicate invention. We refactor a logic program by introducing predicate symbols that do not appear in the input program, which is known as predicate invention (Kramer 2020). Predicate invention is a major research topic in program synthesis and inductive logic programming (Kok and Domingos 2007; Muggleton, Lin, and Tamaddoni-Nezhad 2015; Hocquette and Muggleton 2020; Jain et al. 2021; Silver et al. 2023; Cerna and Cropper 2024). We contribute to this topic by developing an efficient and scalable method to invent predicate symbols to compress a logic program.

Program refactoring. In program synthesis, many researchers (Ellis et al. 2018; Bowers et al. 2023; Cao et al. 2023) refactor functional programs by searching for local changes (new λ-expressions) that optimise a cost function. We differ from these approaches because we (i) consider logic programs, (ii) use a declarative solving paradigm (COP), and (iii) guarantee optimal refactoring. ALPS (Dumančić et al. 2019) compresses facts in a logic program, whereas we compress rules.

KNORF. The most similar work is KNORF (Dumančić, Guns, and Cropper 2021). Given a logic program $\mathcal{P}$ as input, KNORF works as follows. For each rule $r \in \mathcal{P}$, KNORF enumerates every subset s of r. For each subset s and for each combination h of the variables in s, KNORF creates a new rule aux_{s_h}. KNORF then creates a COP problem where there is a decision variable for each aux_{s_h}. It also creates decision variables to state whether a rule $r \in \mathcal{P}$ is refactored using aux_{s_h}. KNORF then uses a COP solver to find a subset of the invented rules that leads to a refactoring with maximal compression. KNORF has many scalability issues. Foremost, it enumerates all subsets of all rules and thus struggles to scale to programs with large rules and many rules. Specifically, for a program with n rules and a maximum rule size k, KNORF uses $O(n2^k)$ decision variables. By contrast, our MAXREFACTOR approach does not enumerate all sub-

sets of rules. Instead, as we describe in Section 4, we define a new rule aux_i by creating decision variables to represent whether any literal in $\mathcal{P}$ is in aux_i. Besides, we create decision variables to state whether a rule and a literal in $\mathcal{P}$ is refactored using aux_i. Since the number of invented rules is a predefined constant, MAXREFACTOR only needs $O(nk)$ decision variables. Finally, we propose the refactoring using linear invented rules, which can expand the space for invented rules, enabling MAXREFACTOR to find better refactorings than KNORF.

3 Problem Setting

We focus on refactoring knowledge in the form of a logic program, specifically a definite logic program with the least Herbrand model semantics (Lloyd 2012). For simplicity, we use the term logic program to refer to a definite logic program. We assume familiarity with logic programming but restate some key terms. A logic program is a set of *rules* of the form:

$$h \leftarrow a_1, a_2, \ldots, a_m$$

where h is the *head literal* and $a_1, \ldots, a_m$ are the *body literals*. A *literal* is a predicate symbol with one or more variables. For example, *parent(A,B)* is a literal with the predicate symbol *parent* and two variables *A* and B. The predicate symbol of a literal a is denoted as $pred(a)$ and the set of its variables is denoted as $var(a)$. The head literal of a rule is true if and only if all the body literals are true. The head literal of a rule r is denoted as $head(r)$ and the set of body literals is denoted as $body(r)$. The *size* of a rule r is defined as $size(r) = |body(r)| + 1$, which is the total number of literals. The *size* of a logic program $\mathcal{P}$ is defined as $size(\mathcal{P}) = \sum_{r \in \mathcal{P}} size(r)$.

3.1 Knowledge Refactoring

Our goal is to reduce the size of a logic program whilst preserving its semantics. However, checking the semantic equivalence between two logic programs is undecidable (Shmueli 1987). Therefore, we focus on finding syntactically equivalent refactorings. Syntactic equivalence implies semantic equivalence but the reverse is not necessarily true.

We check syntactic equivalence by *unfolding* (Tamaki and Sato 1984) refactored programs. Unfolding is a transformation in logic programming. We adapt the unfolding definition of Nienhuys-Cheng and Wolf (1997) to (i) resolve multiple literals, whereas the original definition only resolves one literal each time, and (ii) prohibit variables that only appear in the body of a rule[2]:

Definition 1 (**Rule unfolding**). Given a logic program $\mathcal{P} = \{c_1, c_2, \ldots, c_N\}$ and a rule r, where r does not have variables that only appear in its body, then unfolding $\mathcal{P}$ upon r means constructing the logic program $\mathcal{Q} = \{d_1, d_2, \ldots, d_N\}$, where each d_i is the resolvent of c_i and r if $head(r)$ is unifiable with any literal in $body(c_i)$, otherwise $d_i = c_i$.

[2]Allowing variables that only appear in the body may result in unsound syntactic equivalence check as they can be arbitrarily renamed.

