Imperial College London
Department of Computing

Efficiently learning efficient programs

Andrew Cropper

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London and
the Diploma of Imperial College London, June 2017

Abstract

Discovering efficient algorithms is central to computer science. In this thesis, we aim to discover efficient programs (algorithms) using machine learning. Specifically, we claim we can *efficiently* learn programs (Claim 1), and learn *efficient* programs (Claim 2).

In contrast to universal induction methods, which learn programs using only examples, we introduce *program induction* techniques which additionally use background knowledge to improve learning efficiency. We focus on inductive logic programming (ILP), a form of program induction which uses logic programming to represent examples, background knowledge, and learned programs.

In the first part of this thesis, we support Claim 1 by using appropriate background knowledge to efficiently learn programs. Specifically, we use logical minimisation techniques to reduce the inductive bias of an ILP learner. In addition, we use higher-order background knowledge to extend ILP from learning first-order programs to learning higher-order programs, including the support for higher-order predicate invention. Both contributions reduce learning times and improve predictive accuracies.

In the second part of this thesis, we support Claim 2 by introducing techniques to learn minimal cost logic programs. Specifically, we introduce Metaopt, an ILP system which, given sufficient training examples, is guaranteed to find minimal cost programs. We show that Metaopt can learn minimal cost robot strategies, such as quicksort, and minimal time complexity logic programs, including non-deterministic programs.

Overall, the techniques introduced in this thesis open new avenues of research in computer science and raise the potential for algorithm designers to discover novel efficient algorithms, for software engineers to automate the building of efficient software, and for AI researchers to machine learn efficient robot strategies.

Acknowledgements

I have been incredibility fortunate to have had Stephen Muggleton as a supervisor, and I thank him for supervising this work. Stephen's enthusiasm for and knowledge of science has been inspiring, and I will be forever grateful to him for providing me with the most intellectually rewarding period of my life.

I thank Katsumi Inoue for acting as my host during multiple research visits to the National Institute of Informatics, and Joshua Tennebaum for acting as my host during a research visit to the Massachusetts Institute of Technology.

Finally, I thank Laura for everything.

Declaration of originality

I declare that the work in this thesis is my own and all related other work is appropriately referenced.

Copyright declaration

Contents

Chapter 1

Introduction

Discovering efficient algorithms is central to computer science, as illustrated by the major open problems in the field [12, 40, 30]. In this thesis, we aim to discover efficient programs (algorithms) using machine learning. Specifically, we claim:

- **Claim 1:** we can *efficiently* machine learn programs
- **Claim 2:** we can machine learn *efficient* programs

Program induction

Machine learning programs from data is called *program induction*. The aim is to learn a program that models a set of examples. For instance, consider the following examples written as Prolog facts[1], where the first argument is the input and the second is the output:

```
f([m,a,c,h,i,n,e],e).
f([l,e,a,r,n,i,n,g],g).
f([a,l,g,o,r,i,t,h,m],m).
```

Given these examples and background predicates *head/2*, *tail/2*, and *empty/1*, a program induction system could learn a program that finds the last element of the input list, such as:

[1] We assume familiarity with Edinburgh Prolog syntax [138]

```
f(A,B):-head(A,B),tail(A,C),empty(C).
f(A,B):-tail(A,C),f(C,B).
```

A program induction system should learn more accurate programs given more examples.

Efficiently learning programs

The idea of machine learning goes back to Turing [134] who anticipated the difficulty in programming a computer with human intelligence and instead suggested building computers that learn similar to how a human child learns. Turing also suggested learning with background knowledge, and hinted at the difficultly of learning without it [88]. In contrast to universal induction methods [123, 124, 70], which induce programs only from examples, program induction systems use background knowledge to improve learning efficiency. In the above example, the background knowledge contains definitions for the predicates *head/2*, *tail/2*, and *empty/1*. Because they assume background knowledge, program induction approaches are less general than universal induction methods, but are more practical because the background knowledge is a form of inductive bias [81] which restricts the hypothesis space. Given no background knowledge, and thus no inductive bias, program induction methods are equivalent to universal induction methods. In the first part of this thesis (Chapters 4 and 5), we support Claim 1 by using appropriate background knowledge to efficiently learn programs.

Learning *efficient* programs

Consider the following examples:

```
f([l,o,g,i,c,a,l],l).
f([i,n,d,u,c,t,i,v,e],i).
f([l,e,a,r,n,i,n,g],n).
```

Given these examples and background predicates *head/2*, *tail/2*, *member/2*, and *msort/2*[2], a program induction system could learn a program that finds the duplicate in the input list. Two such programs are:

Example 1 (Program 1)

[2]A mergesort predicate provided by most Prolog systems

```
f(A,B):-head(A,B),tail(A,C),member(B,C).
f(A,B):-tail(A,C),f(C,B).
```

Example 2 (Program 2)

```
f(A,B):-msort(A,C),f1(C,B).
f1(A,B):-head(A,B),tail(A,C),head(C,B).
f1(A,B):-tail(A,C),f1(C,B).
```

Although the programs give the same result[3], they differ in efficiency. Program 1 goes through the elements of the list in turn checking whether the same element exists in the rest of the list with time complexity $O(n^2)$. By contrast, Program 2 first sorts the list and then goes through checking whether any adjacent elements are the same with time complexity $O(n \log n)$. Although textually larger, both in the number of clauses and literals, Program 2 is more efficient than Program 1. However, existing program induction approaches [123, 124, 70, 92, 87, 97] cannot distinguish between the efficiencies of programs, and instead learn textually simple programs. As the above example shows, smaller programs are not necessarily more efficient than larger ones. In the second part of this thesis (Chapters 6, 7 and 8), we address this limitation and support Claim 2 by introducing techniques to learn efficient programs.

Inductive logic programming

We introduce program induction techniques based on inductive logic programming (ILP) [86], a form of machine learning that uses logic programming to represent examples, background knowledge, and learned programs. Existing ILP approaches [92, 87, 97, 106] cannot distinguish between the efficiencies of programs and instead rely on an Occamist bias to learn textually simple programs, such as those with the fewest literals [67] or clauses [97]. For instance, Golem [92] and Progol [87] could both learn sorting algorithms from examples, but when given background knowledge suitable for learning quicksort, both systems learned variants of insertion sort because the program was smaller and there was no bias for learning more efficient algorithms. By contrast, in Chapter 6, we introduce Metaopt, an ILP system biased towards learning efficient programs.

[3] success set equivalent when restricted to the target predicate

Meta-interpretive learning

We focus on meta-interpretive learning (MIL) [96, 97, 23, 25, 24], a form of ILP based on a Prolog meta-interpreter. In contrast to a standard Prolog meta-interpreter, which tries to prove a goal by fetching first-order clauses whose heads unify with the goal, a MIL learner additionally attempts to prove a goal by fetching higher-order clauses, called metarules, whose heads unify with the goal. The resulting meta-substitutions are saved and can be reused in later proofs. Following the proof of a set of goals, a logic program is formed by projecting the meta-substitutions onto their corresponding metarules, allowing for a form of ILP which supports predicate invention and learning recursive programs, both long-standing challenges in ILP [94]. We contribute to the theory and implementation of MIL.

1.1 Contributions

To support Claims 1 and 2, we make the following contributions:

Efficiently learning programs (Claim 1)

Contribution 1: metarules Program induction systems use background knowledge to restrict the hypothesis space. A MIL learner takes metarules as part of the background knowledge. Metarules are a form of declarative bias [98, 108] that determine the structure of learnable programs which in turn defines the hypothesis space. Selecting which metarules to use is a trade-off between efficiency and expressivity: the hypothesis space increases given more metarules [72], so we wish to use fewer metarules, but if we use too few metarules then we lose expressivity. In Chapter 4, we make these contributions to this problem:

- We use Plotkin's clausal theory reduction algorithm [103] to logically reduce sets of metarules.
- We show that when this approach is applied to a finite hypothesis language, only two metarules are necessary to entail all hypotheses in that language.
- We conduct experiments which show that, compared to learning with non-minimal sets of metarules, learning with minimal sets of metarules improves predictive accuracies and reduces learning times.

Contribution 2: higher-order programs Compared to other forms of machine learning, an advantage of using ILP for program induction is its ability to learn first-order programs [86], which are intrinsically more expressive than propositional programs. In Chapter 5, we extend ILP to support learning higher-order programs by allowing a MIL learner to use higher-order definitions as background knowledge. We show that learning higher-order programs can reduce the textual complexity required to express target classes of programs which in turn reduces the hypothesis space. We introduce Metagol$_{AI}$, a MIL learner which supports learning higher-order programs and higher-order predicate invention, such as inventing predicates for use in the higher-order abstractions *map/3* and *reduce/4*. Overall, we make these contributions:

- We define higher-order *definitions*, *abstractions*, and *inventions*.
- We provide sample complexity results which show that learning higher-order programs can reduce (1) learning times, and (2) the number of examples required to reach high predictive accuracies.
- We introduce Metagol$_{AI}$, a MIL learner which supports learning higher-order programs and higher-order predicate invention.
- We conduct experiments which show that, compared to learning first-order programs, learning higher-order programs can improve predictive accuracies by up to 50% and reduce learning times by four orders of magnitude.

This work is the first to demonstrate higher-order predicate invention and the efficiency and accuracy advantages of using higher-order abstractions [25].

Learning efficient programs (Claim 2)

Contribution 3: cost minimisation problem When learning programs from data, we should aim to learn efficient programs. However, as mentioned, existing program induction systems cannot distinguish between the efficiencies of programs. In Chapter 6, we make these contributions to this problem:

- We introduce the *cost minimisation problem*, a general framework for learning efficient programs.

- We introduce Metaopt, a MIL learner which solves the cost minimisation problem using a new search procedure called iterative descent.
- We prove that, given sufficient training examples, Metaopt converges on minimal cost programs.

Contribution 4: learning efficient robot strategies In Chapter 7, we use Metaopt to learn efficient *resource complexity* robot strategies [24]. In contrast to traditional AI planning [114], which involves the generation of a plan as a sequence of actions transforming a particular initial state to a particular final state, a strategy can be viewed as a potentially infinite set of plans, applicable to a class of initial/final state pairs [24]. Specifically, we make these contributions:

- We introduce the resource complexity minimisation problem, a variant of the cost minimisation problem, where resource complexity is a user-defined measure of the efficiency of a robot strategy.
- We conduct experiments on learning robot librarian, postman, and sorter strategies which show that Metaopt learns minimal resource complexity strategies in all cases.

This work is the first to demonstrate learning efficient robot strategies [24].

Contribution 5: learning efficient logic programs

In Chapter 8, we use Metaopt to learn efficient time complexity logic programs. Specifically, we make these contributions:

- We introduce *tree complexity*, a program cost function based on the size of a SLD-tree at the point of which a goal is proved by a logic program.
- We introduce the *tree complexity* minimisation problem, a variant of the cost minimisation problem.
- We conduct experiments on programming puzzles and real-world string transformation problems which show that Metaopt learns minimal tree complexity programs, including non-deterministic programs, which correspond to minimal time complexity programs.

This work is the first to demonstrate learning efficient time complexity programs.

1.2 Publications

We have published parts of this thesis:

- Parts of Chapter 4 appeared in [23]. I contributed (1) parts of the theoretical framework, in particular the work on the logical reduction of metarules, (2) by conducting the experiments, and (3) by writing half of the paper.
- Parts of Chapter 5 appeared in [25]. I contributed (1) the idea of learning higher-order programs, (2) the implementation Metagol$_{AI}$, (3) by conducting the experiments, and (3) by writing two-thirds of the paper.
- Parts of Chapters 6 and 7 appeared in [24]. I contributed (1) the idea of learning efficient robot strategies, (2) the implementation Metagol$_{O}$, (3) by conducting the experiments, and (3) by writing half of the paper.
- Parts of Chapter 6 and 8 appeared in [27]. I contributed almost all of the work for this paper.

We have also published papers related to this thesis [28, 22, 36].

1.3 Outline

The first part of the thesis (Chapters 4 and 5) focuses on Claim 1 and introduces techniques to efficiently learn programs. The second part of this thesis (Chapters 6, 7, and 8) focuses on Claim 2 and introduces techniques to learn efficient programs. Chapters 4, 5, 6 and 7 each include a discussion of related work specific to that chapter and a brief summary. The rest of this thesis is organised as follows:

Chapter 2: Related work We discuss related work, including algorithms and computation, inductive inference, and program induction.

Chapter 3: Meta-interpretive learning We describe MIL and prerequisite concepts from logic programming and ILP.

Chapter 4: Logical minimisation of metarules We describe work on improving the efficiency of a MIL learner by reducing the number of metarules required.

Chapter 5: Learning higher-order programs We describe work on improving the efficiency of a MIL learner by introducing techniques to learn higher-order programs.

Chapter 6: Metaopt We introduce the *cost minimisation problem*, a general framework for learning efficient programs and Metaopt, a MIL learner which solves the problem.

Chapter 7: Learning efficient robot strategies We describe work on learning robot strategies with minimal resource complexity.

Chapter 8: Learning efficient logic programs We describe work on learning logic programs with minimal tree complexity and thus minimal time complexity.

Chapter 9: Conclusions We conclude the thesis and discuss future work.

1.4 Summary

In this chapter, we have stated that the goal of this thesis is to machine learn programs (algorithms) from data, which we call program induction. We have highlighted that existing approaches cannot learn efficient programs. By contrast, we have claimed we can (1) *efficiently* learn programs, and (2) learn *efficient* programs. We have outlined the contributions of the thesis, namely (1) the introduction of techniques to improve learning efficiency by using appropriate background knowledge, and (2) the introduction of the first algorithm that learns efficient programs. In the next chapter, we cover related literature for the rest of the thesis, including overviews of algorithms, computation, and inductive inference.

Chapter 2

Related work

In this chapter, we detail work related to the thesis, including work on algorithms, computation, inductive inference, program induction, and ILP.

2.1 Algorithms and computation

Informally, an algorithm is a sequence of precise instructions to perform a task. Computation is the process of applying an algorithm to some input. Although mathematicians have studied algorithms for millennia, it was not until the 1930s that the concept of an algorithm was formalised mathematically.

In 1928, Hilbert proposed the *Entscheidungsproblem* [49], which asks whether there exists an 'effectively calculable' procedure (an algorithm) that determines whether a statement in first-order-logic is provable using the rules of logic. Before the question could be answered, the notion of an algorithm had to be formally defined, which was done independently by three logicians. Godel proposed 'general recursive functions' [61], Church proposed λ-definability based on his λ-calculus [17], and Turing proposed theoretical machines, now called Turing machines [135].

Turing showed [135] that these three models were equivalent, i.e. a function is general recursive if and only if it is λ-computable if and only if it is computable on a Turing machine. The connection between the informal and formal notions of algorithm is called the Church-Turing thesis [121], which states that a function is computable by a human following an algorithm, ignoring resource limitations, if and only if it is computable by a Turing machine.

