Learning Logic Programs by Discovering Higher-Order Abstractions

Céline Hocquette\textsuperscript{1}, Sebastijan Dumančić\textsuperscript{2}, Andrew Cropper\textsuperscript{1}
\textsuperscript{1}University of Oxford
\textsuperscript{2}TU Delft
\{celine.hocquette, andrew.cropper\}@cs.ox.ac.uk; s.dumancic@tudelft.nl

Abstract

We introduce the higher-order refactoring problem, where the goal is to compress a logic program by discovering higher-order abstractions, such as \texttt{map}, \texttt{filter}, and \texttt{fold}. We implement our approach in \texttt{STEVIE}, which formulates the refactoring problem as a constraint optimisation problem. Our experiments on multiple programs, including program synthesis and visual reasoning, show that refactoring can improve the learning performance of an inductive logic programming system, specifically improving predictive accuracies by 27\% and reducing learning times by 47\%. We also show that \texttt{STEVIE} can discover abstractions that transfer to multiple domains.

1 Introduction

Abstraction is seen as crucial for AI [Saitta and Zucker, 2013; Russell, 2019; Bundy and Li, 2023]. Despite its argued importance, abstraction is often overlooked in machine learning [Marcus, 2020; Mitchell, 2021]. To address this limitation, we introduce an approach that automatically discovers higher-order abstractions to improve the learning performance of a machine learning algorithm.

To motivate discovering higher-order abstractions, consider learning a logic program from examples to make an input string uppercase, such as $[l,o,g,i,c] \mapsto [L,O,G,I,C]$. For this problem, we could learn the program:

\[
\begin{align*}
    h_1 &= \{ \\
    f(A,B) &\leftarrow \text{empty}(A), \text{empty}(B) \\
    f(A,B) &\leftarrow \text{head}(A,C), \text{uppercase}(C,E), \\
    &\text{head}(B,E), \text{tail}(A,D), f(D,F), \text{tail}(B,F) \\
    \} 
\end{align*}
\]

This program recursively uppercases each element. Although correct, this program is verbose. Alternatively, we could learn:

\[
\begin{align*}
    \{ f(A,B) &\leftarrow \text{map}(A,B,\text{uppercase}) \} 
\end{align*}
\]

This program uses the higher-order abstraction \texttt{map} to avoid needing to learn how to recursively iterate over a list. As this scenario shows, using abstractions can allow us to learn smaller programs, which are often easier to learn than larger ones [Cropper \textit{et al.}, 2020].

The goal of ILP is to induce a hypothesis (a logic program) that generalises the examples with respect to the background knowledge (BK), a logic program which encodes information related to the examples. Recent work in inductive logic programming (ILP) has shown that using user-provided higher-order abstractions, such as \texttt{map}, \texttt{filter}, and \texttt{fold}, can drastically improve the learning performance of an ILP system [Cropper \textit{et al.}, 2020; Purgal \textit{et al.}, 2022]. For instance, if given \texttt{map} as input, these approaches can learn the aforementioned higher-order string transformation program.

The major limitation of these recent approaches is that they need a human to provide the necessary abstractions as input, i.e. these approaches cannot discover abstractions.

To overcome this limitation, we introduce an approach that automatically discovers useful higher-order abstractions, which can then be used by an ILP system. The idea is to refactor a logic program by discovering higher-order abstractions that compress it.

Our refactoring approach works in two stages: abstract and compress. In the abstract stage, given a first-order program, we discover higher-order abstractions. In the compress stage, we search for a subset of the abstractions that compresses the first-order program.

To illustrate our idea, consider the program:

\[
\begin{align*}
    h_2 &= \{ \\
    f(A,B) &\leftarrow \text{empty}(A), \text{empty}(B) \\
    f(A,B) &\leftarrow \text{head}(A,C), \text{increment}(C,E), \\
    &\text{head}(B,E), \text{tail}(A,D), f(D,F), \text{tail}(B,F) \\
    \} 
\end{align*}
\]

This program takes a list of natural numbers and adds one to each element, e.g. $[3,4,5] \mapsto [4,5,6]$.