Example 1 (**Rule unfolding**). Consider the program:

$$\mathcal{P} = \left\{ \begin{array}{l} \textit{g(A)} \leftarrow \textit{p(A), aux(A,B)} \\ \textit{g(A)} \leftarrow \textit{p(B), p(C), aux(A,B), aux(A,C)} \\ \textit{g(A)} \leftarrow \textit{p(B), q(A,B), r(B)} \end{array} \right\}$$

and the rule r : *aux(A,B)* $\leftarrow$ *p(B), q(A,B)*. The result of unfolding $\mathcal{P}$ upon r is:

$$\mathcal{Q} = \left\{ \begin{array}{l} \textit{g(A)} \leftarrow \textit{p(A), p(B), q(A,B)} \\ \textit{g(A)} \leftarrow \textit{p(B), p(C), q(A,B), q(A,C)} \\ \textit{g(A)} \leftarrow \textit{p(B), q(A,B), r(B)} \end{array} \right\}$$

Given a set of rules S, $\mathsf{unfold}(\mathcal{P}, S)$ is the result of successively unfolding $\mathcal{P}$ upon every rule in S.

To refactor a logic program, we want to introduce invented rules:

Definition 2 (**Invented rule**). Given a logic program $\mathcal{P}$ and a finite set of variables $\mathcal{X}$, an invented rule r satisfies four conditions:

1. $pred(head(r)) \notin \mathcal{P}$
2. $\forall a \in body(r),\ pred(a) \in \mathcal{P}$
3. $\forall a \in body(r),\ var(a) \subseteq \mathcal{X}$
4. $var(head(r)) = \bigcup_{a \in body(r)} var(a)$

In this paper, we use the prefix *aux* to denote the predicate symbol of an invented rule.

We want to refactor an input rule using invented rules. We call such rules *refactored rules*:

Definition 3 (**Refactored rule**). Let r be a rule and S be a set of invented rules. Then r' is a refactored rule of r iff $\mathsf{unfold}(\{r'\}, S) = \{r\}$. For a logic program $\mathcal{P}$ and a set of invented rules S, $\mathcal{I}(\mathcal{P}, S)$ denotes the set of all possible refactored rules of rules in $\mathcal{P}$.

Example 2 (**Refactored rule**). Given the rules in $\mathcal{Q}$ and the invented rule *aux* in Example 1, the first two rules in $\mathcal{P}$ are refactored rules.

We define a *refactored program*:

Definition 4 (**Refactored program**). Given a logic program $\mathcal{P}$ and a finite set of invented rules S, a logic program $\mathcal{Q} \subseteq \mathcal{P} \cup S \cup \mathcal{I}(\mathcal{P}, S)$ is a refactored program of $\mathcal{P}$ iff $\mathsf{unfold}(\mathcal{Q} \setminus S, \mathcal{Q} \cap S) = \mathcal{P}$, and we refer to $\mathcal{Q}$ as a *proper refactoring* if $\forall S' \subset \mathcal{Q} \cap S$, $\mathsf{unfold}(\mathcal{Q} \setminus S, \mathcal{Q} \cap S') \neq \mathcal{P}$.

To help explain the intuition behind Definition 4, note the following three properties of a refactored program. First, a refactored program consists of three kinds of rules (a) input rules from $\mathcal{P}$, (b) invented rules from S, and (c) refactored rules from $\mathcal{I}(\mathcal{P}, S)$. Second, $\mathcal{P}$ and $\mathcal{Q}$ must be syntactically equivalent, which means that we can unfold $\mathcal{Q}$ upon the invented rules to get $\mathcal{P}$. Third, we are only interested in programs $\mathcal{Q}$ which are proper refactorings of $\mathcal{P}$ as any non-proper refactoring containing $\mathcal{Q}$ is always a larger program.

We want *optimal knowledge refactoring (OKR)*:

Definition 5 (**Optimal knowledge refactoring problem**). Given a logic program $\mathcal{P}$ and a finite set of invented rules S, the optimal knowledge refactoring problem is to find a refactored program $\mathcal{Q}$ such that for any refactored program $\mathcal{Q}'$, $size(\mathcal{Q}) \leq size(\mathcal{Q}')$.

We prove that the OKR problem is $\mathcal{NP}$-hard:

Theorem 1. The optimal knowledge refactoring problem is $\mathcal{NP}$-hard.

Proof (Sketch). The full proof is in Appendix A. We provide a reduction from *maximum independent set in 3-regular Hamiltonian graphs* (Fleischner, Sabidussi, and Sarvanov 2010) to a propositional variant of OKR where $c_1, c_2 \in S$ may refactor the same rule of $\mathcal{P}$ iff $body(c_1) \cap body(c_2) = \emptyset$ (OPKR$_{WD}$). Let $G = (V, E)$ be a 3-regular Hamiltonian graph and C a maximum independent set of G. Observe that in $G' = (V \setminus C, E')$, where E' contains all edges from E with endpoints in $V \setminus C$, every vertex has degree at most 2. Thus, computing a maximum weighted independent set of G' is possible in polynomial time. Furthermore, it can be shown that optimal refactorings of instances derived from a 3-regular Hamiltonian graph G are always a superset of a maximum independent set of G. Thus, we can reduce OPKR$_{WD}$ to OPKR (*propositional* OKR), a fragment of OKR. □

4 MAXREFACTOR

Given an input logic program $\mathcal{P}$, a space of possible invented rules S, and a maximum number K of invented rules to consider, MAXREFACTOR refactors $\mathcal{P}$ by finding $C \subseteq S$ where $|C| = K$. In other words, MAXREFACTOR solves the optimal knowledge refactoring problem.