Having formally defined the notion of an algorithm, a negative answer to

the Entscheidungsproblem was given by Church [16] who demonstrated the existence of uncomputable functions in his λ-calculus, and by Turing [135] who showed there cannot exist a general method that decides whether any Turing machine halts, known as the halting problem.

Turing Machines

A Turing machine [135] is a hypothetical machine. It has an infinite tape divided into cells, where the tape acts as memory. Initially the tape contains only the input string and is blank everywhere else. The machine has a tape head that performs three operations: (1) read the symbol on the cell under the head, (2) edit the symbol by writing a new symbol or erasing it, (3) move the tape left or right by one cell. The choice of operation depends on a set of user-specified instructions. The machine continues to execute instructions or halts. A universal Turing machine (UTM) is a Turing machine that can simulate any Turing machine. When a programming language can do what a Turing machine can do, that language is called Turing complete. If a problem is solvable in a Turing complete language then it is solvable in all such languages.

Algorithm efficiency

A decidable function can always be computed on a Turing machine (or an equivalent model) given sufficient resources. However, a decidable function may not necessarily be computed efficiently. Algorithm efficiency refers to the resources required by an algorithm to compute a function. Two common [121] resource measures are (1) time complexity, which measures how long an algorithm takes to compute a function, and (2) space complexity, which measures how much memory an algorithm needs to compute a function. Efficiency is measured as a function of the length of the string representing the input. Worst-case analysis measures the maximum resources used on all inputs of a particular length. Average-case complexity measures the average resources used on all inputs of a particular length. The exact resources required by an algorithm is often difficult to establish. For example, different programming languages may influence the running time of an algorithm. Therefore, we approximate efficiency using asymptotic analysis [121], which measures the efficiency of an algorithm as the input length grows. This approximation only considers the highest order term in an expression of the cost of an algorithm, because the

highest order term dominates the other terms in the limit. For example, the function $f(n) = n^3 + 2n^2 + 5x + 10$ is asymptotically at most n^3 written as $0(n^3)$.

Kolmogorov complexity

Algorithmic information theory [15] studies the resources required to represent strings. For example, consider these two strings:

```
00000000000000000000
10110100110000000011
```

Whilst the first string has the short description "twenty zeros", the second string has no obviously simpler description than the string itself. Kolmogorov complexity [62] measures the resources needed to specify a string x as the shortest program that computes x. The shortest program is typically interpreted as the shortest Turing machine. Kolmogorov complexity is used in inductive inference and machine learning to measure the resources required to represent algorithms.

2.2 Logical reasoning

In this thesis, we aim to learn programs from data, which can be seen as deriving conclusions (programs) from premises (data). Logical reasoning uses formal logic to derive conclusions from premises, where conclusions and premises are formed of rules or facts. Two key concepts in logic are soundness and completeness [99]. A logical system is sound if and only if its inference rules prove only formulas that are valid with respect to its semantics, i.e. soundness is the property of being able to only prove true things. A logical system is complete if and only if all valid formula can be derived from the axioms and the inference rules, i.e. completeness is the property of being able to prove all true things.

We use techniques that combine the three major forms of logical reasoning: deduction, induction, and abduction [102].

Deduction Deduction is the process of deriving facts from premises formed of rules and facts. For example, consider these two statements:

$$\text{mortal(A)} \leftarrow \text{man(A)}$$
$$\text{man(socrates)}$$

Given these statements, we can use the implication elimination rule of deduction to derive the fact $mortal(socrates)$. In deductive reasoning, if all premises are true and the rules of deduction are followed, then derived conclusions are necessarily true. Godel showed [86] that a small set of inference rules is complete for deriving all consequences of formulae in first-order-logic. Later, Robinson demonstrated [113] that a single rule of inference, called resolution, is sound and complete for finding refutations of statements in clausal form (Section 3.1).

Abduction Abduction seeks to explain observations and, as with deduction, is the process of deriving facts from rules and facts. For example, consider these two statements:

$$\text{mortal(A)} \leftarrow \text{man(A)}$$
$$\text{mortal(socrates)}$$

To explain the fact $mortal(socrates)$ we could abduce the fact $man(socrates)$. However, unlike deductive reasoning, abductive reasoning derives conclusions which are not guaranteed to be correct. In this example, the fact $man(socrates)$ is not necessarily true.

Induction Induction is the process of forming rules from examples, and is the goal of this thesis. For example, consider these two statements:

$$\text{man(socrates)}$$
$$\text{mortal(socrates)}$$

Using induction, we could form the rule that all men are mortal:

$$\text{mortal(A)} \leftarrow \text{man(A)}$$

We could also form the rule that all things mortal are men:

$$\text{man(A)} \leftarrow \text{mortal(A)}$$

Conclusions formed in inductive reasoning are not logically valid, i.e. induced rules may be incorrect.

Inducing rules from data is the *induction problem*. The induction problem has a long history in philosophy. Epicurus noted that there typically exist many hypotheses consistent with the data [52]. For example, we can form multiple rules from the two facts above about Socrates. Logically, we cannot use the data to rule out any of these hypotheses; they must all be kept as potential explanations. This notion of keeping all consistent hypotheses is known as Epicurus's principle of multiple explanations. In practice, however, we often want to select one hypothesis, or at least exclude certain hypotheses. Occam's principle (often called razor because it shaves away unnecessary assumptions), suggests that amongst all hypotheses consistent with the data, the simplest is the most likely [52]. For example, consider the number sequence 1, 3, 5, 7. Suppose that n represents the position in the sequence, then one hypothesis to describe this sequence is the expression $(2n)-1$, where the next number is 9. An alternative hypothesis is the expression $2n-1+(n-1)(n-2)(n-3)(n-4)$, where the next number is 33. Although both hypotheses are valid, Occam's principle says prefer the former to the latter because it is simpler. Many learning algorithms [123, 124, 97, 70] rely on an Occamist bias to decide between hypotheses.

2.3 Inductive inference

Solomonoff induction

Solomonoff's universal inductive inference [123, 124] is a solution to the induction problem. It combines Epicurus' and Occam's principles in a probabilistic way using Bayes' theorem [81], represents hypotheses as Turing machines, and uses Kolmogorov complexity as a prior. The idea is that given data E, one way to find a hypothesis H (a Turing machine) that generated E is to try all possible hypotheses on a UTM. To adhere to Occam's principle, Solomonoff induction assigns a prior probability to each hypothesis based on its Kolmogorov complexity. However, Solomonoff induction is incomputable because it tries every possible hypothesis, some of which will run forever. Because of the Halting problem, we cannot determine beforehand that any of the hypotheses will not terminate

Levin search

Levin's universal search [70] is another solution to the induction problem. Similar to Solomonoff induction, Levin search tries all possible hypotheses on a UTM. The key difference is that Levin search allocates each program p time equal to $2^{l(p)}$, where $l(p)$ is the size of the program in bits. If nothing else is known about the problem except the observations and assuming the solution can be verified in polynomial time, Levin search is the asymptotically fastest way of finding a program to solve the problem. The algorithm has the property that the total time taken to find a solution is $O(t)$, where t is the time used by fastest program p to compute the solution. The search time of the whole process is at most a constant factor larger than t. However, this constant is $2^{l(p)}$, making Levin search impractical.

2.4 Automatic programming

Automatic programming is the automatic generation of a computer program to perform a task. Universal induction methods, such as Solomonoff induction and Levin search, are forms of automatic programming. However, universal methods are impractical because they only take examples as input. Other approaches take additional information to improve efficiency.

2.4.1 Deductive approaches

Deductive approaches [77] to automatic programming build programs from full specifications, where a specification precisely states the requirements and behaviour of the desired program. For example, to build a program that returns the last element of a non-empty list, we could provide a formal specification written in Z notation [125]:

```
last : seq_0 X --> X
forall s : seq_0 X last s = s(#s)
```

Deductive approaches also take informal full specifications, such as a specification written as a Prolog program:

```
last([A]).
last([A|B]):-last(B).
```

A drawback of deductive approaches is that formulating a specification is hard and typically requires a domain expert. In fact, formulating a specification can be as hard as finding a solution. For example, formulating the specification for the following string transformations is non-trivial:

```
`alan.turing@cam.ac.uk' => `Alan Turing'
`alonzo.church@princeton.edu' => `Alonzo Church'
`kurt.godel@ias.edu' => `Kurt Godel'
```

2.4.2 Program induction

Deductive approaches take full specifications as input and are efficient at building programs. Universal induction methods take only examples as input and are inefficient at building programs. We focus on the area between these which we call *program induction* – also called inductive programming [47], programming by example [71], and inductive program synthesis [104].

Similar to universal induction methods, program induction systems learn programs from incomplete specifications, typically input/output examples. In contrast to universal induction methods, program induction systems use background knowledge, and are thus less general than universal methods, but are more practical because the background knowledge is a form of inductive bias [81] which restricts the hypothesis space. When given no background knowledge, and thus no inductive bias, program induction methods are equivalent to universal induction methods.

Early work on program induction includes Plotkin on least generalisation [103], Vere on induction algorithms for predicate calculus [137], and Summers on inducing Lisp programs [130]. Interest in program induction has grown recently, partly due to applications in real-world problems, such as end-user programming [45] and computer education [46].

We can classify program induction approaches as either task-specific or general-purpose. Task-specific approaches focus on a specific domain and are often restricted to specific data types, such as numbers [120] and strings [44, 141]. By contrast, general-purpose systems work on many domains, but are typically less efficient. MagicHaskeller [55] is a general-purpose system that learns Haskell functions by instantiating higher-order functions from a pre-defined vocabulary. Igor2 [60] also learns recursive Haskell programs and supports auxiliary function invention but is restricted because it requires the first k examples of a target theory to generalise over a whole class. Esher [3]

learns recursive programs but needs to ask an oracle for examples each time a recursive call is encountered. In contrast to these approaches, this thesis is based on MIL, a general form of program induction which learns recursive Prolog programs from examples and background knowledge.

2.5 Machine learning

Program induction has been studied in many areas of machine learning, such as inductive logic programming [86], genetic programming [140], and deep learning [143]. The goal of machine learning is to develop algorithms that improve their performance over time through experience. Mitchell [81] defines machine learning as:

Definition 1 (Machine learning) A learning algorithm is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

Experience E refers to examples. Performance measure P typically measures predictive accuracy on unseen examples but is only one criterion of performance. Michie [80] suggested three performance criteria. The *weak* criterion measures how well the learned program performs on unseen data (predictive accuracy). The *strong* criterion additionally requires that the program is readable by a human. Finally, the *ultra-strong* criterion additionally requires that a human can understand and draw consequences from the program. Most forms of machine learning only support the weak criterion because the learned hypotheses are largely incomprehensible to a human, such as the hyperplanes learned by a support vector machine. By contrast, the hypotheses in program induction are computer programs, which can be read and understood by a human. Declarative forms of program induction, in which the induced programs express the logic of a computation without describing its control flow [73], such as ILP, are particularly suited for human interpretability.

We can broadly classify machine learning approaches by the type of examples (experience) available. The three common classifications are [81]: supervised learning, unsupervised learning, and semi-supervised learning. In supervised learning the task is to learn a function that generalises from labelled examples to unlabelled examples. In unsupervised learning the task is to learn how unlabelled data is organised. Semi-supervised learning is a mix of supervised

and unsupervised learning, where the examples are labelled and unlabelled. Program induction typically focuses on supervised learning.

2.5.1 Computational learning theory

Computational learning theory studies what can be learned efficiently. Two key criteria of learning efficiency are [81]:

Definition 2 (Sample complexity) Sample complexity is the number of examples an algorithm needs to successfully learn a correct hypothesis.

Definition 3 (Time complexity). Time complexity is the time an algorithm needs to successfully learn a correct hypothesis.

We use both criteria throughout this thesis, but because we can measure the time complexity of a learning algorithm in the same way as any other algorithm (Section 2.1), this section focuses on sample complexity.

In the above definitions, the term *successfully* is vague. What does it mean for a learning algorithm to successfully learn a hypothesis? Two theories have tried to answer this question.

In Gold's theory of *language identification in the limit* [43], a learning algorithm reads an infinite sequence of examples one by one and is said to successfully identify a language in the limit if and only if after a certain number of examples the algorithm chooses the correct hypothesis and does not change this hypothesis as more examples are provided. Two drawbacks of Gold's theory are (1) it represents an all-or-nothing approach to learning, where a hypothesis must be totally correct with respect to all the seen examples, and (2) it gives little indication of how many examples are required to converge on the correct hypothesis [99].

In contrast to Gold's theory, Valiant's theory of probably approximately correct learning (PAC) [136] is concerned with approximations of learnability. In Valiant's theory, a learning algorithm PAC-identifies a concept if and only if it learns with high probability $(1-\delta)$ a hypothesis that is approximately consistent $(1-\epsilon)$ with the examples. This theory gives a looser definition of successfully learning a concept, which is usually considered to be a better model in machine learning [99].

These two theories define when a learning algorithm successfully learns a hypothesis, but neither tells us how many examples are required to do so. Using

PAC-learning, Blumer [9] showed that as a hypothesis space grows, you need more examples to PAC-learn a hypothesis. The result, known as the Blumer bound, implies that given two hypothesis spaces which (1) both contain a target hypothesis, and (2) are of different sizes, then searching the smaller hypothesis space will reduce learning times and improve predictive accuracies compared to searching the larger hypothesis space. The idea of reducing a hypothesis space without excluding the target hypothesis is the focus of Chapters 4 and 5. We formally describe the Blumer bound in Chapter 3.

2.6 Logic programming

Logic has long been viewed as central to achieving AI [134, 78]. Logic programming [129, 74] is a form of declarative programming [73] based on formal logic. In contrast to imperative programming, which views a program as a sequence of step-by-step instructions (a procedure), logic programming views a program as a logical theory, where computation is viewed as finding a proof of the theory (program). A search for a proof is based on deductive reasoning, specifically Robinson's resolution principle [113], a single rule of deductive inference which is refutation complete for theories in clausal form. Resolution takes as input a clausal theory P and a goal G and tries to derive the empty clause, which represents falsity. To improve efficiency, Kowalski introduced SLD-resolution [63], which is sound and complete for a restricted form of a logic called Horn logic. Although restricted, Horn logic is Turing complete [132]. Prolog, a popular logic programming language, is based on Horn logic – although contains extra-logical features, such as cuts. We detail logic programming in Section 3.1.

2.7 Inductive logic programming

ILP is a form of machine learning that uses logic programming to represent examples, background knowledge, and induced hypotheses (programs). We use the *learning from entailment* setting of ILP [93], where an example corresponds to an observation about the truth or falsity of a formula F and a hypothesis H covers F if H entails F ($H \models F$). Two other learning settings are (1) *learning from interpretations* [7], where an example is a logical interpretation I and an example is covered by a hypothesis H if I is a model for H, and (2) *learning from proofs* [101], where an example is a proof P and a hypothesis H covers an

example if P is a proof of the hypothesis H. Learning from proofs in ILP is similar to learning programs from execution traces [66], which some researchers view as program induction [112]. However, proofs and interpretations carry more information than examples and are therefore easier to learn from [94]. Therefore, such approaches should be distinguished from learning from examples.