Suppose we want to refactor the program $P = h_1 \cup h_2$. In the abstract stage, we discover abstractions of $P$, such as\textsuperscript{3}:

\[
\begin{align*}
    h_3 &= \{ \\
    f(A,B) &\leftarrow \text{map}(A,B,\text{uppercase}) \\
    \} 
\end{align*}
\]

The invented relation $\text{ho}$ defines a higher-order abstraction which corresponds to \texttt{map}. The symbol $X$ is a higher-order variable that quantifies over predicate symbols.

In the compress stage, we search for a subset of abstractions that compresses the input program. We formulate this problem as a constraint optimisation problem (COP) [Rossi \textit{et al.}, 2006]. We output a refactored program with abstractions,

\textsuperscript{3}There are more abstractions but we exclude them for brevity.
such as \( P' = h_3 \cup h_4 \), where \( h_4 \) is:

\[
\begin{align*}
  h_4 = \{ & f(A,B) \leftarrow \text{ho}(A,B,\text{uppercase}) \\
  & g(A,B) \leftarrow \text{ho}(A,B,\text{increment}) \}
\end{align*}
\]

In this program, the relations \( f \) and \( g \) are defined with the abstraction \( \text{ho} \). As this example shows, abstractions can compress a program, i.e. \( P' \) has fewer literals (14) than \( P \) (20).

The above scenario shows how discovering higher-order abstractions in one domain can help an ILP system perform better in that domain by allowing it to learn smaller programs. In this paper, we show that abstractions discovered in one domain, such as program synthesis, can be reused by an ILP system in a different domain, such as chess. Although there is much work on transfer learning [Torrey and Shavlik, 2009] and cross-domain transfer learning [Kumaraswamy et al., 2015], as far as we know, we are the first to show the automatic discovery of abstractions that generalise across domains.

1.1 Novelty and Contributions

The three main novelties of this paper are (i) the idea of discovering higher-order abstractions to refactor a logic program, (ii) encoding this refactoring problem as a COP, and (iii) showing cross-domain transfer of discovered abstractions. The impact is that we can drastically improve the learning performance of an ILP system, compared to not discovering abstractions. Moreover, as the idea connects many areas of AI, including machine learning, program synthesis, and constraint optimisation, we hope the idea interests a broad audience.

Overall, our contributions are:

- We introduce the higher-order refactoring problem, where the goal is to refactor a logic program by discovering higher-order abstractions.
- We introduce STEVIE which discovers higher-order abstractions and finds an optimal solution to the higher-order refactoring problem by formulating it as a COP.
- We evaluate our approach on multiple domains, including program synthesis, visual reasoning, and robot strategy learning. Our empirical results show that refactoring can improve the learning performance of an ILP system, specifically improving predictive accuracies by 27% and reducing learning times by 47%. We also show that discovered abstractions can be reused across domains.

2 Related Work

Higher-order logic. Many authors advocate using higher-order logic to represent knowledge [McCarty, 1995; Muggleton et al., 2012]. Although some approaches use higher-order logic to specify the structure of learnable programs [Raedt and Bruynooghe, 1992; Muggleton et al., 2015; Kaminski et al., 2019], most only learn first-order programs [Blockeel and Raedt, 1998; Srinivasan, 2001; De Raedt et al., 2015; Evans and Grefenstette, 2018; Dai and Muggleton, 2021; Evans et al., 2021; Cropper and Morel, 2021]. Some approaches use higher-order abstractions [Cropper et al., 2020; Purgal et al., 2022] but need user-defined abstractions as input. By contrast, we automatically discover abstractions.

Predicate invention. Feng and Muggleton [1992] consider higher-order extensions of Plotkin’s (1971) least general generalisation, where a predicate variable replaces a predicate symbol. By contrast, we introduce new predicate symbols, i.e. we perform predicate invention (PI), a repeatedly stated difficult challenge [Muggleton and Buntine, 1988; Kok and Domingos, 2007; Muggleton et al., 2012; Russell, 2019; Kramer, 2020; Jain et al., 2021; Cropper et al., 2022; Silver et al., 2023]. While most work on predicate invention invents first-order predicate symbols, we invent higher-order symbols.