Before describing how we search for a refactoring, we first introduce *linear invented rules* to restrict the space of possible invented rules.

4.1 Linear Invented Rule

The space of possible invented rules S directly influences the effectiveness and efficiency of refactoring. For example, KNORF defines S as all subsets of rules in an input program. For instance, if there is an input rule *g(A) ← p(A,B), q(B,C), r(B,D)*, KNORF adds four possible invented rules to S:

$$\left\{ \begin{array}{l} aux_1(A,B,C) \leftarrow p(A,B),\ q(B,C) \\ aux_2(A,B,D) \leftarrow p(A,B),\ r(B,D) \\ aux_3(B,C,D) \leftarrow q(B,C),\ r(B,D) \\ aux_4(A,B,C,D) \leftarrow p(A,B),\ q(B,C),\ r(B,D) \end{array} \right\}$$

Defining S this way has the advantage that it restricts which literals can appear in an invented rule. However, this approach also has disadvantages. As we show in Section 1, this approach cannot invent a rule that is not a subset of any input rule (e.g. aux_2 in $\mathcal{P}_3$), nor can the size of the invented rule be larger than any input rule (e.g., aux_3 in $\mathcal{Q}_2$).

To overcome these limitations, we use a new approach to define the space of invented rules. We denote the set of body predicate symbols in the logic program $\mathcal{P}$ as $Pr(\mathcal{P})$ and a finite set of variables as $\mathcal{X}$. We define the set of all possible body literals $L = \left\{p(v) \mid p \in Pr(\mathcal{P}), v \in \mathcal{X}^{arity(p)}\right\}$. We define S as the space of all combinations of literals from L, i.e., the power set of L. The size of S is exponential in the size of L and S is clearly a superset of the space considered by KNORF. Therefore, our key insight is not to consider all combinations of L and to instead only consider the *linear invented rules*:

Definition 6 (Linear invented rule). A linear invented rule is rule where no variable occurs more than once in its body.

Example 3 (Linear invented rule). The rule *aux$_1$(A,B,C) ← p(A,B), q(B,C)* is not linear because its body literals share the variable *B*. By contrast, the rule *aux$_2$(A,B,C,D) ← p(A,B), q(C,D)* is linear.

For any invented rule r, we can always build a linear invented rule $lin(r)$ as follows: (i) rewrite the body literals such that the predicate symbols remain unchanged and the variables occur linearly, then (ii) build a new head literal such that $var(head(lin(r))) = \bigcup_{a \in body(lin(r))} var(a)$.

To motivate the use of linear invented rules in refactoring, we show the theorem:

Theorem 2. If the optimal knowledge refactoring problem has a solution using the set of invented rules $C \subseteq S$ then it has a solution using $C' \subseteq S_{lin}$, where $S_{lin} = \{lin(s) \mid s \in S\}$.

Proof. Suppose there is an invented rule $r \in C$, we can always build a linear invented rule $r' = lin(r)$, then $C' = C \setminus \{r\} \cup \{r'\}$. First, observe that $|C'| \leq |C|$ because either an invented rule has a unique linear invented rule generalising it, or multiple invented rules of C are generalised by the same linear invented rule. Second, for any body literal in the input program that is covered by r, it can also be covered by r' via variable mapping. Thus, we can always get a refactored program of the same or smaller size using C'. □

Theorem 2 implies that we can restrict S to linear invented rules without compromising the optimal solution. As we show below, defining S as linear invented rules allows us to simplify the COP formulation.

4.2 COP Encoding

According to Theorem 1, the optimal knowledge refactoring problem is $\mathcal{NP}$-hard, making it computationally challenging. Constraint programming (CP) is a successful framework for modeling and solving hard combinatorial problems (Hebrard 2018). Therefore, MAXREFACTOR formulates this search problem as a constrained optimisation problem (COP) (Rossi, Van Beek, and Walsh 2006). Given (i) a set of *decision variables*, (ii) a set of *constraints*, and (iii) an *objective function*, a COP solver finds an assignment to the decision variables that satisfies all the specified constraints and minimises the objective function.

We describe our COP encoding below[3].

Decision Variables MAXREFACTOR builds decision variables to determine (a) which literals are in which invented rules, and (b) how to refactor input rules using the invented rules.