An advantage of using ILP over other forms of machine learning is that, because of the expressivity of logic (Horn logic is Turing complete [132]), ILP systems can learn complex relational theories, such as solutions to the Michalski trains problem [65]. By contrast, most other forms of machine learning, such a neural networks [69], are restricted to finite, propositional, feature-based representations of examples and concepts, and thus struggle to learn complex relational theories.

Another advantage of ILP is that because hypotheses are logic programs, they can be read by humans, potentially supporting Michie's *strong* and *ultra-strong* criteria of learning, which is not the case in many other learning approaches [94].

Predicate invention Predicate invention has been repeatedly stated as an important challenge in ILP [90, 128, 94]. The idea behind predicate invention is for an ILP system to introduce new predicates to improve learning performance. In program induction, predicate invention can be seen as inventing auxiliary functions, as one does when manually writing a program, for example to reduce code duplication or to improve the readability of a program. Popular ILP systems, such as FOIL [106], Progol [87], and ALEPH [127], do not support predicate invention, nor do most program induction systems. Meta-level abduction [53] uses abduction and meta-level reasoning to invent predicates that represent propositions. By contrast, MIL, on which this thesis is based, uses abduction to invent predicates representing relations, i.e. relations which are not in the initial background knowledge nor in the examples. For instance, in [97], MIL invented a predicate corresponding the parent relation when learning a grandparent relation. In chapter 5, we extend MIL and the associated Metagol implementation to support higher-order predicate invention for use in higher-order constructs, such as *map/3*, *reduce/3*, and *fold/5*.

Recursion Recursion is fundamental to computer science and algorithms [2], yet it has been difficult for ILP to learn recursive programs [94] – which is also the case for program induction in general (Section 2.4.2). The ATRE system

was used to learn a recursive ontology theory from biological text [76] and to learn recursive patterns from biomedical text [5]. However, ARTE cannot easily be extended to learn general programs from examples [28]. Moreover, ARTE was not shown to learn recursive theories from small numbers of examples [28]. By contrast, MIL supports learning general recursive programs, such as learning definite clause grammars and recursive definitions for the concept of a staircase [97]. In this thesis, we further demonstrate the ability of MIL to learn general recursive programs from small numbers of examples, such as learning robot strategies for quicksort (Chapter 7) and to learn efficient time complexity programs to find duplicate elements in a list (Chapter 8).

Higher-order logic McCarthy [79] and Lloyd [75] advocated using higher-order logic to represent knowledge. Similarly, in [94], the authors argued that using higher-order representations in ILP provide more flexible ways of representing background knowledge.

MIL uses higher-order metarules and a meta-interpreter (which is intrinsically higher-order) to learn programs from examples. Metarules are a form of declarative bias [98, 108]. In contrast to other forms of declarative bias in ILP, such as modes [87, 127] or grammars [20], metarules are logical statements that can be reasoned about. Metarules were introduced in the Blip system [35], who used similar metarules to us, such transitivity (chain) and converse (inverse) metarules. Metarules are also called second-order schemata [109].

In [57], the authors explore generality measures for metarules, which they call rule schemas, in their RDT system. A generality order is necessary because the RDT system searches the hypothesis space (which is defined by the metarules) in a top-down general-to-specific order. A key difference between RDT and MIL is that whereas RDT requires metarules of increasing complexity (e.g. rules with an increasing number literals in the body), MIL derives more complex metarules through predicate invention.

Determining which metarules are necessary to learn certain classes of programs has yet to be explored. In Chapter 4, we address this issue by using logical reduction techniques to find logically minimal sets of metarules, which improves predictive accuracies and lowers learning times, and we show that in some cases only two metarules are needed to derive a whole class of metarules

Although learning higher-order programs has been considered in program induction [60, 55], it has been under explored in ILP. In Chapter 5, we extend MIL from learning first-order programs to learning higher-order programs,

which improves predictive accuracies and lowers learning times.

ILP systems In this thesis, we develop Metagol, an ILP system based on the MIL framework. It is difficult to directly compare Metagol to other ILP systems. For instance, some notable ILP systems, such as MIS [119] and CIGOL [90] are interactive, i.e. they require input from a user during the learning, whereas Metagol does not.

As described above, Metagol supports predicate invention, the automatic introduction of new predicate symbols, which has long been a challenge for ILP [90, 128, 94] and which most ILP systems do not support [119, 106, 92, 87, 127, 111, 21, 67]. Systems that do support predicate invention support different levels of predicate invention. For instance, Cigol [90] only introduces new predicates when a negative example is confirmed by an oracle (i.e. Cigol is an interactive system). By contrast, Metagol automatically introduces new predicate symbols without the need for negative examples. In addition, ASPAL [4] and INSPIRE [118] both support predicate invention, but both are yet to demonstrate nested predicate, which is frequently demonstrated in this thesis using Metagol.

In many of our experiments, we learn explicitly recursive programs (i.e. programs where the recursion is not hidden in the background knowledge), such as in Chapter 8 where we use Metagol to learn an efficient and recursive programs to find a duplicate element in a list. Many ILP systems cannot support recursion [90, 92, 21]. Even systems that do support recursion, such as ALPEH [127] and FOIL [106], only support limited recursion, and cannot, for instance, learn recursive grammars from example sequences [96, 97]. By contrast, Metagol has been shown able to learn such recursive grammars [96]. The same issues apply to Progol [87], which due to its limitations for learning recursive theories and predicate invention, is unable to find a complete theory for the Michalski trains problem [97], whereas Metagol can.

Another dimension to compare ILP systems is whether they support non-observational predicate learning [89]. In observational predicate learning (OPL), the examples are described by the same predicate as that of the expected hypothesis. However, OPL is inadequate for certain problems, such as learning event calculus domain specific axioms from fluent time-trace observations [83]. To overcome this deficiency a form of non-OPL is required, which is sometimes called *theory completion* [84]. However, Metagol's support for predicate invention blurs the line between OPL and non-OPL, in that part of an induced

hypothesis can be described by predicates not given in examples.

Metagol learns definite clause logic programs, described as Prolog programs. Many recent ILP systems [4, 67, 118] learn answer set programs (ASP), rather than Prolog programs. These systems have advantages over Metagol, such as that ASP are purely declarative, where the order of the clauses and body literals does not matter, which is not the case in Prolog. These ASP approaches can also learn non-monotonic programs (i.e. programs with negated literals), which is not yet supported by Metagol – although some ILP systems can learn non-monotonic Prolog programs, such as XHAIL [111], TAL [21], and IMPARO [59]. However, directly comparing Metagol to ASP-based systems is difficult. One reason for the difficulty is that most of the experiments in this thesis concern learning programs that manipulate lists, such as learning to sort lists of arbitrary length (Chapter 7) and learning complex transformation programs (Chapter 8). However, many ASP systems disallow explicit lists, such as the popular Clingo system [42], and thus a direct comparison is difficult.

Robot strategies ILP research has largely focused on learning to classify examples, i.e. learning a program to determine whether an example belongs to a certain class [94]. However, many problems require more general programs, such as learning *robot strategies* [24]. In contrast to non-recursive robot plans [68], applicable only to a specific initial/final state pair, a recursive strategy is applicable to a potentially infinite set of initial/final state pairs.. Although ILP has been used for robotics [68, 10], learning strategies has been under explored because of the lack of support for learning recursive theories. MIL has been used [97] to learn a recursive robot strategy to build a stable wall from bricks. In this thesis, we further demonstrate the ability for MIL to learn recursive robot strategies, such as learning higher-order robot waiter strategies (Chapter 5) and efficient recursive robot sorting strategies (Chapter 7).

Learning efficiency Techniques to improve learning efficiency in ILP include probabilistic search techniques [126], the use of query packs [6], the use of special purpose hardware [38], and parallelism [32]. By contrast, we focus on using appropriate background knowledge to improve efficiency. Specifically, in Chapter 4, we reduce the hypothesis space of a MIL learner without losing expressivity by reducing the number of metarules used as background knowledge. In Chapter 5, we reduce the learning time of a MIL learner by allowing for higher-order definitions to be defined as background knowledge, which

allows for a MIL learner to learn higher-order programs. We show that the ability to learn higher-order programs reduces the textual complexity required to express target classes of programs which in turn reduces the hypothesis space and learning times.

Efficient programs Although algorithm efficiency is central to computer science, learning efficient programs has yet to be explored in program induction. In ILP, learning efficient programs has long been stated as an important topic [93] but it has been considered a difficult problem because there is no declarative difference between the answers computed by an efficient program, such as quicksort, and an inefficient program, such as bubble sort [94]. Instead, ILP systems typically rely on an Occamist bias to learn textually simple programs, such as those with the fewest literals [67] or clauses [97], and therefore ignore the efficiency of programs. For instance, Golem [92] and Progol [87] could both learn sorting algorithms from examples, but when given background knowledge suitable for learning quick sort, both systems learned variants of insertion sort because the program was smaller. We address this issue by introducing techniques to learn minimal cost programs. We use these techniques in Chapter 7 to learn efficient robot strategies and in Chapter 8 to learn efficient time complexity programs. We address this issue by introducing techniques to learn minimal cost programs, which we use to learn efficient robot strategies (Chapter 7) and learn efficient time complexity programs (Chapter 8).

2.8 Summary

In this chapter, we have outlined work related to the thesis. We have highlighted the differences between the various forms of automatic programming, where at one extreme you have universal inductive inference methods that only take examples as input and are general but impractical, and at the other you have deductive methods that take full logical specifications as input and are efficient but not general. This thesis is between the two in an area we call program induction, which takes examples and background knowledge as input. We have also discussed logic programming and ILP, including stating the merits of using ILP for program induction, especially using MIL which addresses long-standing challenges in ILP. In the next chapter, we formally describe relevant concepts from logic programming, ILP, and MIL.

Chapter 3

Meta-interpretive learning

In this chapter, we describe relevant concepts from logic programming and ILP. We also introduce MIL, on which this thesis is based, and Metagol, a MIL learner.

3.1 Logic programming

We start by stating relevant notation from logic programming. We refer the reader to [99] for a detailed overview of logic programming and ILP.

A variable is a string of characters starting with an uppercase letter. A function symbol is a string of characters starting with a lowercase letter. A predicate symbol is a string of characters starting with a lowercase letter. The arity n of a predicate or a function symbol p is the number of arguments it takes and is denoted as p/n. The set of predicate symbols with arity greater than 0 is called the predicate signature and is denoted as $\mathcal{P}$. A constant is a predicate symbol with arity zero. The set of constants symbols is called the constant signature and is denoted as $\mathcal{C}$. A variable is first-order if it can be substituted by a constant or function symbol. The set of first-order variables is denoted as $\mathcal{V}_1$. A term is a variable, a constant symbol, or a function symbol of arity n immediately followed by a bracketed n-tuple of terms. A term is ground if it contains no variables. The Herbrand universe is the set of all ground terms that can be formed with function and constant symbols and is denoted as $\mathcal{U}$. An atom is a formula $p(t_1, \ldots, t_n)$, where p is a predicate symbol of arity n and each t_i is a term. An atom is ground if all of its terms are ground. The set of all ground atoms that can be formed from $\mathcal{P}$ and $\mathcal{U}$ is the Herbrand

base and is denoted as $\mathcal{B}$. The negation symbol is $\neg$. A literal is an atom A or its negation $\neg A$. A finite (possibly empty) set of literals is a clause. A clause represents the disjunction of its literals. The variables in a clause are implicitly universally quantified. A clause is ground if it contains no variables. Simultaneously replacing variables $v_1, \ldots, v_n$ in a formula with terms $t_1, \ldots, t_n$ is called a substitution and is denoted as $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$. A substitution θ unifies atoms A and B in the case $A\theta = B\theta$. A clause C θ-subsumes a clause D whenever there exists a substitution θ such that $C\theta \subseteq D$.

In this thesis, we restrict ourselves to Horn clauses. A Horn clause is a clause with at most one positive literal. A definite clause is a Horn clause with exactly one positive literal:

Definition 4 (Definite clause) A (first-order) definite clause is of the form:

$$A_0 \leftarrow A_1, \ldots, A_m$$

where $m >= 0$ and each A_i is an atom of the form $p(t_1, \ldots, t_n)$, such that $p/n \in \mathcal{P}$ and $t_i \in \mathcal{C} \cup \mathcal{V}_1$. The atom A_0 is the head and the conjunction $A_1, \ldots, A_m$ is the body.

A fact is a definite clause with no body literals. A goal is a Horn clause with no head, i.e. no positive literal. A definite logic program is a set of definite clauses.

By restricting ourselves to definite programs, we lose expressive power but gain efficiency [99]. In particular, deduction based on SLD-resolution, which is outlined below, is refutation complete for Horn clauses [99]. In addition, we restrict ourselves to definite programs without function symbols, called datalog programs. Datalog programs are more expressive than relational databases but are also decidable [29]. Definite programs with function symbols have the expressive power of Turing machines and consequently are undecidable [132].

Higher-order logic programming

MIL uses higher-order logic. Higher-order logic extends first-order logic to allow for quantification over predicate and function symbols, and is therefore intrinsically more expressive than first-order logic [37]. In higher-order logic, a variable is higher-order if it can be substituted by a predicate symbol. The set of higher-order variables is denoted as $\mathcal{V}_2$. A higher-order term is a higher-order variable or a predicate symbol. An atom is higher-order if it has at least one higher-order term. A higher-order definite clause is a Horn clause with at least one higher-order atom:

Definition 5 (Higher-order definite clause) A higher-order definite clause is of the form:

$$A_0 \leftarrow A_1, \ldots, A_m$$

where $m >= 0$ and each A_i is an atom of the form $p(t_1, \ldots, t_n)$, such that $p/n \in \mathcal{P} \cup \mathcal{V}_2$ and $t_i \in \mathcal{C} \cup \mathcal{P} \cup \mathcal{V}_1 \cup \mathcal{V}_2$.

If a logic program contains a higher-order clause, then it is a higher-order logic program.

3.1.1 Computation

We focus on learning efficient programs, so we are concerned with the resources required by a program to compute an answer. Computation of a logic program starts with a goal and has one of two outcomes: success or failure. If the outcome is a success, then the final variables in the goal are included as part of the outcome. Because of the non-determinism of logic programs, a goal can have multiple successful outcomes. Computation of a definite program is based on SLD-resolution [63]. We state relevant concepts from SLD-resolution, taken from [99].

Definition 6 (SLD-resolvent) Let $G_0 = \leftarrow A_0, \ldots A_m$ be a goal and $C = B_0 \leftarrow B_1, \ldots, B_n$ be a Horn clause, such that the literal A_i is unifiable with B_0 with the substitution θ. Then the goal $G_1 = \leftarrow A_1, A_{i-1}, \ldots, B_1, \ldots, B_n, \ldots, A_{i+1}, \ldots, A_m$ is said to be derivable from G_0 and C with the substitution θ. The goal G_1 is called the SLD-resolvent.

