Learning Logic Programs by Discovering Higher-Order Abstractions

Céline Hocquette[1], **Sebastijan Dumančić**[2], **Andrew Cropper**[1]
[1]University of Oxford
[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 *map*, *filter*, and *fold*. We implement our approach in STEVIE, which formulates the refactoring problem as a constraint optimisation problem. Our experiments on multiple domains, 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 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:

$$h_1 = \left\{ \begin{array}{l} \textit{f(A,B)} \leftarrow \textit{empty(A), empty(B)} \\ \textit{f(A,B)} \leftarrow \textit{head(A,C), uppercase(C,E),} \\ \qquad\qquad \textit{head(B,E), tail(A,D), f(D,F), tail(B,F)} \end{array} \right\}$$

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

$$\{\ \textit{f(A,B)} \leftarrow \textit{map(A,B,uppercase)}\ \}$$

This program uses the higher-order abstraction *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 *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 *map*, *filter*, and *fold*, can drastically improve the learning performance of an ILP system [Cropper *et al.*, 2020; Purgal *et al.*, 2022]. For instance, if given *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:

$$h_2 = \left\{ \begin{array}{l} \textit{g(A,B)} \leftarrow \textit{empty(A), empty(B)} \\ \textit{g(A,B)} \leftarrow \textit{head(A,C), increment(C,E),} \\ \qquad\qquad \textit{head(B,E), tail(A,D), g(D,F), tail(B,F)} \end{array} \right\}$$

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[1]:

$$h_3 = \left\{ \begin{array}{l} \textit{ho(A,B,X)} \leftarrow \textit{empty(A), empty(B)} \\ \textit{ho(A,B,X)} \leftarrow \textit{head(A,C), X(C,E), head(B,E),} \\ \qquad\qquad \textit{tail(A,D), ho(D,F,X), tail(B,F)} \end{array} \right\}$$

The invented relation *ho* defines a higher-order abstraction which corresponds to *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 *et al.*, 2006]. We output a refactored program with abstractions,

[1]There are more abstractions but we exclude them for brevity.

such as $P' = h_3 \cup h_4$, where h_4 is:

$$h_4 = \left\{ \begin{array}{l} f(A,B) \leftarrow ho(A,B,uppercase) \\ g(A,B) \leftarrow ho(A,B,increment) \end{array} \right\}$$

In this program, the relations f and g are defined with the abstraction *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 [McCarthy, 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 λ-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 $\delta(P) = \cup_{p \in T}\{r \in P|$ the head predicate symbol of the rule r 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 $\mathcal{A}(P)$.

We denote invented predicate symbols with the prefix *ho*.

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

f(A) ← head(A,B), one(B), tail(A,C), head(C,D), one(D)

Some abstractions of this rule are:

ho_1(A,X) ← X(A,B), one(B), tail(A,C), X(C,D), one(D)
ho_2(A,X) ← head(A,B), X(B), tail(A,C), head(C,D), X(D)
ho_3(A,X,Y) ← X(A,B), Y(B), tail(A,C), X(C,D), Y(D)

Consider the recursive definition:

g(A,B) ← head(A,B)
g(A,B) ← tail(A,C), g(C,B)

Some abstractions of this definition are:

ho_4(A,B,X) ← X(A,B)
ho_4(A,B,X) ← tail(A,C), ho_4(C,B,X)

ho_5(A,B,X) ← head(A,B)
ho_5(A,B,X) ← X(A,C), ho_5(C,B,X)

ho_6(A,B,X,Y) ← X(A,B)
ho_6(A,B,X,Y) ← Y(A,C), ho_6(C,B,X,Y)

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 $\mathcal{A}(P)$, $x_1, \ldots, x_n$ be higher-order variables, and $p_1, \ldots, p_n$ be predicate symbols in the bodies of rules in P. Then the rule $h(v_1, ..., v_k) \leftarrow h'(v_1, ..., v_k, p_1, \ldots, p_n)$ is an *instantiation*. The set of all instantiations of abstractions of P is $\mathcal{I}(\mathcal{A}(P))$.