For task (a), for each possible invented rule r_k for $k \in [1, K]$ and $p \in Pr(\mathcal{P})$, we use an integer variable $r_{k,p}$ to

[3]COP problems can also be encoded as MaxSAT problems (Bacchus, Järvisalo, and Martins 2021), an optimisation variant of SAT. We also develop a MaxSAT encoding and include it in the appendix.

indicate the number of literals with predicate symbol p in the body of r_k.

For example, consider the input program:

$$\mathcal{P} = \left\{ \begin{array}{l} c_1 : g(A) \leftarrow p(B)^{a1}, p(C)^{a2}, q(A,B)^{a3}, q(B,C)^{a4} \\ c_2 : g(A) \leftarrow p(B)^{a1}, q(A,B)^{a2}, q(A,C)^{a3}, s(C)^{a4} \end{array} \right\}$$

where the superscript denotes the index of a body literal. For simplicity, assume that $K = 1$, i.e. we can only invent one rule denoted r_1. The decision variables for inventing this rule are $\{r_{1,p}, r_{1,q}, r_{1,s}\}$. Suppose $r_{1,p} = r_{1,q} = 1$ and $r_{1,s} = 0$. Then $r_1 : aux_1(A,B,C) \leftarrow p(A), q(B,C)$.

For task (b), MAXREFACTOR creates two groups of decision variables: (i) $use_{c,k}$ to denote whether an input rule c is refactored with the invented rule r_k, and (ii) $cover_{c,a,k}$ to denote whether a body literal a in an input rule c is covered by an invented rule r_k, which means this literal is not in the refactoring of c.

Concerning (i), for each input rule $c \in \mathcal{P}$ and $k \in [1, K]$ we use a Boolean variable $use_{c,k}$ to indicate that c is refactored using the invented rule r_k. However, as we show later in the example, an input rule can be refactored using the same invented rule multiple times. Therefore, we use the Boolean variable $use^t_{c,k}$ where $t \geq 1$ to indicate that r_k is used t times[4] in the refactoring of c.

Concerning (ii), for each input rule $c \in \mathcal{P}$, literal $a \in body(c)$, $k \in [1, K]$, and integer $t \geq 1$, we use a Boolean variable $cover^t_{c,a,k}$ to indicate that a is covered by the t-th instance of r_k in the refactoring.

Regarding the above example, the decision variables used to determine the refactoring of each input rule are:

$$use^1_{c_1,1} \quad use^2_{c_1,1} \quad use^1_{c_2,1} \quad use^2_{c_2,1}$$

$$\begin{array}{llll} cover^1_{c_1,a1,1} & cover^2_{c_1,a1,1} & cover^1_{c_1,a2,1} & cover^2_{c_1,a2,1} \\ cover^1_{c_1,a3,1} & cover^2_{c_1,a3,1} & cover^1_{c_1,a4,1} & cover^2_{c_1,a4,1} \\ cover^1_{c_2,a1,1} & cover^2_{c_2,a1,1} & cover^1_{c_2,a2,1} & cover^2_{c_2,a2,1} \\ cover^1_{c_2,a3,1} & cover^2_{c_2,a3,1} & cover^1_{c_2,a4,1} & cover^2_{c_2,a4,1} \end{array}$$

If the gray decision variables are false and the rest are true then the refactored rules are:

$$\mathcal{Q} = \left\{ \begin{array}{l} g(A) \leftarrow aux_1(B,A,B), aux_1(C,B,C) \\ g(A) \leftarrow aux_1(B,A,B), q(A,C), s(C) \end{array} \right\}$$

The refactored rules are constructed through the following steps: (i) If a *use* variable is true for an invention I and an input rule R, then create a body literal P for the refactored rule of R where all the variables of P are unknown. (ii) For every body literal L covered by I (denoted by the *cover* variable), bind the variables of L to the first unknown variables of P corresponding to the same predicate symbol. (iii) If there are still unknown variables in P, bind the variables of any literals covered by I with the same predicate symbols to them.

[4]The domain of t is defined by the maximum number of times a predicate symbol appears in an input rule.

Constraints MAXREFACTOR ensures the validity of the refactoring through a set of constraints.

We use Constraint (1) to ensure that for $k \in [1, K]$, if the invented rule r_k is used t times in the refactoring of $c \in \mathcal{P}$ then r_k is used $t-1$ times in the refactoring of c.

$$\forall c, k, t\,(t > 1),\ use^t_{c,k} \rightarrow use^{t-1}_{c,k} \tag{1}$$

We use Constraint (2) to ensure that for $k \in [1, K]$, if r_k is used t times to refactor $c \in \mathcal{P}$, then r_k cannot contain a predicate symbol p which is not the predicate symbol of at least one literal in $body(c)$.

$$\forall c, p, k, t\,(p \text{ not in } c),\ \neg\left(use^t_{c,k} \wedge r_{k,p} > 0\right) \tag{2}$$