SLD-resolution allows any for any selection rule to be used to select the literal in a goal. Prolog uses a selection rule that always selects the leftmost literal in a goal [99].

Definition 7 (SLD-derivation) Let P be a definite program and G_0 be an initial goal. Then a SLD-derivation of the goal G_n is a (possibly infinite) sequence of goals $G_0, \ldots, G_n$ such that every G_{i+1} is the SLD-resolvent of G_i with some clause in P.

Definition 8 (SLD-refutation) A SLD-refutation is a SLD-derivation of the empty clause.

There can be multiple SLD-derivations of a goal G_0 with respect to a definite program P because G_0 could be resolved with multiple Horn clauses in P. We represent all possible derivations as a SLD-tree:

Definition 9 (SLD-tree) Let P be a definite program and G_0 be an initial goal. Then a SLD-tree of $P \cup \{G_0\}$ is a (possible infinite) tree where each node is a goal and an edge between two nodes represents a SLD-derivation between them.

We are interested in branches of the tree that contain empty clauses as leaves:

Definition 10 (Successful branch) Let P be a definite program, G_0 be an initial goal, and T be a SLD-tree for $P \cup \{G_0\}$. Then a successful branch is a path between the root (G_0) and a leaf containing the empty clause.

A successful branch corresponds to a SLD-refutation of $P \cup \{G_0\}$, from which can obtain a computed answer for $P \cup \{G_0\}$:

Definition 11 (Computed answer) Let P be a definite program, G_0 be an initial goal, and $\theta_1, \ldots \theta_n$ be the sequence of mgu used in some SLD-refutation of $P \cup \{G_0\}$ (i.e. a successful branch). Then a computer answer θ for $P \cup \{G_0\}$ is the restriction of the composition of $\theta_1, \ldots \theta_n$ to the variables in G_0.

If a branch is not successful, then it is a failure:

Definition 12 (Failure branch) Let P be a definite program, G_0 be an initial goal, and T be a SLD-tree for $P \cup \{G_0\}$. Then a failure branch is a path between the root (G_0) and a leaf containing a non-empty goal.

In a failure branch, the non-empty goal is a leaf because no further derivation steps are possible from such a goal, and thus such a branch would not lead to a refutation.

The resources required to compute an answer for a definite program with a goal depends on (1) the size of the corresponding SLD-tree, and (2) the strategy used to search the tree. In Chapter 8, we introduce techniques to learn efficient programs by taking into account the size of SLD-trees.

3.2 Meta-interpretive learning

We now describe MIL, introduced in [96, 97] but extended in this thesis.

MIL is a form of ILP. The goal of an ILP system is to take as input background knowledge B and examples E and to return a hypothesis H that models the examples, where B, E, and H are all logic programs. The general ILP setting allows for B, E, and H to be any logical formulae. In this thesis, we restrict ourselves to definite programs without function symbols, i.e. datalog programs. This restriction is to ensure decidability of both the meta-interpreter and the learned programs, since the Herbrand base of a datalog program is always finite. In particular, we focus on learning from entailment where examples are facts, rather than general clauses, which is known as the example setting [93].

MIL is based on a Prolog meta-interpreter. The key difference between a MIL learner and a standard Prolog meta-interpreter is that whereas a standard Prolog meta-interpreter attempts to prove a goal by repeatedly fetching first-order clauses whose heads unify with a given goal, a MIL learner additionally attempts to prove a goal by fetching higher-order metarules (Figure 3.1), supplied as background knowledge, whose heads unify with the goal. The resulting meta-substitutions are saved and can be reused in later proofs. Following the proof of a set of goals, a definite program is formed by projecting the meta-substitutions onto their corresponding metarules, allowing for a form of ILP which supports predicate invention and learning recursive theories.

We now formally define the MIL setting. We first define metarules, which we have adapted from [96, 97]:

Definition 13 (Metarule) A metarule is a higher-order formula of the form:

$$\exists\pi\forall\mu\ A_0 \leftarrow A_1,\ldots,A_m$$

where $m >= 0$, π and μ are disjoint sets of higher-order variables, and each A_i is an atom of the form $p(t_1,\ldots,t_n)$ such that $p/n \in \mathcal{P} \cup \pi \cup \mu$ and each $t_i \in \mathcal{C} \cup \mathcal{P} \cup \pi \cup \mu$

The distinction between metarules and higher-order definite clauses is that, whereas the variables in a higher-order definite clause are all universally quantified, the variables in a metarule can be existentially quantified. Two commonly used metarules are the *identity* and *chain* metarules:

Example 3 (Quantified *identity* metarule)

$$\exists P\exists Q\forall A\forall B\ P(A,B) \leftarrow Q(A,B)$$

Example 4 (Quantified *chain* metarule)

$$\exists P \exists Q \exists R \forall A \forall B \forall C \; P(A,B) \leftarrow Q(A,C), R(C,B)$$

When describing metarules, we typically omit the quantifiers. Instead, we denote existentially quantified variables as uppercase letters starting from P and universally quantified variables as uppercase letters starting from A. Figure 3.1 shows the metarules used in this thesis.

Name	**Metarule**
Identity	$P(A,B) \leftarrow Q(A,B)$
Precon	$P(A,B) \leftarrow Q(A), R(A,B)$
Curry	$P(A,B) \leftarrow Q(A,B,R)$
Chain	$P(A,B) \leftarrow Q(A,C), R(C,B)$
Tailrec	$P(A,B) \leftarrow Q(A,C), P(C,B)$

Figure 3.1: Example metarules. The letters P, Q, and R denote existentially quantified variables. The letters A, B, and C denote universally quantified variables.

Choosing appropriate metarules is critical to the performance in MIL. For instance, in this thesis, we disallow overly general metarules, such as $P(A,B) \leftarrow$. By disallowing such overly general metarules, we can enforce a strict bias on the hypothesis space, which in turn allows us to, in some cases, learn programs using positive examples only (Section 7.5). Deciding which metarules to use is the focus of Chapter 4.

We now define the MIL input, which is similar to a standard ILP input [99] but additionally takes a set of metarules as background knowledge. As is standard in ILP [107], we assume a language of examples $\mathcal{E}$, background knowledge $\mathcal{B}$, and hypotheses $\mathcal{H}$.

Definition 14 (MIL input) The MIL input is a tuple (B, E) where:

- $B \subseteq \mathcal{B}$ and $B = B_C \cup M$ where B_C is compiled definite program background knowledge and M is a set of metarules
- $E = (E^+, E^-)$ is a tuple where $E^+ \subseteq \mathcal{E}$ and $E^- \subseteq \mathcal{E}$ are sets of ground facts representing positive and negative examples respectively

In contrast to [96, 97], our MIL input definition treats metarules as part of the background knowledge. In the above definition, we use the term *compiled* definite program background knowledge. We explain the reason for using this term in Chapter 5 when we introduce non-compiled background knowledge. We define a consistent hypothesis:

Definition 15 (Consistent hypothesis) Let (B, E) be a MIL input. Then a definite program hypothesis $H \in \mathcal{H}$ is consistent if and only if $B \cup H \models E^+$ and $B \cup H \not\models E^-$.

For convenience, we define the version space [81], which contains only hypotheses consistent with the examples:

Definition 16 (Version space) Let (B, E) be a MIL input. Then the version space $\mathcal{VS}_{B,E}$ is the subset of consistent hypotheses from $\mathcal{H}$.

We now define a MIL learner:

Definition 17 (MIL learner) A MIL learner takes an input (B, E) and outputs a definite program $H \in \mathcal{VS}_{B,E}$.

A MIL learner uses metarules to build a definite program by searching for a proof of a set of goals. A proof is based on a sequence of meta-substitutions:

Definition 18 (Meta-substitution) Let M be a metarule with the name x, C be a horn clause, θ be a unifying substitution of M and C, and $\Sigma \subseteq \theta$ be the substitutions where the variables are all existentially quantified in M, such that $\Sigma = \{v_1/t_1, \ldots, v_n/t_n\}$. Then a meta-substitution for M and C is an atom of the form:

$$sub(x, [v_1/t_1, \ldots, v_n/t_n])$$

To illustrate meta-substitutions, suppose a MIL learner is given background predicates *reverse/2* and *head/2*, the *chain* metarule (Figure 3.1), and the goal:

$$last([a, l, g, o, r, i, t, h, m], m) \leftarrow$$

Given this input, a MIL learner can perform the meta-substitution:

$$sub(chain, [P/last, Q/reverse, R/head])$$

This meta-substitution is then projected onto the corresponding metarule to derive the program:

last(A,B) ← reverse(A,C), head(C,B)

MIL supports *inventions*:

Definition 19 (Invention) Let (B, E) be a MIL input and $H \in \mathcal{VS}_{B,E}$ be an output hypothesis. Then a predicate p/a is an *invention* if and only if it is in the predicate signature of H and not in the predicate signature of $B \cup E$.

Metagol, described later in this chapter, supports inventions by adding new predicate symbols to the predicate signature.

Language classes, expressivity, and complexity

Throughout this thesis, we refer to the textual complexity of a logic program:

Definition 20 (Textual complexity) A textual complexity function is of the form:

$$\tau : \mathcal{H} \rightarrow \mathbf{N}$$

In this thesis, we typically measure the textual complexity of a program H as the number of clauses in H:

Definition 21 (Clause complexity) The clause complexity $\tau_c(H)$ of the program H is the number of clauses in H.

We focus on restricted classes of logic programs:

Definition 22 (H^a_m clause) Let a and m be natural numbers and C be a Horn clause. Then C in the class H^a_m if and only if it contains at most m literals in the body and each literal has arity at most a.

We mainly focus on the class H^2_2:

Definition 23 (H^2_2 class) A Horn clause is in the class H^2_2 if and only if it contains at most 2 literals in the body and each literal has arity at most 2.

The class of H^2_2 programs with one function symbol has UTM expressivity [132]. We restate a result from [72] concerning the number of programs that can be constructed in this class:

Theorem 1 (Number of programs in H^2_2) Given p predicate symbols and m metarules, the number of H^2_2 programs expressible with n clauses is $O(m^n p^{3n})$.

Proof 1 The number of clauses which can be constructed from a H^2_2 metarule given p predicate symbols is at most p^3. Therefore the set of such clauses $S_{m,p}$ which can be formed from m distinct H^2_2 metarules using p predicate symbols has cardinality at most mp^3. It follows that the number of programs which can be formed from a selection of n rules chosen from $S_{m,p}$ is at most $(mp^3)^n = O(m^n p^{3n})$.

In the first part of this thesis, we introduce techniques to efficiently learn programs by reducing the hypothesis space. Our results are based on the Blumer bound [9], which states that given a hypothesis space of size $|H|$, the number of examples m required for a concept to be PAC-learnable is as follows:

$$m \geq \frac{1}{\epsilon}(ln|H| + ln\frac{1}{\delta})$$

3.3 Metagol

Metagol [26] is a MIL learner. Introduced in [96], we have extended it in this thesis. The Prolog code[1] for Metagol is outlined below:

```
learn(Pos,Neg,Prog):-
    prove(Pos,[],Prog),
    not(prove(Neg,Prog,Prog)).
prove([],Prog,Prog).
prove([Atom|Atoms],Prog1,Prog2):-
    prove_aux(Atom,Prog1,Prog3),
    prove(Atoms,Prog3,Prog2).
prove_aux(Atom,Prog,Prog):-
    prim(Atom),!,
    call(Atom).
prove_aux(Atom,Prog1,Prog2):-
    member(sub(Name,Subs),Prog1),
    metarule(Name,Subs,(Atom:-Body)),
    prove(Body,Prog1,Prog2).
```

[1] Available at https://github.com/metagol/metagol

```
prove_aux(Atom,Prog1,Prog2):-
    metarule(Name,Subs,(Atom:-Body)),
    prove(Body,[sub(Name,Subs)|Prog1],Prog2).
```

Metagol works as follows. Given sets of facts representing positive examples, Metagol treats each fact as a goal and tries to prove each goal (`Atom`) in turn. Metagol first tries to prove a goal deductively using background knowledge by delegating the proof to Prolog (`call(Atom)`). Failing this, Metagol tries to unify the goal with the head of a metarule (`metarule(Name,Subs,(Atom :- Body))`) and to bind the existentially quantified variables in a metarule to symbols in the signature. Metagol saves the resulting meta-substitution and tries to prove the body of the metarule. After proving all goals, a Prolog program is formed by projecting the meta-substitutions onto their corresponding metarules. Metagol checks the consistency of the learned program with the negative examples. If the program is inconsistent, then Metagol backtracks to explore different branches of the SLD-tree.

Metagol uses iterative deepening to ensure that the first consistent hypothesis returned has the minimal number of clauses. The search starts at depth 1. At depth d the search returns a consistent hypothesis with at most d clauses if one exists. Otherwise it continues to depth $d+1$. At each depth d, Metagol introduces $d-1$ new predicate symbols. New predicates symbols are formed by taking the name of the task and adding underscores and numbers. For example, if the task is f and the depth is 4 then Metagol will add the predicate symbols f_3, f_2, and f_1 to the predicate signature.

Example 5 To illustrate Metagol, suppose you have the following background knowledge:

```
mother(ann,amy).
mother(ann,andy).
mother(amy,amelia).
mother(amy,bob).
mother(linda,gavin).
father(steve,amy).
father(steve,andy).
father(gavin,amelia).
```

And the following metarules (written in Prolog):

```
metarule(ident,[P,Q],([P,A,B]:-[[Q,A,B]])).
metarule(chain,[P,Q,R],([P,A,B]:-[[Q,A,C],[R,C,B]])).
```

Then you can call Metagol with a lists of positive (Pos) and negative (Neg) examples:

```
  Pos = [
    grandparent(ann,amelia),
    grandparent(steve,amelia),
    grandparent(steve,spongebob),
    grandparent(linda,amelia)
  ],
  Neg = [grandparent(amy,amelia)],
  learn(Pos,Neg,Prog),
  pprint(Prog).
```

The result of this call is an answer substitution for the variable `Prog`. This answer substitution is a list of meta-substitutions:

```
[
    sub(ident,grandparent_1,2,[grandparent_1,father]),
    sub(ident,grandparent_1,2,[grandparent_1,mother),
   sub(chain,grandparent,2,[grandparent,grandparent_1,grandparent_1]
]
```

These meta-substitutions are then projected onto the corresponding metarules to form the program:

```
grandparent(A,B):-grandparent_1(A,C),grandparent_1(C,B).
grandparent_1(A,B):-father(A,B).
grandparent_1(A,B):-mother(A,B).
```

Chapter 4 uses this version of Metagol. In Chapter 5, we extend this version to support learning higher-order programs. In all future cases, including the experiments, we call Metagol (or renamed variants) in the same way as this example.