Representation change. Simon [1981] views abstraction as changing the representation of a problem to make it easier to solve. Propositionalisation [Lavrac and Dzeroski, 1994; Paes et al., 2006] transforms a first-order problem into a propositional one to use efficient propositional learning algorithms. A disadvantage of propositionalisation is the loss of a compact representation language (first-order logic). By contrast, we change a first-order problem to a higher-order one. Theory revision [Adé et al., 1994; Richards and Mooney, 1995; Paes et al., 2017] revises a program so that it entails missing answers or does not entail incorrect answers. Theory refinement improves the quality of a theory, such as its execution or readability [Sommer, 1995; Wrobel, 1996]. By contrast, we refactor a theory to improve learning performance.

Compression. Chaitin [2006] emphasises compression in abstraction. Theory compression [Raedt et al., 2008] selects a subset of a program minimising the impact on performance with respect to the examples. By contrast, we only consider the program, not the examples. ALPS [Dumančić et al., 2019] compresses facts, while we compress logic programs. KNorF [Dumančić et al., 2021] refactors logic programs by framing the problem as a COP. Whereas KNorF performs first-order refactoring, we perform higher-order refactoring. Several approaches [Ellis et al., 2018; Bowers et al., 2023; Cao et al., 2023] refactor functional programs by searching for local changes (new \( \lambda \)-expressions) that increase a cost function. We differ because we (i) consider logic programs, (ii) guarantee optimal compression, and (iii) can transfer knowledge across domains. Moreover, these approaches only evaluate the compression rate, while we show that compressing a program can improve the learning performance of an ILP system.

3 Problem Setting

We assume familiarity with logic programming [Lloyd, 2012] but have included a summary in the appendix. We restate key terminology. A first-order variable can be bound to a constant symbol or another first-order variable. A higher-order variable can be bound to a predicate symbol or another higher-order variable. A clause is a set of literals. A clause is higher-order if it has at least one higher-order variable. A definite clause is a clause with exactly one positive literal. We use the term rule synonymously with definite clause. A definite program is a set of definite clauses with the least Herbrand model semantics. We refer to a definite program as a logic program. A logic program is higher-order if it has at least one higher-order clause. The size(\( P \)) of the logic program \( P \) is the number of literals in \( P \). A definition is
a set of rules with the same head predicate symbol (positive predicate symbol). The set of definitions of the logic program \(P\) with the head predicate symbols \(T\) is \(\delta(P) = \bigcup_{r \in T} \{r \in P\} \text{ the head predicate symbol of the rule } r \text{ is } p\).

### 3.1 Abstraction and Instantiation

The idea of an abstraction is to replace predicate symbols with predicate variables in the body of a rule and to add these variables to the head of the rule. We define an abstraction:

**Definition 1 (Abstraction).** Let \(P\) be a logic program, \(d \in \delta(P)\) be a definition with the head predicate symbol \(h\) of arity \(k\), \(\{p_1, \ldots, p_n\}\) be a subset of the predicate symbols in the bodies of rules in \(d, x_1, \ldots, x_n\) be higher-order variables, and \(h'\) be an invented predicate symbol not in \(P\). Let \(a\) be the definition obtained from \(d\) by replacing (1) every instance of \(p_i\) with \(x_i\), and (2) every literal \(h(v_1, \ldots, v_k)\) with the literal \(h'(v_1, \ldots, v_k, x_1, \ldots, x_n)\). Then \(a\) is an abstraction of \(P\).

The set of all abstractions of \(P\) is \(A(P)\).

We define invented predicate symbols with the prefix \(\text{ho}\).