We use Constraint (3) to ensure that for $k \in [1, K]$, that any predicate symbol p of a literal a in the body of $c \in \mathcal{P}$ covered by invented rule r_k, is the predicate symbol of a literal in the body of r_k. Below, $e(a, p)$ abbreviates $pred(a) = p$.

$$\forall c, a, k, t\,(e(a,p))\ cover^t_{c,a,k} \rightarrow \left(use^t_{c,k} \wedge r_{k,p} > 0\right) \tag{3}$$

We use Constraint (4) to ensure that for $k \in [1, K]$ and $c \in \mathcal{P}$, if the predicate symbol p occurs in multiple body literals of c and $r_{k,p}$ times in r_k then a single use of r_k can only cover $r_{k,p}$ of the occurrences of p in c.

$$\forall c, p, k, t,\ \left(\sum_{a \in body(c) \wedge e(a,p)} cover^t_{c,a,k}\right) \leq r_{k,p} \tag{4}$$

Note that if $r_{k,p}$ is larger than the number of body literals with predicate symbol p in a rule, it would still be valid, because these literals can be refactored by duplication. For example, given the input rule $g(A) \rightarrow p(A,B)$ and the invented rule $aux(A,B,C,D) \rightarrow p(A,B),p(C,D)$, we can refactor the same literal twice and get $g(A) \rightarrow aux(A,B,A,B)$.

Objective Before introducing our objective function, we first create two useful yet redundant decision variables. First, for each $k \in [1, K]$, we use a Boolean variable $used_k$ to indicate whether the invented rule r_k is used in the refactoring. Second, for each input rule $c \in \mathcal{P}$ and literal $a \in body(c)$, we use a Boolean variable $covered_{c,a}$ to indicate that a is not in the refactoring of c. These variables are directly calculated from use and $cover$ respectively. The constraints are as follows:

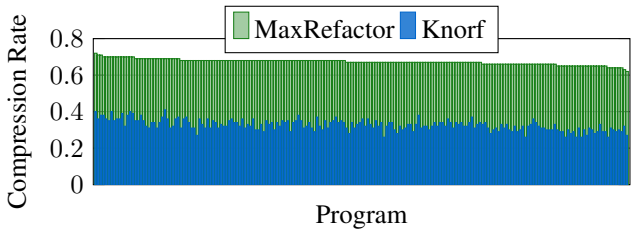
$$\forall k,\ \left(used_k \leftrightarrow \exists c, t,\ use^t_{c,k}\right) \tag{5}$$

$$\forall c, a,\ \left(covered_{c,a} \leftrightarrow \exists k, t,\ cover^t_{c,a,k}\right) \tag{6}$$

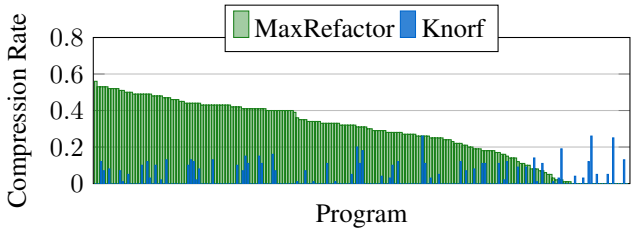
Following Definition 5, MAXREFACTOR searches for the refactored program with the smallest size. In our encoding, MAXREFACTOR achieves this goal by minimising the following objective function using a COP solver:

$$\sum_k used_k + \sum_{k,p} r_{k,p} + \sum_{c,k,t} use^t_{c,k} - \sum_{c,a} covered_{c,a} \tag{7}$$

The first two terms are the size of the head and body literals used to define the invented rules added to the refactored program, respectively. The third term is the size of the invented rules in the refactored rules of the input program. The last term is the total number of covered body literals in the input program, which should be deducted from the size. Note that by adding the objective to the size of the input program, we get the size of the refactored program.

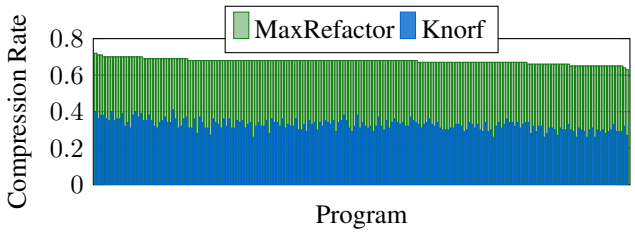


(a) Lego

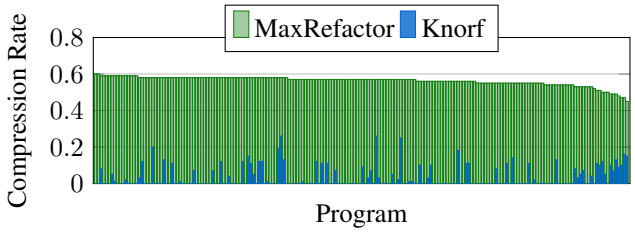


(b) Strings

Figure 1: The compression rates of MAXREFACTOR and KNORF (both calling CP-SAT) with a 600s timeout. There are two bars for each horizontal coordinate, representing the results of MAXREFACTOR (green) and KNORF (blue) on the same program.