3.4 Summary

In this chapter, we have described relevant concepts from logic programming, ILP, and MIL, used in the rest of this thesis. One contribution of this chapter is our definition of the MIL input (Definition 14) which, in contrast to previous work on MIL, considers metarules to be part of the background knowledge. This change is important because one of the claims of this thesis is that we can use appropriate background knowledge to improve learning efficiency, including using appropriate metarules, which the following chapter explores. We have also outlined concepts relating to the computation of logic programs, which are used in Chapter 8 to learn efficient time complexity logic programs.

Chapter 4

Logical minimisation of metarules

In this chapter, we support Claim 1 of this thesis by using appropriate metarules to improve the learning performance of a MIL learner. In particular, we use Plotkin's clausal theory reduction algorithm to find logically minimal sets of metarules. In our experiments, we compare learning with minimal and maximal set of metarules. In general, learning with minimal sets of metarules leads to lower runtimes and higher predictive accuracies than larger sets.

4.1 Introduction

A MIL learner takes as input background knowledge formed of first-order definite clauses and higher-order metarules. The metarules guide the search for a proof of a set of goals, where a proof is based on a sequence of meta-substitutions. The metarules determine the structure of learnable programs, which in turn defines the hypothesis space. Selecting which metarules to use is a trade-off between efficiency and expressivity: the hypothesis space increases given more metarules (Theorem 1), so we wish to use fewer metarules, but if we use too few metarules then we lose expressivity. For example, consider the goal:

$$G = \leftarrow last([a,l,g,o,r,i,t,h,m],m)$$

And also consider the metarules:

$$M_1 = P(A,B) \leftarrow Q(A,B)$$
$$M_2 = P(A,B) \leftarrow Q(B,A)$$
$$M_3 = P(A,B) \leftarrow Q(A,C), R(C,D), S(D,B)$$
$$M_4 = P(A,B) \leftarrow Q(A,C), R(C,B)$$

Given G, these metarules, and the background predicates *reverse/2* and *head/2*, a MIL learner can perform the meta-substitution $sub(M_4, [P/last, Q/reverse, R/head$ to resolve G with M_4 to derive the clause:

$$\text{last(A,B)} \leftarrow \text{reverse(A,C)}, \text{head(C,B)}$$

In the worst-case, a MIL learner would have to first try all of the other metarules before trying M_4, which is inefficient. As shown in Theorem 1, for the H^2_2 fragment of logic programs, the number of programs of size n which can be built from p predicate symbols and m metarules is $O(m^n p^{3n})$. This result implies that we can improve the efficiency of a MIL learner by reducing the number of metarules. In this chapter, we introduce techniques to minimise the number of metarules without loosing expressivity by employing logical reduction techniques. To illustrate this idea, consider these two metarules:

$$M_5 = P(A,B) \leftarrow Q(A,B)$$
$$M_6 = P(A,B) \leftarrow Q(A,B), R(A,B)$$

The metarule M_5 subsumes M_6 and therefore M_6 is logically redundant. In Section 4.4.2, we show that only two metarules are necessary for a restricted fragment of H^2_m.

4.2 Related work

Using metarules to build a logic program is similar to the use of refinement operators in ILP [119, 99] to build a definite clause literal-by-literal[1]. As with refinement operators, it seems reasonable to ask about completeness and irredundancy of a set of metarules. This question, which has not been addressed in previous papers on MIL or second-order schemata [109, 35], is investigated in this chapter.

[1]Note that MIL uses example driven test-incorporation for finding consistent programs as opposed to the generate-and-test approach of clause refinement.

Unlike refinement operators, metarules are a form of declarative bias [108]. In contrast to other forms of declarative bias in ILP, such as modes [87, 127] or grammars [20], metarules are logical statements. We can therefore reason about them alongside normal first-order background knowledge, which is explored in the following section.

4.3 Metarules and encapsulation

We use Plotkin's reduction algorithm [103] to logically reduce sets of metarules. However, metarules are higher-order formulas with existentially quantified variables and are therefore incompatible with Plotkin's algorithm, which takes first-order clauses as input. Therefore, we introduce *encapsulation*, a method to transform metarules into first-order clauses. In the rest of this chapter, we assume that the variables in a metarule are all universally quantified.

We first define encapsulation for atoms:

Definition 24 (Atomic encapsulation) Let A be higher-order or first-order atom of the form $p(t_1, .., t_n)$. Then $enc(A) = m(p, t_1, .., t_n)$ is an encapsulation of A.

We extend atomic encapsulation to logic programs:

Definition 25 (Program encapsulation) The logic program $enc(P)$ is an encapsulation of the logic program P in the case $enc(P)$ is formed by replacing all atoms A in P by $enc(A)$.

Unencapsulated	**Encapsulated**
$P(A,B) \leftarrow Q(B,A)$	$m(P,A,B) \leftarrow m(Q,B,A)$
$last(A,B) \leftarrow reverse(A,C), head(C,B)$	$m(last,A,B) \leftarrow m(reverse,A,C), m(head,C,B)$

Figure 4.1: Two examples of encapsulation.

Figure 4.1 shows examples of the encapsulation of metarules and first-order clauses. We extend encapsulation to interpretations of logic programs:

Definition 26 (Interpretation encapsulation) Let I be an interpretation over the predicate and constant symbols in a logic program. Then the encapsulated interpretation $enc(I)$ is formed by replacing each atom A in I by $enc(A)$.

We now have the proposition:

Proposition 1 (Encapsulation models) The logic program P has a model M if and only if $enc(P)$ has the model $enc(M)$.

Proof 2 Follows trivially from the definitions of encapsulated programs and interpretations.

We can now define entailment between logic programs:

Proposition 2 (Entailment) Let P and Q be logic programs. Then $P \models Q$ if and only if every model $enc(M)$ of $enc(P)$ is also a model of $enc(Q)$.

Proof 3 Follows immediately from Proposition 1.

4.4 Logically reducing metarules

Plotkin [103] provides the following definitions as the basis for eliminating logically redundant clauses from a clausal theory:

Definition 27 (Clause redundancy) The clause C is logically redundant in the clausal theory $P \cup \{C\}$ whenever $P \models C$.

If C is redundant in $P \cup \{C\}$ then P is logically equivalent to $P \cup \{C\}$ because $P \models P \cup \{C\}$ and $P \cup \{C\} \models P$. Plotkin defines a reduced clausal theory as:

Definition 28 (Reduced clausal theory) A clausal theory is reduced if and only if it does not contain any redundant clauses.

Plotkin uses these definitions to define an algorithm which given a clausal theory repeatedly identifies and removes redundant clauses until the resulting clausal theory is reduced.

4.4.1 Reduction of metarules in H_2^{2*}

We now use Plotkin's reduction algorithm to reduce clausal theories formed of encapsulated metarules. We focus on a subclass of H_m^2. Exploring broader classes of metarules is left for future work (Section 9.2.1).

As is standard ILP [1, 109, 19], we restrict ourselves to *connected clauses* (i.e. we exclude unconnected clauses), where we want to ensure that the literals in the body of a clause are connected to the head via the clause's variables:

Definition 29 (Connected clause) A clause is connected if the literals in the clause cannot be partitioned into two sets such that the variables appearing in the literals of one set are disjoint from the variables appearing in the literals of the other set.

The point of this restriction is to ignore metarules from which we could derive clauses that are unlikely to be useful in a hypothesis. For instance, the clause $P(A) \leftarrow R(B)$ is unconnected because we can form two disjoint sets $\{A\}$ and $\{B\}$ which contain no intersecting variables. Because R(B) is not connected to the head, it will be true for every monadic fact and is essentially useless. By contrast, the clause $P(A) \leftarrow Q(A)$ is connected because the variables in the literals $P(A)$ and $Q(A)$ cannot be partitioned into two disjoint sets. This clause is more useful because the body is linked to the head.

We add a further chained clause restriction:

Definition 30 (Chained clause) A clause C is *chained* if and only if (1) C is connected, (2) each literal in C is dyadic, and (3) each term variable in C appears at least twice.

We focus on H_m^{2*}, a subclass of H_m^2:

Definition 31 (H_m^{2*} fragment) A metarule is in H_m^{2*} if and only if it is in H_m^2, and it is chained.

The *identity* metarule (Figure 3.1) is in H_m^{2*}:

Example 6 (Metarule in H_m^{2*}) The *identity* metarule $P(A,B) \leftarrow Q(A,B)$ is in H_m^{2*} because it is in H_2^2 and is chained Definition 31.

By contrast, the *curry* metarule (Figure 3.1) is not in H_m^{2*}:

Encapsulated metarules	Reduced set
m(P,A,B) ← m(Q,A,B)	
m(P,A,B) ← m(Q,B,A)	m(P,A,B) ← m(Q,B,A)
m(P,A,B) ← m(Q,A,B), m(R,A,B)	
m(P,A,B) ← m(Q,A,B), m(R,B,A)	
m(P,A,B) ← m(Q,A,C), m(R,B,C)	
m(P,A,B) ← m(Q,A,C), m(R,C,B)	m(P,A,B) ← m(Q,A,C), m(R,C,B)
m(P,A,B) ← m(Q,B,A), m(R,A,B)	
m(P,A,B) ← m(Q,B,A), m(R,B,A)	
m(P,A,B) ← m(Q,B,C), m(R,A,C)	
m(P,A,B) ← m(Q,B,C), m(R,C,A)	
m(P,A,B) ← m(Q,C,A), m(R,B,C)	
m(P,A,B) ← m(Q,C,A), m(R,C,B)	
m(P,A,B) ← m(Q,C,B), m(R,A,C)	
m(P,A,B) ← m(Q,C,B), m(R,C,A)	

Figure 4.2: Encapsulation of all 14 metarules in H_2^{2*} leading to a reduced set of two.

Example 7 (Metarule not in H_m^{2*}) The *curry* metarule $P(A,B) \leftarrow Q(A,B,C)$ is not in H_m^{2*} because the literal Q(A,B,C) is not dyadic and the variable C appears only once.

To find logically minimal sets of metarules, we generated a set of all the metarules in H_2^{2*}, for which there were 14. We ran Plotkin's reduction algorithm on an encapsulation of this set, which resulted in a minimal set of two metarules, shown in Figure 4.2. Since, by construction, this set is logically equivalent to the complete set of 14, this minimal set can be considered a universal set (sufficient to generate all hypotheses) for H_2^{2*}. We repeated this procedure for H_3^{2*} and H_4^{2*}, with cardinalities 226 and 5346 respectively, each time deriving the same minimal set of two metarules. In the following section, we prove that these two metarules are complete for H_m^{2*}.

4.4.2 Completeness theorem for H_m^{2*}

We now show that two metarules are sufficient to entail all metarules in H_m^{2*}. We first name the two elementary metarules:

Definition 32 (Inverse rule) The *inverse* rule is of the form:

$$P(A,B) \leftarrow Q(B,A)$$

Definition 33 (H_2^2 chain rule) The H_2^2 *chain* rule is of the form:

$$P(A,B) \leftarrow Q(A,C),R(C,B)$$

We first show that the *inverse* rule can reposition the variables in a literal:

Lemma 1 (Inverse rule application) Let C be a clause in H_m^{2*}, $m > 0$, and L be a literal in the body of C. Then resolving L with the head of the *inverse* rule gives the resolvent C' with the literal L' where the variables in L' are reversed.

Proof 4 Trivial by construction.

We now show that the *inverse* and H_2^2 *chain* metarules are sufficient to entail every metarule in H_m^{2*}:

Theorem 2 (Completeness theorem for H_m^{2*}) Let C_1 be the *inverse* rule, C_2 be the H_2^2 *chain* rule, S_m be the set of all metarules in H_m^{2*}, and $m > 0$. Then $\{C_1, C_2\} \models R$ for every R in S_m.

Proof 5 Assume the opposite. Thus, there is a metarule R such that $\{C1, C2\} \not\models R$. By lemma 1, we do not need to consider the order of the variables in a literal, so, without loss of generality, let $R = P(A,B) \leftarrow T_1 \ldots T_m$. Because of the definition of chained clauses, the variables A and B must also appear in a body literal of R. There are two cases:

Case 1 Suppose A and B appear in the same body literal, thus:

$$R = P(A,B) \leftarrow \ldots, T_i(B,A), \ldots$$

Clearly, $C1 \models R$, so this case cannot hold.

Case 2 Suppose A and B appear in separate body literals, thus:

$$R = P(A,B) \leftarrow \ldots, T_i(A,\_), T_j(\_,B) \ldots$$

Because A and B appear in separate body literals, the second variable in T_i must be C and C must also appear in another body literal. Suppose C appears in T_j, thus:

$$R = P(A,B) \leftarrow \ldots, T_i(A,C), T_j(C,B) \ldots$$

Clearly, $C2 \models R$, so C must appear in another body literal, thus:

$$R = P(A,B) \leftarrow \ldots, T_i(A,C), T_j(\_,B), T_k(\_,C) \ldots$$

Because A and B do not appear together in T_j and nor do C and B, the literal T_j must contain a new variable:

$$R = P(A,B) \leftarrow \ldots, T_i(A,C), T_j(D,B), T_k(\_,C) \ldots$$

In this case, we can substitute D for C to form:

$$R = P(A,B) \leftarrow \ldots, T_i(A,C), T_j(C,B), T_k(\_,C) \ldots$$

Clearly, $C2 \models R$, so this case cannot hold.

Because these two cases are exhaustive and both contradict the assumption, the proof is complete.

4.4.3 Representing H_m^{2*} programs in H_2^{2*}

We now show that H_2^{2*} is a normal form for H_m^{2*}:

Theorem 3 (H_2^{2*} is a normal form for H_m^{2*}.) Let C be a metarule in H_m^{2*} and $m > 2$. Then there is an equivalent[2] theory in H_2^{2*}.

Proof 6 We prove by construction. Let C be of the form $P \leftarrow T_1, \ldots, T_m$. For any literal T_i in the body of C of the form $T(U_{i+1}, U_i)$ introduce a clause $s_i(A,B) \leftarrow T_i(B,A)$ and replace $T_i(U_{i+1}, U_i)$ in C with $s_i(U_i, U_{i+1})$. Step 2. Introduce clauses $(P(A,B) \leftarrow T_1(A,C), p_1(C,B)), (p_1(A,B) \leftarrow T_2(A,C), p_2(C,B)), \ldots, (p_{m-2}(A,B) \leftarrow t_{m-1}(A,C), t_m(C,B))$. Step 3. Remove C from the theory. You now have an equivalent theory in H_2^{2*}.

This theorem is exemplified below:

[2]The two theories are not logically equivalent because of the introduction of new predicate symbols. However, the theories are success set equivalent when restricted to the target predicate, or when restricted to the predicates of the original theory.

Example 8 Let $C = P(U_1, V) \leftarrow T_1(U_1, U_2), T_2(U_3, U_2), T_3(U_3, U_4), T_4(U_4, V)$. Notice that the variables in T_2 are in the order (U_{i+1}, U_i) and not (U_i, U_{i+1}). We now construct a theory in H_2^2 equivalent to C.