**Example 1 (Abstraction).** Consider the rule:

\[
f(A) \leftarrow \text{head}(A,B), \text{one}(B), \text{tail}(A,C), \text{head}(C,D), \text{one}(D)
\]

Some abstractions of this rule are:

\[
\begin{align*}
\text{ho}_1(A,X) & \leftarrow X(A,B), \text{one}(B), \text{tail}(A,C), \text{head}(C,D), \text{one}(D) \\
\text{ho}_2(A,X) & \leftarrow \text{head}(A,B), X(B), \text{tail}(A,C), \text{head}(C,D), X(D) \\
\text{ho}_3(A,X,Y) & \leftarrow X(A,B), Y(B), \text{tail}(A,C), X(C,D), Y(D)
\end{align*}
\]

Consider the recursive definition:

\[
g(A,B) \leftarrow \text{head}(A,B) \\
g(A,B) \leftarrow \text{tail}(A,C), g(C,B)
\]

Some abstractions of this definition are:

\[
\begin{align*}
\text{ho}_1(A,B,X) & \leftarrow X(A,B) \\
\text{ho}_2(A,B,X) & \leftarrow \text{tail}(A,C), \text{ho}_4(C,B,X) \\
\text{ho}_3(A,B,X) & \leftarrow \text{head}(A,B) \\
\text{ho}_4(A,B,X) & \leftarrow X(A,C), \text{ho}_5(C,B,X) \\
\text{ho}_5(A,B,X,Y) & \leftarrow X(A,B) \\
\text{ho}_6(A,B,X,Y) & \leftarrow Y(A,C), \text{ho}_7(C,B,X,Y)
\end{align*}
\]

An instantiation replaces predicate variables in an abstraction with predicate symbols:

**Definition 2 (Instantiation).** Let \(P\) be a logic program, \(h'(v_1, \ldots, v_k)\) be a head literal in \(P\), \(h'(v_1, \ldots, v_k, x_1, \ldots, x_n)\) be a head literal in \(A(P)\), \(x_1, \ldots, x_n\) be higher-order variables, and \(p_1, \ldots, p_n\) be predicate symbols in the bodies of rules in \(P\). Then the rule \(h(v_1, \ldots, v_k) \leftarrow h'(v_1, \ldots, v_k, p_1, \ldots, p_n)\) is an instantiation.

The set of all instantiations of abstractions of \(P\) is \(I(A(P))\).

**Example 2 (Instantiation).** Some instantiations of the abstractions in Example 1 are:

\[
\begin{align*}
f(A) & \leftarrow \text{ho}_2(A,\text{one}) \\
f(A) & \leftarrow \text{ho}_3(A,\text{head},\text{one}) \\
g(A,B) & \leftarrow \text{ho}_6(A,B,\text{head},\text{tail})
\end{align*}
\]

### 3.2 Higher-Order Refactoring Problem

We define the least Herbrand model \(M(P,B)\) of the programs \(P\) and \(B\) as \(M(P \cup B)\). In the following, we assume a program \(B\) denoting BK and concisely note \(M(P,B)\) as \(M(P)\). When we refactor a program, we want to preserve its semantics. However, we only need to preserve the semantics with respect to head predicate symbols. Therefore, we reason about the least Herbrand model restricted to a set of predicate symbols:

**Definition 3 (Restricted least Herbrand model).** Let \(P\) be a logic program, \(M(P)\) be the least Herbrand model of \(P\), and \(T\) be the head predicate symbols of \(P\). Then the least Herbrand model of \(P\) restricted to \(T\) is \(M_T(P) = \{a \in M(P) \mid \text{the predicate symbol of } a \text{ is in } T\}\).

We define the higher-order refactoring problem:

**Definition 4 (Higher-order refactoring problem).** Let \(P\) be a logic program and \(T\) be the head predicate symbols of \(P\). Then the higher-order refactoring problem is to find \(Q \subseteq P \cup A(P) \cup I(A(P))\) such that \(M_T(Q) = M_T(P)\). We call \(Q\) a solution to the refactoring problem.

**Example 3 (Refactoring).** A refactoring of the program \(P\) in Section 1 is \(P'\).

Our goal is to perform optimal refactoring:

**Definition 5 (Optimal refactoring).** Let \(P\) be a logic program, \(T\) be the head predicate symbols of \(P\), and \(\text{cost}\) be a function which maps logic programs to integers. Then \(Q\) is an optimal solution when (i) \(Q\) is a solution to the refactoring problem, and (ii) there is no \(Q' \subseteq P \cup A(P) \cup I(A(P))\) such that \(Q'\) is a solution to the refactoring problem and \(\text{cost}(Q') < \text{cost}(Q)\).