(a) Lego



(b) Strings

Figure 2: The compression rates of MAXREFACTOR (calling MaxSAT) and KNORF (calling CP-SAT) with a 600s timeout. There are two bars for each horizontal coordinate, representing the results of MAXREFACTOR (green) and KNORF (blue) on the same program.

4.3 The Optimal Refactoring

We show that by solving the aforementioned COP problem, MAXREFACTOR solves the OKR problem (Definition 5):

Theorem 3. Let $\mathcal{P}$ be a logic program such that $|\mathcal{P}| = n$ and S a set of invented rules. Then MAXREFACTOR can solve the OKR problem with $K = \lceil \frac{n}{4} \rceil \lceil \frac{s-1}{2} \rceil$, where $s = \max\{size(c) \mid c \in \mathcal{P}\}$.

The proof is shown in Appendix B.

5 Experiments

To test our claim that MAXREFACTOR can find better refactorings than current approaches, our experiments aim to answer the question:

Q1: How does MAXREFACTOR compare against KNORF on the optimal knowledge refactoring problem?

To answer **Q1**, we compare the compression rate of the refactored programs produced by MAXREFACTOR and KNORF given the same timeout.

To understand the scalability of MAXREFACTOR, our experiments aim to answer the question:

Q2: How does MAXREFACTOR scale on larger and more diverse input programs?

To answer **Q2**, we evaluate the runtime and refactoring rate of MAXREFACTOR on progressively larger programs.

Settings We use two versions of MAXREFACTOR: one which uses the COP solver CP-SAT[5] and another which uses the MaxSAT solver UWrMaxSat (Piotrów 2020). We consider at most two invented rules in a refactoring, as we found it leads to good results. For KNORF, we run the code from its repository[6] and use the same parameters as stated in that paper. All experiments use a computer with 5.2GHz CPUs and 16GB RAM. We run MAXREFACTOR and KNORF on a single CPU.

5.1 Q1: Comparison with KNORF

Datasets We use two datasets, Lego and Strings, from the KNORF paper. Lego has 200 programs, each with a size of 264–411. Strings has 196 programs, each with a size of 779–12,309. We ran KNORF on other datasets but it frequently crashed on them[7] so we only report the results on these two datasets.

Method We compare the compression rate of the refactored programs returned by MAXREFACTOR and KNORF. Given an input program $\mathcal{P}$ and a refactored program $\mathcal{Q}$, the compression rate cr is defined as:

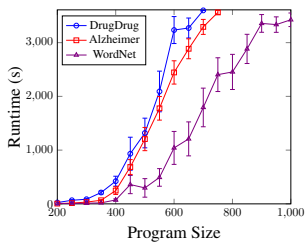
$$cr = \frac{size(\mathcal{P}) - size(\mathcal{Q})}{size(\mathcal{P})}$$

To be clear, a larger cr indicates better refactoring.

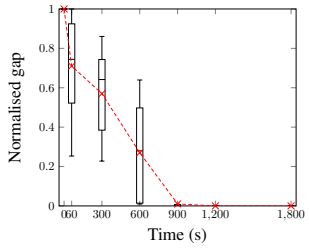
[5]https://developers.google.com/optimization

[6]https://github.com/sebdumancic/knorf_aaai21

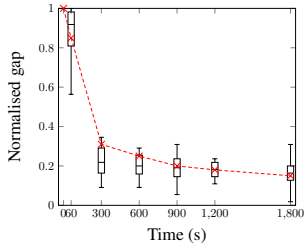
[7]KNORF crashes on other datasets during its COP modeling phase. We contacted the authors of KNORF for advice and they confirmed that it is a fundamental bug related to incorrect assumptions on the input program. We tried to fix the bug according to their advice but it fixed only a small number of crashes and the runtime significantly increased. Therefore, we only report the results on Lego and Strings from the KNORF paper.



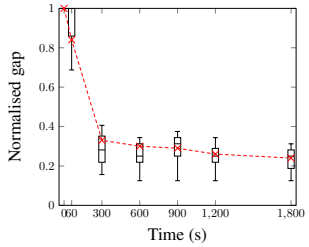
(a) Average runtime of optimal refactoring versus program size. The bars are standard errors.



(b) Normalised solution quality versus the runtime (a Strings task, size=11,947, opt-time=23min).



(c) Normalised solution quality versus the runtime (a DrugDrug task, size=700, opt-time=2,440min).



(d) Normalised solution quality versus the runtime (an Alzheimer task, size=700, opt-time=1,685min).

Figure 3: Scalability results. In (b) to (d), the data points on the red dashed line denote the average normalised gap of 40 independent runs and the boxes show the distribution of these results.