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

f(A) ← ho_2(A,one)
f(A) ← ho_3(A,head,one)
g(A,B) ← ho_6(A,B,head,tail)

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 $M_T(P) = \{a \in M(P) \mid$ the predicate symbol of a 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 $Q \subseteq P \cup \mathcal{A}(P) \cup \mathcal{I}(\mathcal{A}(P))$ 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 *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 $Q' \subseteq P \cup \mathcal{A}(P) \cup \mathcal{I}(\mathcal{A}(P))$ such that Q' is a solution to the refactoring problem and $cost(Q') < cost(Q)$.

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 ψ of at most k predicate symbols in the bodies of rules in d, STEVIE calls the function *create_abs_inst(d, ψ)* (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 x_i, adds each x_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 h'. For instance, if d is the rule in Example 1 and ψ={*head, one*}, the function replaces *head* with *X* and *one* with *Y* to build the abstraction ho_3 in Example 1. STEVIE

Algorithm 1 STEVIE

```
1  def stevie(P, k):
2    abstractions, instantiations = abstract(P, k)
3    return compress(P, abstractions, instantiations)
4
5  def abstract(P, k):
6    abstractions, instantiations = {}, {}
7    for d in δ(P):
8      for size in 1 to k:
9        for ψ in subsets(nonrecbodypreds(d), size):
10         abs, inst = create_abs_inst(d, ψ)
11         if equivalent(abs, abstractions):
12           inst = redefine(inst, abs, abstractions)
13         else:
14           abstractions += abs
15         instantiations += {inst}
16   return abstractions, instantiations
```

never 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:

$$f_1(A) \leftarrow head(A,B),\ one(B)$$
$$f_2(A) \leftarrow head(A,B),\ two(B)$$

The abstractions of the f_1 and f_2 rules with $\psi = \{one\}$ and $\psi = \{two\}$ respectively are equivalent up to renaming of the head predicate symbols, i.e. both of these rules have the abstraction *ho(A, X) ← 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 $d \in \delta(P)$ and abstraction $a \in \mathcal{A}(P)$, we use a Boolean variable i_a^d 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 $d \in \delta(P)$, 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 $a \in \mathcal{A}(P)$, 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 $d \in \delta(P)$, STEVIE uses a constraint to ensure that at most one instantiation is selected for d:

$$\left(\sum_{a \in \mathcal{A}(P)} i_a^d \right) + n_d = 1$$

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

Second, for each abstraction $a \in \mathcal{A}(P)$, 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[2]:

$$s_a \leftrightarrow \bigvee_{d \in \delta(P)} i_a^d$$

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:

$$\sum_{d \in \delta(P)} size(d) \times n_d \tag{1}$$

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

$$\underbrace{\sum_{a \in \mathcal{A}(P)} size(a) \times s_a}_{\text{selected abstractions}} + \underbrace{\sum_{d \in \delta(P), a \in \mathcal{A}(P)} 2 \times i_a^d}_{\text{selected instantiations}} \tag{2}$$

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 *ho_vars(a)* is the number of higher-order variables in the abstraction a:

$$\sum_{a \in \mathcal{A}(P)} ho\_vars(a) \times s_a \tag{3}$$

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.

[2]The 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.

[3]We 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

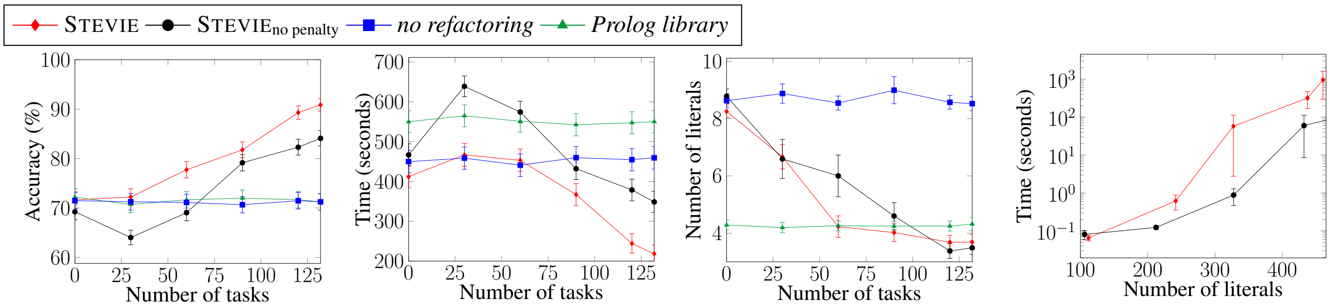
Figure 1a shows that our approach (STEVIE) can increase predictive accuracies by 27% compared to the baselines. Figure 1b shows that our approach can reduce learning times by 47% compared to the baselines. A chi-square test and a Mann-Whitney U rank test confirm ($p < 0.01$) the significance of the difference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

To illustrate higher-order refactoring, consider the tasks *filterodd* 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.