In the next section, we introduce STEVIE, which finds an optimal solution to the refactoring problem.

## 4 STEVIE

Algorithm 1 shows our STEVIE algorithm, which works in two stages: abstract and compress. In the abstract stage, given a first-order logic program, STEVIE builds abstractions and instantiations. In the compress stage, STEVIE searches for a subset of the abstractions and instantiations which compresses the input program. STEVIE formulates this search problem as a COP. We describe these two stages in turn. The appendix includes an example of refactoring.

### 4.1 Abstract

In the abstract stage (line 2), STEVIE builds abstractions and instantiations. To build abstractions for the logic program \(P\), for each definition \(d \in \delta(P)\) and subset \(\psi\) of at most \(k\) predicate symbols in the bodies of rules in \(d\), STEVIE calls the function \(\text{create_abs_inst}(d, \psi)\) (line 10). The value \(k\) is a user parameter. This function follows Definition 1 and replaces every \(p_i \in \psi\) in \(d\) with a new higher-order variable \(v_i\), adds each \(v_i\) to the arguments of the literals with the predicate symbol \(h\), where \(h\) is the head predicate symbol of \(d\), and replaces every occurrence of \(h\) with an invented predicate symbol \(\text{ho}\). For instance, if \(d\) is the rule in Example 1 and \(\psi=\{\text{head, one}\}\), the function replaces \(\text{head} \) with \(X\) and \(\text{one} \) with \(Y\) to build the abstraction \(\text{ho}_3\) in Example 1. STEVIE
Algorithm 1 STEVIE

```python
def stevie(P, k):
    abstractions, instantiations = abstract(P, k)
    return compress(P, abstractions, instantiations)

def abstract(P, k):
    abstractions, instantiations = {}, {}
    for d in δ(P):
        size in 1 to k:
            for ψ in subsets(nonrecbodypreds(d), size):
                abs, inst = create_abs_inst(d, ψ)
                if equivalent(abs, abstractions):
                    inst = redefine(inst, abs, abstractions)
                else:
                    abstractions += {abs}
                    instantiations += {inst}
    return abstractions, instantiations
```

neither abstracts recursive predicate symbols (line 9) as this would change the semantics. This function also returns an instantiation (Definition 2) by replacing predicate variables in an abstraction with ψ. STEVIE prunes abstractions that are identical up to renaming of their head predicate symbol (line 11). In such cases, STEVIE redefines the instantiation in terms of the existing equivalent abstraction (line 12). For instance, consider the rules:

\[
\begin{align*}
    f_1(A) & \leftarrow \text{head}(A,B), \text{one}(B) \\
    f_2(A) & \leftarrow \text{head}(A,B), \text{two}(B)
\end{align*}
\]

The abstractions of the \( f_1 \) and \( f_2 \) rules with \( ψ = \{\text{one}\} \) and \( ψ = \{\text{two}\} \) respectively are equivalent up to renaming of the head predicate symbols, i.e., both of these rules have the abstraction \( ho(A,X) \leftarrow \text{head}(A,B), X(B) \).

### 4.2 Compress

In the compress stage, STEVIE searches for a subset of abstractions and instantiations that compresses the input program (line 3). STEVIE formulates this search problem as a COP. 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. We assume an input logic program \( P \).

#### Decision Variables

STEVIE uses three types of decision variables. First, for each definition \( d \in δ(P) \) and abstraction \( a \in A(P) \), we use a Boolean variable \( i^d_a \) to indicate whether an instantiation of \( a \) defining \( d \) is selected. We later use these variables to ensure that a definition is defined with at most one instantiation. Second, for each definition \( d \in δ(P) \), we use a Boolean variable \( n_d \) to indicate that no instantiation has been selected for \( d \). These variables allow STEVIE to not introduce abstractions and instantiations if they overall increase the complexity of the refactored program. Third, for each abstraction \( a \in A(P) \), we use a Boolean variable \( s_a \) to indicate that at least one instantiation of \( a \) is selected. STEVIE uses these variables to determine the size of the refactored program.