Results Figure 1 compares the compression rates of MAXREFACTOR and KNORF with the same timeout. Both approaches call CP-SAT. For Lego, MAXREFACTOR can increase compression rates by at least 27% across all programs. For Strings, MAXREFACTOR finds better refactorings on 172 out of 196 programs, increasing the compression rates by an average of 30%. In the remaining 24 programs, both MAXREFACTOR and KNORF fail to find any refactoring in half of them, while in the other half MAXREFACTOR's compression rates are on average 10% lower than KNORF's.

We also use a version of MAXREFACTOR that uses a MaxSAT solver (See Appendix C for encoding details). Figure 2 shows the results when compared to KNORF. Given the same timeout (600s), MAXREFACTOR with MaxSAT achieves better refactoring effect compared to CP-SAT. For Strings, MAXREFACTOR finds better refactorings than KNORF on all the 196 programs, increasing the compression rates by 30%–60% with an average of 53%.

In conclusion, these results suggest that MAXREFACTOR performs substantially better than KNORF at solving the knowledge refactoring problem, thereby answering **Q1**.

5.2 Q2: Scalability

Datasets. We evaluate the scalability of MAXREFACTOR on three real-world problems: DrugDrug (Dhami et al. 2018), Alzheimer (King, Sternberg, and Srinivasan 1995), and WordNet (Bordes et al. 2014). Each dataset contains millions of rules so we can randomly sample programs of distinct sizes from them and use the sampled programs to test the performance of MAXREFACTOR. Specifically, for every size in $\{200, 250, 300, \ldots, 1000\}$, we uniformly sample 10 programs from each dataset and run MAXREFACTOR (calling MaxSAT) to refactor them.

Results. Figure 3a shows the runtime required for MAXREFACTOR to find an optimal solution and prove optimality for different sizes of input programs. The timeout for a single run is set to 3600s and the runtime is counted as 3600s if the limit is exceeded. For a program size of 400, the average runtime on all the datasets is less than 500s. The performance varies across different datasets. For DrugDrug and Alzheimer, MAXREFACTOR can scale to programs with a size of 600 and find the optimal refactorings. For WordNet, it can scale to programs with a size of 1000.

Figure 3a shows the runtime required for MAXREFACTOR to solve the optimal solution for different sizes of input programs, i.e. to find an optimal refactoring and, crucially, prove optimality. However, in most cases, MAXREFACTOR can find near-optimal refactorings quickly but takes a while to prove optimality. To illustrate this behaviour, we select 3 challenging programs from different datasets and measure the sizes of the best refactored programs found by MAXREFACTOR under different time limits. Compared to the runtime to prove the optimal refactorings (opt-time), MAXREFACTOR can find refactored programs within 20% of the final size with only 44% (Figure 3b), 0.2% (Figure 3c), and 0.3% (Figure 3d) of the runtime respectively. For example, in the DrugDrug task (Figure 3c), MAXREFACTOR finds a refactoring of size 630 within 300s, while it takes 2,440 minutes to prove that the optimal refactoring size is 623. These results indicate that MAXREFACTOR in practice quickly finds a near-optimal refactoring but takes time to prove optimality.

In conclusion, these results suggest that MAXREFACTOR scales well on large and diverse programs from real-world problems, thereby answering **Q2**.

6 Conclusion and Limitations

We have introduced a novel approach to solve the optimal knowledge refactoring problem, which refactors a logic program to reduce its size. We implemented our ideas in MAXREFACTOR, which formulates this problem as a COP. Our experiments show that MAXREFACTOR drastically reduces the sizes of refactored programs compared to the state-of-the-art approach. Our results also show that MAXREFACTOR exhibits scalability on different programs from real-world problems.

Limitations We do not support recursive invented rules or including other invented predicates in an invented rule. Doing so may lead to better refactoring. Future work should address this limitation.

Acknowledgments

Minghao Liu, Filipe Gouveia, and Andrew Cropper 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. Additionally, we would like to thank Eng Keat Hng for pointing us towards the necessary graph theory literature.

References

Bacchus, F.; Järvisalo, M.; and Martins, R. 2021. Maximum Satisfiabiliy. In *Handbook of Satisfiability - Second Edition*, 929–991. IOS Press.

Bessiere, C.; Coletta, R.; and Petit, T. 2007. Learning Implied Global Constraints. In *IJCAI 2007*, 44–49.

Bonet, B.; Drexler, D.; and Geffner, H. 2024. On Policy Reuse: An Expressive Language for Representing and Executing General Policies that Call Other Policies. In *ICAPS 2024*, 31–39.

Bordes, A.; Glorot, X.; Weston, J.; and Bengio, Y. 2014. A Semantic Matching Energy Function for Learning with Multi-relational Data - Application to Word-sense Disambiguation. *Mach. Learn.*, 94(2): 233–259.

Bowers, M.; Olausson, T. X.; Wong, L.; Grand, G.; Tenenbaum, J. B.; Ellis, K.; and Solar-Lezama, A. 2023. Top-Down Synthesis for Library Learning. In *POPL 2023*, 1182–1213.