- **Step 1**. Introduce the clause $s_1(A,B) \leftarrow T_2(B,A)$ and replace T_2 in C with $s_1(U_2, U_3)$ to form:

$$C' = P(U_1, V) \leftarrow T_1(U_1, U_2), s_1(U_2, U_3), T_3(U_3, U_4), T_4(U_4, V)$$

- **Step 2**. Introduce the clause $p_1(A,B) \leftarrow T_3(A,C), T_4(C,B)$ and replace the literals $T_3(U_3, U_4)$ and $T_4(U_4, V)$ in C' with $p_1(U_3, V)$ to form:

$$C'' = P(U_1, V) \leftarrow T_1(U_1, U_2), s_1(U_2, U_3), p_2(U_3, V)$$

  Introduce the clause $p_2(A,B) \leftarrow s_1(A,C), p_1(C,B)$ and replace the literals $s_1(U_2, U_3)$ and $p_1(U_3, V)$ in C'' with $p_2(U_2, V)$ to derive C''':

$$C''' = P(U_1, V) \leftarrow T_1(U_1, U_2), p_2(U_2, V)$$

- **Step 3**. After removing C, C', C'', and C'' and renaming variables you are left with the following theory, which is equivalent to C:

$$P(A,B) \leftarrow T_1(A,C), p_2(C,B)$$
$$p_2(A,B) \leftarrow s_1(A,C), p_1(C,B)$$
$$p_1(A,B) \leftarrow T_3(A,C), T_4(C,B)$$
$$s_1(A,B) \leftarrow T_2(B,A)$$

4.5 Experiments

We now describe experiments which compare learning with minimal and non-minimal sets of metarules. We test the null hypotheses:

Null Hypothesis 1 Using fewer metarules cannot improve predictive accuracies

Null Hypothesis 2 Using fewer metarules cannot reduce learning times

To test these null hypotheses, we conduct experiments in which we vary the metarules supplied to Metagol. We compare four sets of metarules:

- **Min** the minimal set of metarules described in Section 4.4.2
- $\mathbf{H}_2^{2*}$ all the metarules in the class H_2^{2*}
- $\mathbf{H}_3^{2*}$ all the metarules in the class H_3^{2*}
- $\mathbf{H}_4^{2*}$ all the metarules in the class H_4^{2*}

4.5.1 Learning kinship relations

In this experiment, we learn kinship relations using the dataset in [50][3], which contains 12 dyadic relations: *aunt, brother, daughter, father, husband, mother, nephew, niece, sister, son,* and *wife*, and 104 examples.

Experiment 1: number of training examples

We first compare predictive accuracies and learning times when varying the number of training examples.

Materials We include all relations excluding the target relation as background knowledge.

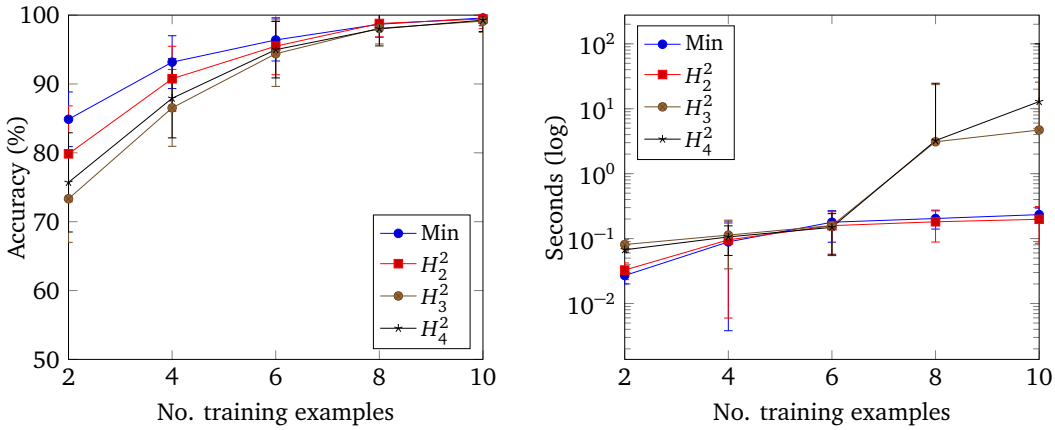
Methods For each m in the set {2,4,6,8,10}, we train using m randomly chosen examples of each relation, half positive and half negative. We test using 5 positive and 5 negative examples, so the default accuracy is 50%. We measure mean predictive accuracies and learning times over all relations over 50 trials. We limit the search to programs of length 6 and enforce a 5-minute timeout.

Results Figure 4.3 shows that using fewer metarules improves predictive accuracies and reduces learning times, rejecting null hypotheses 1 and 2. This result can be explained by the larger hypothesis space searched when using more metarules which, according to the Blumer bound [9], results in higher predictive errors and longer learning times.

Experiment 2: sampled background relations

This experiment compares the effect of using partial background knowledge.

[3] https://archive.ics.uci.edu/ml/datasets/Kinship



(a) Predictive accuracies (b) Learning times

Figure 4.3: Experiment 1 results which show learning performance when varying the number of training examples.

Methods We train using 4 randomly chosen examples of each relation, half positive and half negative. We test using 5 positive and 5 negative examples, so the default accuracy is 50%. For each p in the set $\{1, 2, 3, \ldots, 10\}$, we randomly select p relations to be included as background knowledge, ensuring that the target relation is not included. We measure mean predictive accuracies and learning times over all relations over 30 trials. We limit the search to programs of length 6 and enforce a 5-minute timeout.

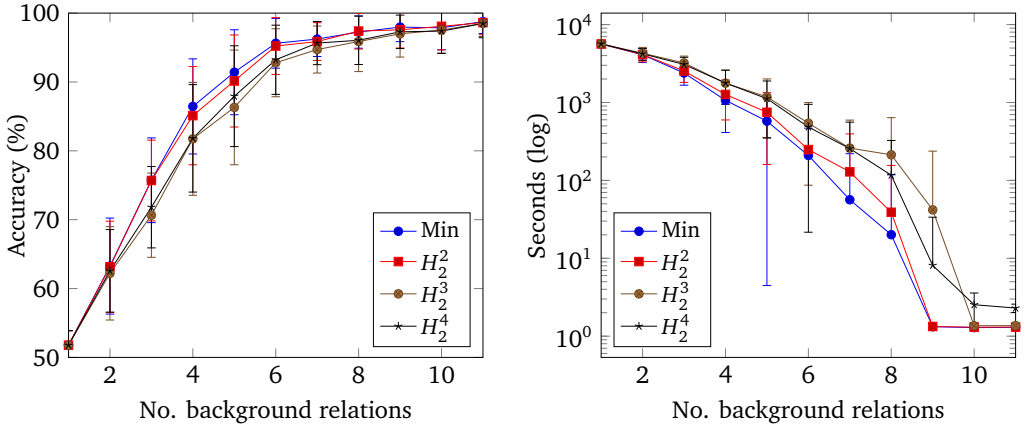
Results Figure 4.4 shows that using fewer metarules improves predictive accuracy and reduces learning times, again rejecting null hypotheses 1 and 2.

Experiment 3: sampling metarules

This experiment compares the effect of sampling metarules from a larger set.

Materials We include all relations excluding the target relation as background knowledge.

Methods We train using 8 randomly chosen examples of each relation, half positive and half negative. We test using 5 positive and 5 negative examples, so the default accuracy is 50%. For each m in the set $\{2, 3, 4, \ldots, 14\}$, we randomly



(a) Predictive accuracies (b) Learning times

Figure 4.4: Experiment 2 results which show learning performance when varying the number of background relations.

selected m metarules from the full enumeration of H_2^{2*} to use. We measure mean predictive accuracies and learning times over all relations over 30 trials. We limit the search to programs of length 6 and enforce a 5-minute timeout.

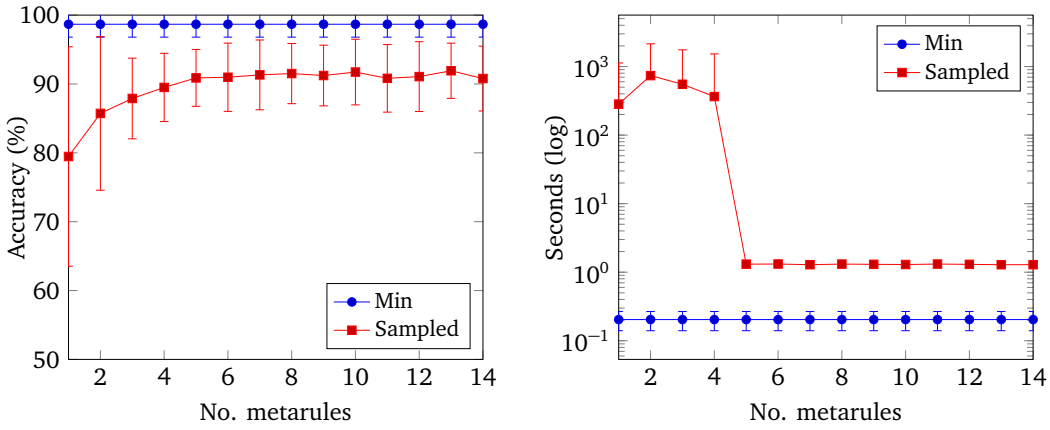
Results Figure 4.5 shows the predictive accuracies and learning times when sampling the number of metarules. The corresponding results when learning with the minimal set of metarules (*Min*) from experiment 1 are provided for comparison. These results show that using the *Min* set of metarules outperforms sampling metarules in terms of predictive accuracy and learning times.

4.5.2 Learning robot plans

We now explore varying the metarules when learning robot plans. Imagine a robot in a two-dimensional space which can perform six dyadic actions: *move_left/2, move_right/2, move_forwards/2, move_backwards/2, grab_ball/2* and *drop_ball/2*. We represent the robot's state as a Prolog list with three elements:

```
[RobotPos,BallPos,HasBall]
```

`RobotPos` and `BallPos` are coordinates and `HasBall` is a boolean representing



(a) Predictive accuracies (b) Learning times

Figure 4.5: Experiment 3 results which show learning performance when sampling the number of metarules.

whether the robot has the ball. The robot's task is to move the ball to a destination. Suppose we have the example:

```
move_ball([0/0,0/0,false],[2/2,2/2,false]).
```

Where the robot and ball start at (0,0) and end at (2,2). Given this example and the minimum set of metarules, Metagol learns program:

```
move(A,B):-move_3(A,C),move_3(C,B).
move_3(A,B):-move_2(A,C),drop(C,B).
move_2(A,B):-grab(A,C),move_1(C,B).
move_1(A,B):-move_forwards(A,C),move_right(C,B).
```

In this program, the robot grabs the ball, moves right, moves forward, drops the ball, and then repeats this process. Metagol also learns a program where the robot performs the *grab_ball/2* and *drop_ball/2* actions only once:

```
move(A,B):-move_3(A,C),drop_ball(C,B).
move_3(A,B):-grab_ball(A,C),move_2(C,B).
move_2(A,B):-move_1(A,C),move_1(C,B).
move_1(A,B):-move_forwards(A,C),move_right(C,B).
```

However, both programs are considered equal because they contain the same number of clauses. Should we wish to prefer programs which minimise the number of grabs and drops, then we would need to associate costs with different operations. This idea of learning programs with minimal costs is explored in the second part of this thesis (Chapters 6, 7, and 8).

Now suppose we exclude *move_right/2* from the background knowledge. Then given the original example, Metagol learns the program:

```
move(A,B):-move_3(B,A)
move_3(A,B):-move_2(A,C),move_2(C,B)
move_2(A,B):-move_1(A,C),move_1(C,B)
move_1(A,B):-grab_ball(A,C),move_left(C,B)
move_1(A,B):-move_backwards(A,C),drop_ball(C,B)
```

Metagol found this program using the *inverse* metarule by backtracking from the goal, replacing *move_right/2* with *move_left/2* and *move_forwards/2* with *move_backwards/2*. If we also remove *move_forwards/2* and *drop_ball/2* from the background knowledge, Metagol learns a program with only five clauses and three primitives, compared to the original four clause program which used six primitives. The construction of an inverse plan is familiar to retrograde analysis of positions in chess [133], in which you go backwards from an end position to work out the moves necessary to get there from a given starting position. This idea of purposely removing background predicates is similar to dimensionality reduction, widely used in other forms of machine learning [122], but which has been under used in ILP [41]. Our preliminary experiments [22] seem to indicate that this reduction is possible, and future work will investigate this idea in more detail.

We compared learning robot plans using different sets of metarules but when learning with non-minimal sets, the learning times were prohibitively slow in all cases, whereas using the minimal set we were able to learn programs.

4.6 Future work

We have shown that there exist minimal sets of metarules for class H^{2*}_m. In future work, we intend to extended the approach to broader classes of metarules, such as those containing monadic and triadic predicates. The ability to encapsulate background knowledge suggests that it may be possible to minimise

the metarules together with a given set of background clauses. Preliminary experiments indicate that this is possible, and we aim to develop this idea in future work.

We have explored sampling metarules from the maximum set. The results suggest there is no benefit in using more metarules than the minimum set. However, this is not always the case. For example, when learning dyadic string transformations [72], the programs only used the *chain* metarule. Thus, in this case, the optimal set of metarules is a subset of the minimal set described in this chapter. In future work, we would like to investigate learning the optimal set of metarules as to minimise the hypothesis space.

4.7 Summary

In this chapter, we have shown that in some cases as few as two metarules are complete and sufficient for generating all hypotheses for a fragment of logic, as in the case of H_m^{2*}. Our experiments show that using fewer metarules achieves higher predictive accuracies and lower learning times than using larger sets of metarules. This result supports Claim 1 of this thesis, i.e. we can efficiently learn programs. In the next chapter, we further support Claim 1 by using higher-order background knowledge to improve learning efficiency.

Chapter 5

Learning higher-order programs

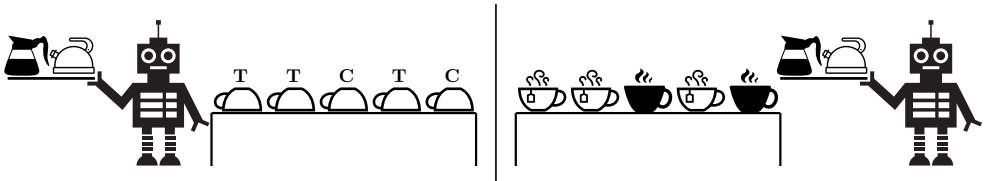
A key feature of ILP is its ability to learn first-order programs which are more expressive than propositional programs [86]. In this chapter, we introduce Metagol$_{AI}$, a MIL learner which extends ILP to support learning of higher-order programs. One may assume that increasing the expressivity of a learner would decrease efficiency. However, we show the opposite: that learning higher-order programs, rather than first-order programs, can improve predictive accuracies and reduce learning times. Specifically, the ability to learn higher-order programs reduces the textual complexity required to express programs which in turn reduces the hypothesis space. Our sample complexity results support Claim 1 of the thesis and show that the approach reduces (1) the number of examples required to reach high predictive accuracy, and (2) learning times.