#### Constraints

STEVIE imposes two types of constraints. First, for each definition \( d \in δ(P) \), STEVIE uses a constraint to ensure that at most one instantiation is selected for \( d \):

\[
\left( \sum_{a \in A(P)} i^d_a \right) + n_d = 1
\]

This constraint is necessary to identify definitions which are not refactored.

Second, for each abstraction \( a \in A(P) \), STEVIE uses a constraint to ensure that the variable \( s_a \) is true if and only if an instantiation of \( a \) is used to refactor at least one definition:

\[
s_a \leftrightarrow \bigvee_{d \in δ(P)} \ i^d_a
\]

#### Objective

Our objective function is the summation of three components: (1) the size of non-abstracted definitions, (2) the size of selected abstractions and instantiations, and (3) a penalty on the number of higher-order variables. We describe these in turn.

The size of non-abstracted definitions is:

\[
\sum_{d \in δ(P)} \text{size}(d) \times n_d
\]

An instantiation is a rule with one body literal so has size 2. The size of selected abstractions and instantiations is:

\[
\sum_{a \in A(P)} \text{size}(a) \times s_a + \sum_{d \in δ(P), a \in A(P)} 2 \times i^d_a
\]

STEVIE penalises the number of higher-order variables in a refactoring. Without it, STEVIE often selects abstractions that remove all the predicate symbols in a definition. For instance, STEVIE might introduce abstractions such as:

\[
ho(A,B,X,Y,Z) \leftarrow X(A,C), Y(C,D), Z(D,B)
\]

Therefore, STEVIE uses the following penalty, where \( \text{ho-vars}(a) \) is the number of higher-order variables in the abstraction \( a \):

\[
\sum_{a \in A(P)} \text{ho-vars}(a) \times s_a
\]

As we show in our experiments, this penalty allows us to find abstractions that lead to better learning performance.

### 4.3 Correctness

We prove the correctness of STEVIE:

**Theorem 1.** STEVIE solves the optimal refactoring problem with respect to our objective function.

The proof is in the appendix. To show this result, we show that (i) STEVIE generates all abstractions and instantiations (Definitions 1 and 2), (ii) any solution to the encoding is a solution to the higher-order refactoring problem (Definition 4), and (iii) the solver finds an optimal solution (Definition 5) with respect to our objective function.

---

2The OR-tools solver that we use treats Boolean variables as integer variables with domain \{0, 1\}. Therefore, both arithmetic and Boolean operators apply to them.
5 Experiments

To test our claim that higher-order refactoring can improve the performance of an ILP system, our experiments aim to answer the question:

Q1 Can higher-order refactoring improve predictive accuracies and reduce learning times?

To answer Q1, we compare the performance of an ILP system with and without the ability to use abstractions discovered by STEVIE. We use the ILP system HOPPER [Purgal et al., 2022] because it can learn recursive programs, perform predicate invention, and use higher-order abstractions as BK\(^3\).

To understand the impact of penalising the number of higher-order variables (component (3) in Section 4.2), our experiments aim to answer the question:

Q2 What is the impact of penalising the number of higher-order variables on learning performance?

To answer Q2, we compare STEVIE with and without the penalty on the number of higher-order variables.

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

Q3 How long does STEVIE take given larger programs?

To answer Q3, we measure the refactoring time of STEVIE on progressively larger programs.

To test our claim that abstractions discovered in one domain can be reused in different domains, our experiments aim to answer the question:

Q4 Can higher-order refactoring improve performance across domains?

To answer Q4, we compare the performance of HOPPER with and without abstractions discovered in a different domain.

Settings. HOPPER uses types to restrict the hypothesis space (the set of all programs). We use a bottom-up procedure to infer types for the abstractions discovered by STEVIE from the types of the first-order BK. STEVIE does not use types. We set HOPPER to use at most three abstractions in a program. We allow HOPPER to use three threads. We use SWI-Prolog to execute the programs learned by STEVIE and HOPPER. We allow STEVIE to discover abstractions with at most three higher-order variables. STEVIE uses the CP-SAT solver [Perron and Furnon, 2019]. STEVIE uses a single CPU. We use a c6a AWS instance with 32vCPU and 64GB of memory.