Cadoli, M.; and Mancini, T. 2006. Automated Reformulation of Specifications by Safe Delay of Constraints. *Artif. Intell.*, 170(8-9): 779–801.

Cao, D.; Kunkel, R.; Nandi, C.; Willsey, M.; Tatlock, Z.; and Polikarpova, N. 2023. Babble: Learning Better Abstractions with E-graphs and Anti-unification. In *POPL 2023*, 396–424.

Cerna, D. M.; and Cropper, A. 2024. Generalisation through Negation and Predicate Invention. In *AAAI 2024*, 10467–10475.

Charlier, B. L.; Khong, M. T.; Lecoutre, C.; and Deville, Y. 2017. Automatic Synthesis of Smart Table Constraints by Abstraction of Table Constraints. In *IJCAI 2017*, 681–687.

Charnley, J.; Colton, S.; and Miguel, I. 2006. Automatic Generation of Implied Constraints. In *ECAI 2006*, 73–77.

De Raedt, L.; Kersting, K.; Kimmig, A.; Revoredo, K.; and Toivonen, H. 2008. Compressing Probabilistic Prolog Programs. *Mach. Learn.*, 70(2): 151–168.

Dhami, D. S.; Kunapuli, G.; Das, M.; Page, D.; and Natarajan, S. 2018. Drug-drug Interaction Discovery: Kernel Learning from Heterogeneous Similarities. *Smart Health*, 9: 88–100.

Drake, L.; Frisch, A.; Gent, I.; and Walsh, T. 2002. Automatically Reformulating SAT-encoded CSPs. In *Workshop on Reformulating Constraint Satisfaction Problems*.

Dumančić, S.; Guns, T.; and Cropper, A. 2021. Knowledge Refactoring for Inductive Program Synthesis. In *AAAI 2021*, 7271–7278.

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*, 7816–7826.

Fleischner, H.; Sabidussi, G.; and Sarvanov, V. I. 2010. Maximum Independent Sets in 3- and 4-regular Hamiltonian Graphs. *Discret. Math.*, 310(20): 2742–2749.

Hebrard, E. 2018. Reasoning about NP-complete Constraints. In *IJCAI 2018*, 5672–5676.

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.; Dumancic, S.; and Cropper, A. 2024. Learning Logic Programs by Discovering Higher-Order Abstractions. In *IJCAI-24*, 3421–3429.

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

Jain, A.; Gautrais, C.; Kimmig, A.; and Raedt, L. D. 2021. Learning CNF Theories Using MDL and Predicate Invention. In *IJCAI 2021*, 2599–2605.

King, R. D.; Sternberg, M. J. E.; and Srinivasan, A. 1995. Relating Chemical Activity to Structure: An Examination of ILP Successes. *New Gener. Comput.*, 13(3&4): 411–433.

Kok, S.; and Domingos, P. M. 2007. Statistical Predicate Invention. In *ICML 2007*, 433–440.

Kramer, S. 2020. A Brief History of Learning Symbolic Higher-Level Representations from Data (And a Curious Look Forward). In *IJCAI 2020*, 4868–4876.

Liberti, L. 2012. Reformulations in Mathematical Programming: Automatic Symmetry Detection and Exploitation. *Math. Program.*, 131(1-2): 273–304.

Lloyd, J. W. 2012. *Foundations of Logic Programming*. Springer Science & Business Media.

Mears, C.; and de la Banda, M. G. 2015. Towards Automatic Dominance Breaking for Constraint Optimization Problems. In *IJCAI 2015*, 360–366.

Muggleton, S. H.; Lin, D.; and Tamaddoni-Nezhad, A. 2015. Meta-interpretive Learning of Higher-order Dyadic Datalog: Predicate Invention Revisited. *Mach. Learn.*, 100(1): 49–73.

Nienhuys-Cheng, S.-H.; and Wolf, R. d. 1997. *Foundations of Inductive Logic Programming*. Springer.

O'Sullivan, B. 2010. Automated Modelling and Solving in Constraint Programming. In *AAAI 2010*, 1493–1497.

Piotrów, M. 2020. Uwrmaxsat: Efficient Solver for MaxSAT and Pseudo-Boolean Problems. In *ICTAI 2020*, 132–136.

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

Rossi, F.; Van Beek, P.; and Walsh, T. 2006. *Handbook of Constraint Programming*. Elsevier.

Rumelhart, D. E.; and Norman, D. A. 1976. Accretion, Tuning and Restructuring: Three Modes of Learning.

Shmueli, O. 1987. Decidability and Expressiveness Aspects of Logic Queries. In *PODS 1987*, 237–249.

Silver, T.; et al. 2023. Predicate Invention for Bilevel Planning. In *AAAI 2023*, 12120–12129.

Tamaki, H.; and Sato, T. 1984. Unfold/Fold Transformation of Logic Programs. In *ICLP 1984*, 127–138.

Vo, H.-P. 2020. Reformulations of Constraint Satisfaction Problems: A Survey.