[4]https://www.swi-prolog.org/pldoc/man?section=apply



(a) Predictive accuracy versus the number of tasks.

(b) Learning time versus the number of tasks.

(c) Average solution length versus the number of tasks.

(d) Optimal refactoring time versus the program size.

Figure 1: Results for the *program synthesis* domain.

```
filterodd(A,B) ← empty(A),empty(B)
filterodd(A,B) ← head(A,C),tail(A,D),odd(C),
                 filterodd(D,B)
filterodd(A,B) ← head(A,C),tail(A,D),even(C),
                 filterodd(D,E),head(B,C),tail(B,E)
```

(a) *filterodd* program which removes the odd elements of a list.

```
filterpos(A,B) ← empty(A),empty(B)
filterpos(A,B) ← head(A,C),tail(A,D),pos(C),
                 filterpos(D,B)
filterpos(A,B) ← head(A,C),tail(A,D),neg(C),
                 filterpos(D,E),head(B,C),tail(B,E)
```

(b) *filterpos* program which removes positive elements of a list.

```
ho_filter(A,B,X,Y) ← empty(A),empty(B)
ho_filter(A,B,X,Y) ← head(A,C),tail(A,D),X(C),
                     ho_filter(D,B,X,Y)
ho_filter(A,B,X,Y) ← head(A,C),tail(A,D),Y(C),head(B,C),
                     ho_filter(D,E,X,Y),tail(B,E)
```

(c) Higher-order *ho_filter* abstraction discovered by STEVIE which returns elements of a list where Y holds and X does not.

```
filterodd(A,B) ← ho_filter(A,B,odd,even)
filterpos(A,B) ← ho_filter(A,B,pos,neg)
```

(d) Instantiations.

Figure 2: Example of STEVIE discovering the higher-order abstraction *ho_filter* to compress programs.

```
multlist(A,B) ← empty(A),one(B).
multlist(A,B) ← head(A,C),tail(A,D),
                multlist(D,E),mult(C,E,B)
```

(a) *multlist* program which returns the product of a list elements.

```
maxlist(A,B) ← empty(A),zero(B).
maxlist(A,B) ← head(A,C),tail(A,D),
               maxlist(D,E),max(C,E,B)
```

(b) *maxlist* program which returns the maximum element of a list.

```
ho_fold(A,B,X,Y) ← empty(A),X(B)
ho_fold(A,B,X,Y) ← head(A,C),tail(A,D),
                   ho_fold(D,E,X,Y),Y(C,E,B)
```

(c) Higher-order *ho_fold* abstraction discovered by STEVIE which recursively combines all elements of a list using the higher-order predicate X and the default value returned by Y.

```
multlist(A,B) ← ho_fold(A,B,one,mult)
maxlist(A,B) ← ho_fold(A,B,zero,max)
```

(d) Instantiations.

```
sumlistplus3(A,B) ← empty(A),one(C),succ(C,D),succ(D,B)
sumlistplus3(A,B) ← head(A,C),tail(A,D),
                    sumlistplus3(D,E),sum(C,E,B)
```

(e) *sumlistplus3* program.

```
sumlistplus3(A,B) ← ho_fold(A,B,inv,sum)
inv(A) ← one(B),succ(B,C),succ(C,A)
```

(f) *sumlistplus3* program using the abstraction *ho_fold*. The predicate *inv* is invented by HOPPER.

Figure 3: Example of STEVIE discovering the higher-order abstraction *ho_fold* to compress programs.

Figure 1c shows that refactoring typically reduces the size 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 can substantially improve learning performance (**Q1**).

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 abstractions 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 exponentially with the program size (number of literals). As the size increases, STEVIE builds more abstractions, leading to more decision variables in the compress stage. Note that the running time is the time STEVIE needs to find an optimal refactoring and prove optimality. 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*).