Method. We measure the predictive accuracy (the proportion of correct predictions on test data) and learning time of HOPPER. We use a maximum learning time of 15 minutes per task and return the best solution found by HOPPER in this time limit. We use a timeout of 1 hour for STEVIE and return the best refactoring found in this time limit. We repeat all the experiments 5 times and calculate the mean and standard error. The error bars in the figures and tables denote standard error. We rename the abstractions in the figures for clarity.

3We also considered METAGOL\(_{HO}\) [Cropper et al., 2020] but it needs user-provided metarules which are difficult to obtain [Cropper et al., 2022].

5.1 Q1: Learning Performance

Domain. We use a dataset of 176 program synthesis tasks and reserve 25% as held-out tasks. The tasks are designed to use a variety of higher-order constructs and require learning recursive programs. For instance, the dataset includes the tasks `counteven`, `filterodd` (Figure 2a), and `maxlist` (Figure 3b). The appendix contains more details, such as example solutions.

Method. Our method has three steps. In step 1, we use HOPPER to independently learn solutions for \(n\) tasks. In step 2, we use STEVIE to refactor the programs learned in step 1. In step 3, we add the abstractions discovered in step 2 by STEVIE to the BK of HOPPER. We then use HOPPER on the held-out tasks. We vary the number \(n\) of tasks in step 1 and measure the performance of HOPPER in step 3. The baseline (no refactoring) is when we do not use STEVIE in step 2, i.e. the baseline is HOPPER without the abstractions discovered by STEVIE. As a second baseline, we use seven standard higher-order abstractions (maplist, foldl, scanl, convlist, partition, include, and exclude) from the SWI-Prolog library apply\(^4\).

The appendix includes a description of these abstractions.

Results

Figure 1a shows that our approach (STEVIE) can increase predictive accuracies by 27% compared to the baselines. Figure 1b shows that our approach can reduce learning times by 47% compared to the baselines. A chi-square test and a Mann-Whitney U rank test confirm \((p < 0.01)\) the significance of the difference in accuracy and learning times respectively.

To illustrate higher-order refactoring, consider the tasks `filterod` and `filterpos`. Figures 2a and 2b show the programs learned by HOPPER for these tasks. STEVIE compresses these programs by discovering the abstraction shown in Figure 2c. This abstraction keeps elements in a list where the higher-order predicate \(Y\) holds and removes elements where the higher-order predicate \(X\) holds, i.e. this abstraction filters a list. STEVIE thus compresses the program from 30 literals (Figures 2a and 2b) to 19 literals (Figures 2c and 2d).

As a second illustration, consider the tasks `multlist` (Figure 3a) and `maxlist` (Figure 3b). STEVIE compresses these programs by discovering the abstraction `fold` (Figure 3c). This abstraction recursively combines the elements of a list using the higher-order predicate \(X\) and the default value given by the higher-order predicate \(Y\). STEVIE thus compresses the program from 16 literals (Figures 3a and 3b) to 12 (Figures 3c and 3d). Moreover, HOPPER reuses the abstraction `fold` to learn programs for more complex tasks. For instance, without abstraction, HOPPER learns a program for `sumlistplus3` with 10 literals (Figure 3e), whereas with the abstraction `fold` it learns a solution with only 6 literals (Figure 3f).

STEVIE can discover many abstractions, such as `map`, `count`, `iterate`, `until`, `member`, and `all`. The appendix includes all the abstractions discovered by STEVIE. HOPPER can combine these abstractions to learn succinct programs for complex tasks. For instance, for the task `sumunicodes`, HOPPER learns a compact solution (1 rule and 3 literals) which uses the abstractions `map` and `fold`. Without abstractions, HOPPER would need to learn a program with at least 5 rules and 21 literals.

4https://www.swi-prolog.org/pldoc/man?section=apply
Whitney U rank test confirm (work shows [Cropper et al., 2020; Purgal et al., 2022], learning
of programs learned by HOPPER from 8 to 4 literals. As recent
work shows [Cropper et al., 2020; Purgal et al., 2022], learning
smaller programs can improve learning performance since the
system searches a smaller hypothesis space.