5.1 Introduction

Suppose you are teaching a robot to pour tea and coffee for all place settings at a table, where each setting has an indication of whether the guest prefers tea or coffee. Figure 5.1 shows an example in terms of initial and final states. Now consider learning a general strategy for the task from a set of examples. Given that there may be an arbitrary number of place settings, existing program induction approaches, such as MIL used in [96, 97, 72], would learn a first-order recursive program, such as:

Example 9 (First-order waiter program)

```
waiter(A,B):-waiter_3(A,B),at_end(B).
```



(a) Initial state (b) Final state

Figure 5.1: Figures (a) and (b) show initial/final state waiter examples respectively. In the initial state, the cups are empty and each guest has a preference for tea **(T)** or coffee **(C)**. In the final state, the cups are facing up and are full with the guest's preferred drink.

```
waiter(A,B):-waiter_3(A,C),waiter(C,B).
waiter_3(A,B):-waiter_2(A,C),move_right(C,B).
waiter_2(A,B):-turn_cup_over(A,C),waiter_1(C,B).
waiter_1(A,B):-wants_tea(A),pour_tea(A,B).
waiter_1(A,B):-wants_coffee(A),pour_coffee(A,B).
```

In this chapter, we extend MIL to support learning higher-order programs, which include higher-order constructs such as *map/3*, *until/4*, and *ifthenelse/5*. Using this approach, we learn an equivalent but more compact strategy, such as:

Example 10 (Higher-order waiter program)

```
waiter(A,B):-until(A,B,at_end,waiter_3).
waiter_3(A,B):-waiter_2(A,C),move_right(C,B).
waiter_2(A,B):-turn_cup_over(A,C),waiter_1(C,B).
waiter_1(A,B):-ifthenelse(A,B,wants_tea,pour_tea,pour_coffee).
```

This extension is implemented in a system called Metagol$_{AI}$ which uses higher-order background knowledge to learn higher-order programs. Metagol$_{AI}$ also supports higher-order predicate invention. For instance, to invent the *waiter_3/2* predicate in Example 10 which is used as an argument in *until/4*.

5.2 Related work

Many authors have advocated using higher-order logic for knowledge representation [79]. Lloyd [75] advocates using higher-order logic in the learning

process, but the approach focused on learning functional programs and did not support predicate invention.

Early work in ILP [39, 109, 35] used higher-order schema to specify the overall form of programs to be learned, similar to how metarules are used in MIL. However, these works did not consider learning higher-order programs. By contrast, we use higher-order logic as a learning representation and to represent learned hypotheses.

Feng and Muggleton [37] investigated inductive generalisation in higher-order logic using a restricted form of lambda calculus. However, their approach does not support first-order nor second-order predicate invention. By contrast, we introduce higher-order definitions which treat predicate symbols as first-class citizens. This approach supports a form of abstraction which goes beyond typical first-order predicate invention [115] in that the use of higher-order definitions combined with meta-interpretation drives both the search for a hypothesis and higher-order predicate invention, leading to more accurate and compact higher-order programs.

In this chapter, we introduce *abstractions*, which can be seen as hiding irrelevant or lower level details. This idea is in some ways similar to the work of Broda et al [13], who explored using continuous actions such as *move hand until you touch the table* to map low level sensor data to high level events.

5.3 Framework

To support learning higher-order programs, we introduce higher-order definitions:

Definition 34 (Higher-order definition) A higher-order definition is a set of higher-order definite clauses (Definition 5) with matching head predicates.

Two example higher-order definitions are:

Example 11 (Map definition)

map([],[],F) ←
map([A|As],[B|Bs],F) ← F(A,B), map(As,Bs)

Example 12 (Until definition)

until(A,A,Cond,F) ← Cond(A)
until(A,B,Cond,F) ← not(Cond(A)), F(A,C), until(C,B,Cond,F)

In contrast to metarules, which are individual formulas with existentially quantified variables, higher-order definitions are sets of universally quantified definite clauses. By only containing universally quantified variables, a MIL learner does not need to find and save meta-substitutions for the variables in a higher-order definition. We elaborate on this difference in the rest of this chapter.

Abstraction In computer science, *code abstraction* [14] involves hiding complex code to provide a simpler interface for users to select key details. In this work, we define an abstraction as a higher-order definite clause that contains at least one atom which takes a predicate as an argument:

Definition 35 (Abstraction) An abstraction is a higher-order definite clause of the form:

$$\forall \tau p(s_1, \ldots, s_m) \leftarrow q(u_1, \ldots, u_n, v_1, \ldots, v_o)$$

where $o > 0$, $\tau \subseteq \mathcal{V}_1 \cup \mathcal{V}_2$, $p, q, v_1, \ldots, v_o \in \mathcal{P}$, and $s_1, \ldots, s_m, u_1, \ldots, u_n \in \mathcal{V}_1$.

In the following abstraction example, the final argument of *map/3* is ground to the predicate symbol *succ*:

Example 13 (Abstraction)

f(A,B) ← map(A,B,succ)

A MIL learner uses abstractions to generate inventions (Definition 19). For example, to invent a chained successor clause for use in a *map/3* definition:

Example 14 (Invention)

f(A,B) ← map(A,B,f1)
f1(A,B) ← succ(A,C),succ(C,B)

5.3.1 Abstracted MIL

A MIL input (B, E) (Definition 14), consists of background knowledge B and examples E. The background knowledge $B = B_C \cup M$ consists of definite program background knowledge B_C and a set of metarules M. We now extend MIL by assuming that $B = B_C \cup B_I \cup M$, where B_i is a set of higher-order definitions. To help distinguish between B_C and B_I, we call B_C *compiled* background knowledge and B_I *interpreted* background knowledge. The distinction between B_C and B_I is subtle. Whereas a clause from B_C is proved deductively by calling Prolog, a clause from B_I is proved through meta-interpretation, which allows for predicate invention to be combined with abstractions to invent higher-order predicates. The distinction between B_I and M is that the clauses in B_I are all universally quantified, whereas the metarules in M contain existentially quantified variables whose meta-substitutions form the hypothesised program. We discuss this distinction in more detail in Section 5.4.

5.3.2 Language classes, expressivity, and complexity

As Chapter 4 showed, metarules determine the hypothesis space. For instance, the *chain* metarule (Figure 3.1) restricts clauses to be definite with two body atoms and only arity two predicates. This restriction corresponds to the class H^2_2 in which the number of programs expressible with n clauses and $|M|$ metarules is $O(|M|^n p^{3n})$ (Theorem 1). We now update this bound for the abstracted MIL framework:

Lemma 2 (Number of abstracted H^2_2 programs) Given p predicate symbols, $|M|$ metarules, and abstractions each with at most $k \geq 1$ higher-order variables, then the number of H^2_2 programs expressible with n clauses is:

$$O(|M|^n p^{(2+k)n})$$

Proof 7 Since each abstraction has at most $k \geq 1$ higher-order variables the number of clauses S_p which can be constructed from an H^2_2 metarule given p predicate symbols is at most $max(p^3, p^{2+k}) = p^{2+k}$. The set of such clauses $S_{m,p}$ has cardinality at most $|M|p^{2+k}$. It follows that the number of programs constructed from a selection of n rules chosen from $S_{m,p}$ is at most:

$$\binom{|M|p^{2+k}}{n} \leq (|M|p^{2+k})^n = O(|M|^n p^{(2+k)n}).$$

We use this result to develop sample complexity results for unabstracted versus abstracted MIL:

Theorem 4 (Sample complexity of unabstracted MIL) Unabstracted MIL has a polynomial sample complexity:

$$m_u \geq \frac{1}{\epsilon}(n\ ln(|M|) + 3n\ ln(p) + ln\frac{1}{\delta})$$

Proof 8 According to the Blumer bound [9] the error of consistent hypotheses is bounded by ϵ with probability at least $(1-\delta)$ once $m_u \geq \frac{1}{\epsilon}(ln|H| + ln\frac{1}{\delta})$, where $|H|$ is the size of the hypothesis space. From Theorem 1, $|H| = O(|M|^n p^{3n}$ for unabstracted MIL. Applying logs and substituting gives:

$$m_u \geq \frac{1}{\epsilon}(n\ ln(|M|) + 3n\ ln(p) + ln\frac{1}{\delta})$$

Theorem 5 (Sample complexity of abstracted MIL) Abstracted MIL has a polynomial sample complexity:

$$m_a \geq \frac{1}{\epsilon}(n\ ln(|M|) + (2+k)n\ ln(p) + ln\frac{1}{\delta}$$

Proof 9 Analogous to Theorem 4 using Lemma 2.

We now consider the ratio of these bounds in the case $n_u \gg p$:

Proposition 3 (Ratio of unabstracted and abstracted bounds) Given that m_u, m_a are the bounds on the number of training examples required to achieve error less than ϵ with probability at least $1-\delta$ and n_u, n_a are the numbers of clauses in the minimum expression of the target theories in these cases then the ratio $m_u : m_a$ approaches $n_u : n_a$ in the case $n_u \gg p$.

Proof 10 Since $n_u \gg p$ it follows $m_u : m_a \approx (n_u ln(m_u) : n_a ln(m_u)) = n_u : n_a$.

Proposition 3 indicates that abstraction in MIL reduces sample complexity proportional to the number of clauses required to express abstracted hypotheses. For instance, in Example 10, the predicates *until/4* and *ifthenelse/5* reduce the hypothesis size by one clause each, compared to Example 9. Thus the minimal hypothesis reduces from six clauses to four leading to a sample complexity reduction of 3 : 2. Figure 5.2 tabulates higher-order predicates with corresponding clause reductions.

HO predicate	Reduction
until/4	1
ifthenelse/5	1
map/3	1
filter/3	2

Figure 5.2: Reductions in the number of clauses when using higher-order predicates.

5.4 Metagol$_{AI}$

We now introduce Metagol$_{AI}$ which extends Metagol to support learning higher-order programs. The Prolog code for Metagol$_{AI}$ is:

```
learn(Pos,Neg,Prog):-
    prove(Pos,[],Prog),
    not(prove(Neg,Prog,Prog)).
prove([],Prog,Prog).
prove([Atom|Atoms],Prog1,Prog2):-
    prove_aux(Atom,Prog1,Prog3),
    prove(Atoms,Prog3,Prog2).
prove_aux(Atom,Prog,Prog):-
    prim(Atom),!,
    call(Atom).
prove_aux(Atom,Prog1,Prog2):-
    interpreted((Atom:-Body)),
    prove(Body,Prog1,Prog2).
prove_aux(Atom,Prog1,Prog2):-
    member(sub(Name,Subs),Prog1),
    metarule(Name,Subs,(Atom:-Body)),
    prove(Body,Prog1,Prog2).
prove_aux(Atom,Prog1,Prog2):-
    metarule(Name,Subs,(Atom:-Body)),
    prove(Body,[sub(Name,Subs)|Prog1],Prog2).
```

The key difference between Metagol$_{AI}$ and Metagol is the introduction of the second *prove_aux/3* clause in the meta-interpreter, denoted in boldface. This

clause allows Metagol$_{AI}$ to prove a goal by fetching a clause from the interpreted background knowledge (such as *map/3*) whose head unifies with a given goal. The distinction between compiled and interpreted background knowledge is that whereas a clause from the compiled background knowledge is proved deductively by calling Prolog, a clause from the interpreted background knowledge is proved through meta-interpretation. Meta-interpretation allows for predicate invention to be driven by the proof of conditions (as in *filter/3*) and functions (as in *map/3*). Interpreted background knowledge is different to metarules because the clauses are all universally quantified. By contrast, metarules contain existentially quantified variables whose meta-substitutions form the hypothesised program. The following examples show the three forms of background knowledge used by Metagol$_{AI}$:

Example 15 (Compiled background knowledge)

```
empty([]).
head([H|_],H).
tail([_|T],T).
move_forward(X/Y1),X/Y2):-Y2 is Y1+1.
```

Example 16 (Interpreted background knowledge)

```
interpreted(([map,[],[],F]:-[])).
interpreted(([map,[A|As],[B|Bs],F]:-
        [[F,A,B],[map,As,Bs,F]])).
interpreted(([fold,[],Acc,Acc,F]:-[])).
interpreted(([fold,[A|As],B,Acc1,F]:-
        [[F,Acc1,A,Acc2],[fold,As,B,Acc2,F]])).
```

Example 17 (Metarules)

```
metarule(identity,[P,Q],([P,A,B]:-[[Q,A,B]])).
metarule(precon,[P,Q,R],([P,A,B]:-[[Q,A],[R,A,B]])).
metarule(postcon,[P,Q,R],([P,A,B]:-[[Q,A,B],[R,B]])).
metarule(curry1,[P,Q,R],([P,A,B]:-[[Q,A,B,R]])).
metarule(chain,[P,Q,R],([P,A,B]:-[[Q,A,C],[R,C,B]])).
```

Algorithm Metagol$_{AI}$ works in the same way as Metagol, except for the use of interpreted background knowledge. Metagol$_{AI}$ first tries to prove a goal deductively using compiled background knowledge by delegating the proof to Prolog (*call(Atom)*), similar to how Metagol works. Failing this, Metagol$_{AI}$ tries to unify the goal with the head of a clause in the interpreted background knowledge (*background((Atom:-Body))*) and tries to prove the body of the matched clause. Metagol does not perform this additional step. Failing this, Metagol$_{AI}$ continues to work in the same way as Metagol. Metagol$_{AI}$ uses negation as failure [18] to negate predicates in the compiled background knowledge. To prevent floundering [99], Metagol$_{AI}$ ensures that the goal is ground. Negation of invented predicates is unsupported and is left for future work.

5.5 Experiments

We now describe three experiments which compare learning first-order and higher-order programs. We compare Metagol$_{AI}$ (which supports interpreted background knowledge) with Metagol (which does not support interpreted background knowledge), i.e. we compare abstracted MIL with unabstracted MIL. We test the null hypotheses:

Null hypothesis 1 Metagol$_{AI}$ cannot learn programs with higher predictive accuracies than Metagol

Null hypothesis 2 Metagol$_{AI}$ cannot learn programs with lower running times than Metagol

Common materials In each experiment, we provide Metagol$_{AI}$ and Metagol with the same background knowledge. The only variable in the experiments is the learning system. The only difference between the two systems is the additional clause used by Metagol$_{AI}$, described in the implementation section.

The compiled background knowledge varies in each experiment. The interpreted background knowledge is the same for each experiment and contains five higher-order definitions: *map/3*, *reduce/3*, *reduceback/3*, *until/4*, and *ifthenelse/5*. We use the metarules shown in Example 17.