Task	Baseline	STEVIE
do5times	50 ± 0	$\mathbf{100 \pm 0}$
line1	50 ± 0	$\mathbf{100 \pm 0}$
line2	50 ± 0	$\mathbf{100 \pm 0}$
string1	50 ± 0	$\mathbf{100 \pm 0}$
string2	50 ± 0	$\mathbf{100 \pm 0}$
string3	50 ± 0	$\mathbf{100 \pm 0}$
string4	50 ± 0	$\mathbf{100 \pm 0}$
chessmapuntil	50 ± 0	$\mathbf{98 \pm 1}$
chessmapfilter	50 ± 0	$\mathbf{100 \pm 0}$
chessmapfilteruntil	50 ± 0	$\mathbf{98 \pm 1}$
droplastk	50 ± 0	$\mathbf{100 \pm 0}$
encryption	50 ± 0	$\mathbf{100 \pm 0}$
length	80 ± 12	$\mathbf{100 \pm 0}$
rotateN	50 ± 0	$\mathbf{100 \pm 0}$
waiter	50 ± 0	$\mathbf{100 \pm 0}$

Table 1: Predictive accuracies. We only include tasks where the two approaches differ. The full table is in the appendix.

Results. Table 1 shows the predictive accuracies. The learning times are in the appendix. These results show that transferring abstractions never degrades accuracies, and improves accuracies in 5/7 transfer domains. A paired t-test confirms ($p < 0.01$) the significance of the difference in accuracy for all tasks in Table 1 except *length*. For instance, STEVIE discovers the abstractions *filter* and *map* in the *program synthesis* domain and HOPPER uses these abstractions for the task *string1* to learn a program which filters lowercase letters and lowercases the remaining ones. HOPPER also reuses these abstractions to learn a solution for the task *chessmapfilter*. Similarly, HOPPER reuses the abstraction *until* to draw a diagonal line in the *ascii art* domain (Figure 4). HOPPER struggles on some tasks because STEVIE does not discover a helpful abstraction. For instance, the task *isPalindrome* needs the abstraction *condList*, which returns true if the input list is empty and otherwise calls a predicate on the list. STEVIE does not discover this abstraction because it does not compress the input program. HOPPER also struggles on some tasks because of type inconsistencies. For instance, the task *droplast* involves learning a program which, given a list of lists, drops the last element from each list. While STEVIE discovers the abstraction *map*, this abstraction applies to arguments of type *list* instead of *lists of lists*. Overall, these results suggest that higher-order refactoring can improve learning performance in different domains (**Q4**).

```
line2(A,B) ← ho_until(A,B,inv_0,at_right)
inv_0(A,B) ← draw1(A,C),right(C,D),down(D,B)
```

Figure 4: *line2* program which draws a diagonal line in an image. The predicate `inv_0` is invented by HOPPER.

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.

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.

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

[Adé *et al.*, 1994] Hilde Adé, Bart Malfait, and Luc De Raedt. RUTH: an ILP theory revision system. In Zbigniew W. Ras and Maria Zemankova, editors, *Methodologies for Intelligent Systems, 8th International Symposium, ISMIS, Charlotte, North Carolina, USA*, volume 869 of *Lecture Notes in Computer Science*, pages 336–345. Springer, 1994.

[Blockeel and Raedt, 1998] Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. *Artif. Intell.*, 101(1-2):285–297, 1998.

[Bowers *et al.*, 2023] Matthew Bowers, Theo X. Olausson, Lionel Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. Top-down synthesis for library learning. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023.

[Bundy and Li, 2023] Alan Bundy and Xue Li. Representational change is integral to reasoning. *Philos Trans A Math Phys Eng Sci.* ., 2023.

[Cao *et al.*, 2023] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. Babble: learning better abstractions with e-graphs and anti-unification. *Proceedings of the ACM on Programming Languages*, 7(POPL):396–424, 2023.

[Chaitin, 2006] Gregory Chaitin. The limits of reason. *Scientific American*, 294(3):74–81, 2006.

[Cretu and Cropper, 2022] Bogdan Cretu and Andrew Cropper. Constraint-driven multi-task learning. *arXiv preprint arXiv:2208.11656*, 2022.

[Cropper and Morel, 2021] Andrew Cropper and Rolf Morel. Learning programs by learning from failures. *Mach. Learn.*, 110(4):801–856, 2021.