Overall, these results suggest that higher-order refactoring
is not equally helpful and that finding good ones is important.

5.2 Q2: Higher-Order Variables Penalty

Figures 1a and 1b show that penalising the number of higher-
order variables can increase predictive accuracies by 8% and
decrease learning times by 37%. A chi-square test and a Mann-
Whitney U rank test confirm (p < 0.01) the significance of
the difference in accuracy and learning times respectively. This
result suggests that component (3) of our objective function
can improve performance. Without this penalty, STEVIE finds
abstractions with many higher-order variables. These abstrac-
tions are less helpful as HOPPER must search through the space
of all possible instantiations which is larger with many higher-
order variables. This result indicates that all abstractions are
not equally helpful and that finding good ones is important.

Overall, these results suggest penalising the number of higher-
order variables can improve learning performance (Q2).

5.3 Q3: Scalability

Figure 1d shows the running time of STEVIE increases exponen-
tially with the program size (number of literals). As the
We remove abstractions that use a relation undefined in the synthesis domain. As Dumančić, Guns, and Cropper [2021] show, for refactoring problems, a solver can quickly find an almost optimal solution but takes a while to find an optimal one. Overall, these results suggest that the scalability (in terms of proving optimality) of STEVIE is limited (Q3).

5.4 Q4: Transfer Learning

Experiment 1 explores whether discovering abstractions can improve learning performance on a single domain. We now explore whether abstractions discovered in one domain can improve performance in different domains.

Domains. We use 35 existing tasks which all benefit from higher-order abstractions [Lin et al., 2014; Cropper et al., 2020; Cretu and Cropper, 2022; Purgal et al., 2022]. These tasks are from 7 domains: chess, ascii art, string transformations, robot strategies, list manipulation, tree manipulation, and arithmetic. These domains have diverse BK with little overlap. The appendix contains a description of the domains.

Method. Our experimental approach is similar to Experiment 1 but the domains differ in steps 1 and 3. In step 1, HOPPER solves tasks from the program synthesis domain. In step 2, STEVIE discovers abstractions from the programs learned in step 1. In step 3, HOPPER solves tasks in a transfer domain. We infer the type of abstractions discovered by STEVIE from the types of the BK in the synthesis domain and use a hard-coded type mapping to transfer them to other domains. We remove abstractions that use a relation undefined in the target domain to ensure they can be executed. The baseline is not applying STEVIE in step 2 (no refactoring).

Experiment 2 shows that compression alone is not the best metric for identifying abstractions which improve learning performance the most. Future work should investigate alternative objective functions.

Refactoring time. Experiment 3 shows that STEVIE can optimally refactor programs with around 460 literals in 16 minutes but struggles on larger programs. Future work should improve scalability, such as improving our COP encoding and using parallel COP solving.

6 Conclusions and Limitations

We introduced an approach that refactors a logic program by discovering higher-order abstractions. We implemented our approach in STEVIE, which formulates this refactoring problem as a COP. Our experiments on multiple domains show that higher-order refactoring can drastically improve the performance of an ILP system, namely improving predictive accuracies and reducing learning times. Our results also show that abstractions discovered in one domain can transfer to different domains. For instance, we can discover the abstractions map, filter, and fold in the program synthesis domain and use them in the chess domain.

6.1 Limitations

Objective function. Experiment 2 shows that compression alone is not the best metric for identifying abstractions which improve learning performance the most. Future work should investigate alternative objective functions.

Appendices, Code, and Data

A longer version of this paper with the appendices is available at https://arxiv.org/pdf/2308.08334.pdf. The experimental code and data are available at https://github.com/celinehocquette/ijcai24-stevie.
Acknowledgements

The first and third authors are supported by the EPSRC fellowship (EP/V040340/1). The authors thank David Cerna, Filipe Gouveia, and Minghao Liu for valuable feedback. For open access, the authors have applied a CC BY public copyright licence to any author-accepted manuscript version arising from this submission.

References