[Cropper *et al.*, 2020] Andrew Cropper, Rolf Morel, and Stephen H. Muggleton. Learning higher-order logic programs. *Mach. Learn.*, 109(7):1289–1322, 2020.

[Cropper *et al.*, 2022] Andrew Cropper, Sebastijan Dumancic, Richard Evans, and Stephen H. Muggleton. Inductive logic programming at 30. *Mach. Learn.*, 111(1):147–172, 2022.

[Dai and Muggleton, 2021] Wang-Zhou Dai and Stephen H. Muggleton. Abductive knowledge induction from raw data. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada*, pages 1845–1851, 2021.

[De Raedt *et al.*, 2015] Luc De Raedt, Anton Dries, Ingo Thon, Guy Van den Broeck, and Mathias Verbeke. Inducing probabilistic relational rules from probabilistic examples. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI, Buenos Aires, Argentina*, pages 1835–1843, 2015.

[Dumančić *et al.*, 2019] Sebastijan Dumančić, Tias Guns, Wannes Meert, and Hendrik Blockeel. Learning relational representations with auto-encoding logic programs. In *28th International Joint Conference on Artificial Intelligence, IJCAI*, pages 6081–6087, 2019.

[Dumančić *et al.*, 2021] Sebastijan Dumančić, Tias Guns, and Andrew Cropper. Knowledge refactoring for inductive program synthesis. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI*, pages 7271–7278, 2021.

[Ellis *et al.*, 2018] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. Learning libraries of subroutines for neurally-guided bayesian program induction. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems, NeurIPS, Montréal, Canada*, pages 7816–7826, 2018.

[Evans and Grefenstette, 2018] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *J. Artif. Intell. Res.*, 61:1–64, 2018.

[Evans *et al.*, 2021] Richard Evans, José Hernández-Orallo, Johannes Welbl, Pushmeet Kohli, and Marek Sergot. Making sense of sensory input. *Artificial Intelligence*, 293:103438, 2021.

[Feng and Muggleton, 1992] Cao Feng and Stephen H. Muggleton. Towards inductive generalization in higher order logic. In *Proceedings of the Ninth International Workshop on Machine Learning (ML 1992), Aberdeen, Scotland, UK*, pages 154–162. Morgan Kaufmann, 1992.

[Jain *et al.*, 2021] Arcchit Jain, Clément Gautrais, Angelika Kimmig, and Luc De Raedt. Learning CNF theories using MDL and predicate invention. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021*, pages 2599–2605, 2021.

[Kaminski *et al.*, 2019] Tobias Kaminski, Thomas Eiter, and Katsumi Inoue. Meta-interpretive learning using hex-programs. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI, Macao, China*, pages 6186–6190, 2019.

[Kok and Domingos, 2007] Stanley Kok and Pedro M. Domingos. Statistical predicate invention. In Zoubin Ghahramani, editor, *Machine Learning, Proceedings of the Twenty-Fourth International Conference (ICML), Corvallis, Oregon, USA*, volume 227 of *ACM International Conference Proceeding Series*, pages 433–440. ACM, 2007.

[Kramer, 2020] Stefan Kramer. A brief history of learning symbolic higher-level representations from data (and a curious look forward). In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 4868–4876, 2020.

[Kumaraswamy *et al.*, 2015] Raksha Kumaraswamy, Phillip Odom, Kristian Kersting, David Leake, and Sriraam Natarajan. Transfer learning via relational type matching. In *IEEE*

International Conference on Data Mining, ICDM, Atlantic City, NJ, USA, pages 811–816. IEEE Computer Society, 2015.

[Lavrac and Dzeroski, 1994] Nada Lavrac and Saso Dzeroski. *Inductive logic programming - techniques and applications*. Ellis Horwood series in artificial intelligence. Ellis Horwood, 1994.

[Lin *et al.*, 2014] Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua B. Tenenbaum, and Stephen H. Muggleton. Bias reformulation for one-shot function induction. In *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 525–530. IOS Press, 2014.

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

[Marcus, 2020] Gary Marcus. The next decade in AI: four steps towards robust artificial intelligence. *CoRR*, abs/2002.06177, 2020.

[McCarthy, 1995] John McCarthy. Making robots conscious of their mental states. In *Machine Intelligence 15*, 1995.

[Mitchell, 2021] Melanie Mitchell. Abstraction and analogy-making in artificial intelligence. *CoRR*, abs/2102.10717, 2021.

[Muggleton and Buntine, 1988] Stephen H. Muggleton and Wray L. Buntine. Machine invention of first order predicates by inverting resolution. In John E. Laird, editor, *Machine Learning, Proceedings of the Fifth International Conference on Machine Learning, Ann Arbor, Michigan, USA*, pages 339–352. Morgan Kaufmann, 1988.

[Muggleton *et al.*, 2012] Stephen H. Muggleton, Luc De Raedt, David Poole, Ivan Bratko, Peter A. Flach, Katsumi Inoue, and Ashwin Srinivasan. ILP turns 20 - biography and future challenges. *Mach. Learn.*, 86(1):3–23, 2012.

[Muggleton *et al.*, 2015] Stephen H. Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. *Mach. Learn.*, 100(1):49–73, 2015.

[Paes *et al.*, 2006] Aline Paes, Filip Zelezný, Gerson Zaverucha, C. David Page Jr., and Ashwin Srinivasan. ILP through propositionalization and stochastic k-term DNF learning. In *Inductive Logic Programming, 16th International Conference, ILP Santiago de Compostela, Spain, Revised Selected Papers*, volume 4455 of *Lecture Notes in Computer Science*, pages 379–393. Springer, 2006.

[Paes *et al.*, 2017] Aline Paes, Gerson Zaverucha, and Vítor Santos Costa. On the use of stochastic local search techniques to revise first-order logic theories from examples. *Mach. Learn.*, 106(2):197–241, 2017.

[Perron and Furnon, 2019] Laurent Perron and Vincent Furnon. Or-tools. *Google.[Online]. Available: https://developers. google. com/optimization*, 2019.

[Plotkin, 1971] G.D. Plotkin. *Automatic Methods of Inductive Inference*. PhD thesis, Edinburgh University, August 1971.

[Purgal *et al.*, 2022] Stanislaw J. Purgal, David M. Cerna, and Cezary Kaliszyk. Learning higher-order logic programs from failures. In Luc De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI, Vienna, Austria*, pages 2726–2733, 2022.

[Raedt and Bruynooghe, 1992] Luc De Raedt and Maurice Bruynooghe. Interactive concept-learning and constructive induction by analogy. *Mach. Learn.*, 8:107–150, 1992.

[Raedt *et al.*, 2008] Luc De Raedt, Kristian Kersting, Angelika Kimmig, Kate Revoredo, and Hannu Toivonen. Compressing probabilistic prolog programs. *Mach. Learn.*, 70(2-3):151–168, 2008.

[Richards and Mooney, 1995] Bradley L. Richards and Raymond J. Mooney. Automated refinement of first-order horn-clause domain theories. *Mach. Learn.*, 19(2):95–131, 1995.

[Rossi *et al.*, 2006] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., USA, 2006.

[Russell, 2019] Stuart Russell. *Human compatible: Artificial intelligence and the problem of control*. Penguin, 2019.

[Saitta and Zucker, 2013] Lorenza Saitta and Jean-Daniel Zucker. *Abstraction in artificial intelligence and complex systems*. Springer, 2013.

[Silver *et al.*, 2023] Tom Silver, Rohan Chitnis, Nishanth Kumar, Willie McClinton, Tomás Lozano-Pérez, Leslie Pack Kaelbling, and Joshua B. Tenenbaum. Predicate invention for bilevel planning. In *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI , Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI, Washington, DC, USA*, pages 12120–12129. AAAI Press, 2023.

[Simon, 1981] Herbert A Simon. *The sciences of the artificial*. MIT press, 1981.

[Sommer, 1995] Edgar Sommer. FENDER: an approach to theory restructuring (extended abstract). In *Machine Learning: ECML-95, 8th European Conference on Machine Learning, Heraclion, Crete, Greece*, volume 912 of *Lecture Notes in Computer Science*, pages 356–359. Springer, 1995.

[Srinivasan, 2001] Ashwin Srinivasan. The ALEPH manual. *Machine Learning at the Computing Laboratory, Oxford University*, 2001.

[Torrey and Shavlik, 2009] Lisa Torrey and Jude Shavlik. Transfer learning. *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, 1:242, 2009.

[Wrobel, 1996] Stefan Wrobel. First-order theory refinement. In *Advances in Inductive Logic Programming*, pages 14–33, 1996.