Common methods We train using m randomly chosen positive examples for each m in the set {1,2,3,4,5}. We test using 40 examples, half positive and

half negative, so the default accuracy is 50%. We measure mean predictive accuracies and learning times over 20 trials. For each learning task, we enforce a 10-minute timeout.

5.5.1 Robot waiter

This experiment revisits the waiter example in Figure 5.1, in which a robot waiter is learning to serve drinks.

Materials Examples are *waiter/2* atoms where the first argument is the initial state and the second is the final state. A state is a list of facts. In the initial state, the robot starts at position 0; there are d cups facing down at positions $1,\ldots,d$; and for each cup there is a preference for tea or coffee. In the final state, the robot is at position $d+1$; all the cups are facing up; and each cup is filled with the preferred drink. We generate positive examples as follows. For the initial state, we select a random integer d from the interval $[1,20]$ as the number of cups. For each cup, we randomly select whether the preferred drink is tea or coffee and set it facing down. For the final state, we update the initial state so that each cup is facing up and is filled with the preferred drink. To generate negative examples, we repeat the aforementioned procedure but we modify the final state so that the drink choice is incorrect for a random subset of k drinks. The robot can perform the following fluents and actions defined as compiled background knowledge: *at_end/1*, *wants_tea/1*, *wants_coffee/1*, *move_left/2*, *move_right/2*, *turn_cup_over/2*, *pour_tea/2*, and *pour_coffee/2*.

Results Figure 5.3 shows that Metagol$_{AI}$ learns programs with higher predictive accuracies and lower learning times than Metagol, refuting null hypotheses 1 and 2. We can explain these results by looking at the programs learned by Metagol and Metagol$_{AI}$:

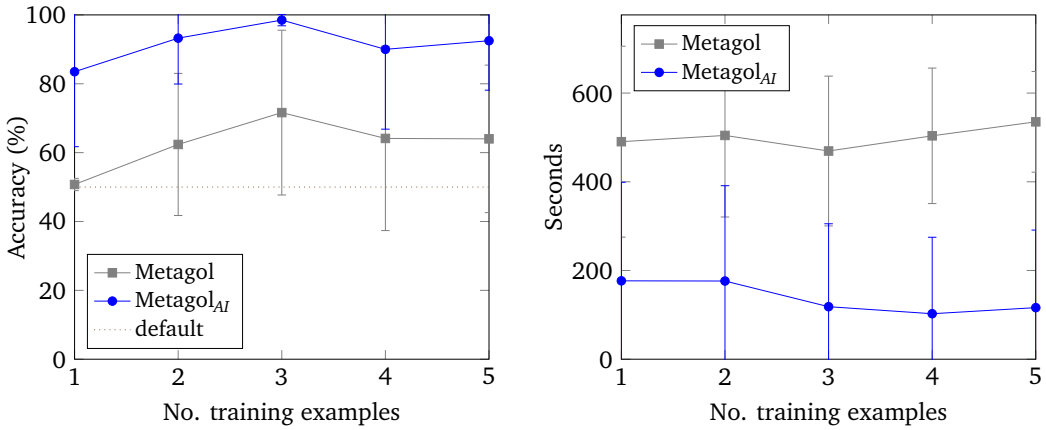
Program 1 (Metagol waiter program)

```
waiter(A,B):-waiter_3(A,B),at_end(B).
waiter(A,B):-waiter_3(A,C),waiter(C,B).
waiter_3(A,B):-waiter_2(A,C),move_right(C,B).
waiter_2(A,B):-turn_cup_over(A,C),waiter_1(C,B).
waiter_1(A,B):-wants_tea(A),pour_tea(A,B).
waiter_1(A,B):-wants_coffee(A),pour_coffee(A,B).
```

Program 2 (Metagol$_{AI}$ waiter program

```
waiter(A,B):-until(A,B,at_end,waiter_3).
waiter_3(A,B):-waiter_2(A,C),move_right(C,B).
waiter_2(A,B):-turn_cup_over(A,C),waiter_1(C,B).
waiter_1(A,B):-ifthenelse(A,B,wants_tea,pour_tea,pour_coffee).
```

Although both programs are general, and handle any number of guests and any assignment of drink preferences, Program 2 is smaller than Program 1 because it uses the higher-order abstractions *until/4* and *ifthenelse/5*. This compactness affects predicate accuracies because, whereas Metagol$_{AI}$ finds programs in the allocated time, Metagol struggles because the programs are too big.



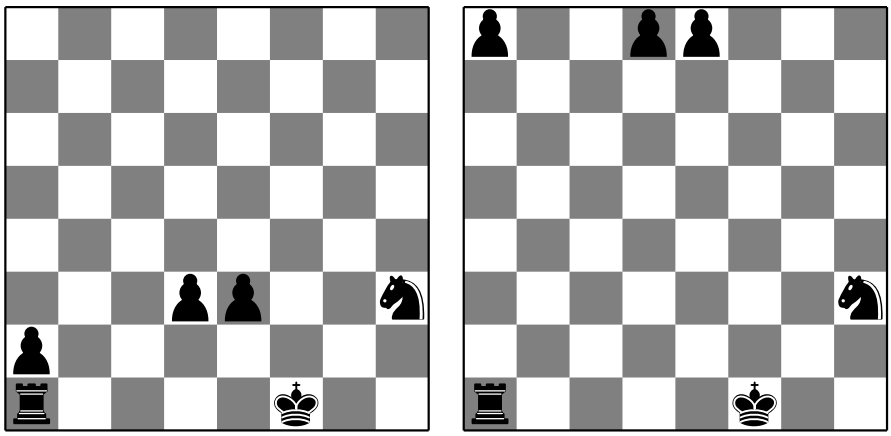
(a) Predictive accuracies (b) Learning times

Figure 5.3: Robot waiter experiment results which show learning performance when varying the number of training examples.

5.5.2 Chess strategy

Programming chess strategies is a difficult task for humans [11]. For example, consider maintaining a wall of pawns to support promotion [48]. In this case, we might start by trying to inductively program the simple situation in which a black pawn wall advances without interference from white. Having constructed such a program one might consider using negative examples involving interposition of white pieces to deal with exceptional behaviour. Figure 5.4 shows such an

example, where in the initial state pawns are at different ranks, and in the final state all the pawns have advanced to rank 8, but the other pieces have remained in the initial positions. In this experiment, we try to learn such strategies.



(a) Initial state (b) Final state

Figure 5.4: Chess initial/final state example.

Materials Examples are *chess/2* atoms where the first argument is the initial state and the second is the final state. A state is a list of pieces, where a piece is denoted as a triple of the form *(Type,Id,X/Y)*, where *Type* is the type (king=k, pawn=p, etc.), *Id* is a unique identifier, and *X/Y* is the position. We generate positive examples as follows. For the initial state, we select a random subset of n pieces from the interval $[2, 16]$ and randomly place them on the board. For the final state, we update the initial state so that each pawn finishes at rank 8. To generate negative examples, we repeat the aforementioned procedure but we randomise the final state positions, whilst ensuring that the input/output pair is not a positive example. We use the following compiled background knowledge:

```
at_rank8((_,_,_/8)).
is_pawn((p,_,_)).
not_pawn(X):-not(is_pawn(X)).
empty([]).
move_forward((Type,Id,X/Y1),(Type,Id,X/Y2)):-
    Y1 < 8,Y2 is Y1+1.
```

```
move_forward(A,B,Id):-
    append(Prefix,[(Type,Id,X/Y1)|Suffix],A),
    Y1 < 8,Y2 is Y1+1,
    append(Prefix,[(Type,Id,X/Y2)|Suffix],B).
```

Results Figure 5.5a shows that Metagol$_{AI}$ learns programs approaching 100% accuracy after two examples. By contrast, Metagol learns programs with around default accuracy. This result refutes null hypothesis 1. The log-lin plot in Figure 5.5b shows that Metagol$_{AI}$ learns programs quicker than Metagol, refuting null hypothesis 2. We can explain these results by looking at the sample programs learned by Metagol and Metagol$_{AI}$:

Program 3 (Metagol chess program)

```
chess(A,B):-chess_2(A,C),chess_2(C,B).
chess_2(A,B):-chess_1(A,C),chess_1(C,B).
chess_1(A,B):-move_forward(A,B,p3).
chess_2(A,B):-move_forward(A,B,p5).
```

Program 4 (Metagol$_{AI}$ chess program)

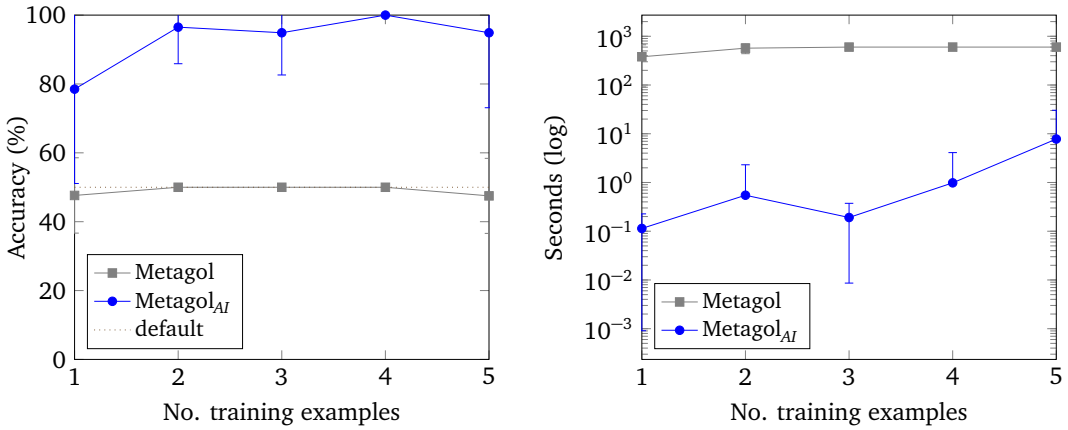
```
chess(A,B):-map(A,B,chess_1).
chess_1(A,A):-not_pawn(A).
chess_1(A,B):-until(A,B,at_rank8,move_forward).
```

Metagol$_{AI}$ learns a small higher-order program using the abstractions *map/3* and *until/4*, where the *map/3* operation decomposes the problem into smaller sub-goals of moving a single piece to rank 8. These sub-goals are solved by the *chess_1/2* predicate. By contrast, Metagol learns a larger recursive and more specific first-order program.

5.5.3 Droplast

In this experiment, the goal is to learn a program that drops the last element from each sublist of a given list – a problem frequently used to evaluate program induction systems [60]. Below are input/output examples for this problem:

```
droplast([[a,b],[a,b,c],[a,b,c,d]],[[a],[a,b],[a,b,c]]).
droplast([[l,o,n,d,o,n],[p,a,r,i,s]],[[l,o,n,d,o],[p,a,r,i]]).
droplast([[a,n,n],[b,o,b],[c,h,u,c,k]],[[a,n],[b,o],[c,h,u,c]]).
```



(a) Predictive accuracies

(b) Learning times

Figure 5.5: Chess experiment results which show learning performance when varying the number of training examples.

Materials Examples are *droplast/2* atoms where the first argument is the initial list and the second is the final list. We generate positive examples as follows. To form the input, we select a random integer i from the interval $[2, 20]$ as the number of sublists. For each sublist i, we select a random integer k from the interval $[1, 100]$ and populate sublist i with k random integers. To form the output, we wrote a Prolog program to drop the last element from each sublist. We use the following compiled background knowledge:

```
head([H|_],H).
tail([_|T],T).
concat([H|T],B,C):-append([H|T],[B],C).
concat(A,B,C):-append([A],[B],C).
```

Results Metagol$_{AI}$ achieved 100% accuracy after two examples (plot omitted for brevity). Metagol$_{AI}$ learned the program:

Program 5 (Metagol$_{AI}$ droplast program)

```
droplast(A,B):-map(A,B,droplast_3).
droplast_3(A,B):-droplast_2(A,C),droplast_1(C,B).
droplast_2(A,B):-droplast_1(A,C),tail(C,B).
droplast_1(A,B):-reduceback(A,B,concat).
```

This program contains notable sub-programs. The invented predicate *droplast_1/2* reverses a given list. The invented predicate *droplast_3/2* drops the last element from a single list by (1) reversing the list by calling *droplast_1/2*, (2) dropping the head from the reversed list, and (3) reversing the shortened list back to the original order by again calling *droplast_1/2*. Finally, *droplast/2* maps over the input list and applies *droplast_3/2* to each sublist to form the output list. This program highlights invention through the repeated calls to *droplast_1/2* and abstraction through the higher-order functions. By contrast, Metagol was unable to learn any program for this problem because the corresponding first-order program is too long and the search is impractical.

Further discussion To further demonstrate invention and abstraction, consider learning a program *ddroplast/2* which extends the *droplast* problem so that, in addition to dropping the last element from each sublist, it also drops the last sublist, such as:

```
ddroplast([[a,b],[a,b,c],[a,b,c,d]],[[a],[a,b]]).
ddroplast([[l,o,n,d,o,n],[p,a,r,i,s]],[[l,o,n,d,o]]).
ddroplast([[a,n,n],[b,o,b],[c,h,u,c,k]],[[a,n],[b,o]).
```

Given two examples of this problem under the same conditions as in Section 5.5.3, Metagol$_{AI}$ learns the program:

Program 6 (Metagol$_{AI}$ ddroplast program)

```
ddroplast(A,B):-ddroplast_4(A,C),ddroplast_3(C,B).
ddroplast_4(A,B):-map(A,B,ddroplast_3).
ddroplast_3(A,B):-ddroplast_2(A,C),ddroplast_1(C,B).
ddroplast_2(A,B):-ddroplast_1(A,C),tail(C,B).
ddroplast_1(A,B):-reduceback(A,B,concat).
```

This program is similar to Program 5 but it makes an additional final call to the invented predicate *ddroplast_3/2*, which is used twice in the program as a higher-order argument in *ddroplast_4/2* and as a first-order predicate in *ddroplast/2*.

5.6 Future work

The experiments in this chapter have focused on using functional constructs, such as *map/3* and *until/4*. In future work, we would like to investigate using relational constructs. For instance, consider this higher-order definition of a closure:

closure(P,A,B) ← P(A,B)
closure(P,A,B) ← P(A,C), closure(P,C,B)

This definition could be used to learn compact abstractions of relations, such as:

ancestor(A,A) ← closure(parent,A,B)
lessthan(A,A) ← closure(increment,A,B)
subterm(A,B) ← closure(headortail,A,B)

Moreover, the issue of how metarules might themselves be learned could be treated in a similar fashion using higher-order programs such as:

chain(P,Q,R,A,B) ← Q(A,C), R(C,B)
inverse(P,Q,A,B) ← Q(B,A)

5.7 Summary

In this chapter, we have introduced Metagol$_{AI}$ which uses higher-order definitions and abstractions to support learning higher-order programs. We have shown that this approach reduces the size of programs necessary to represent target classes of programs. This reduction in program size reduces the hypothesis space which in turn reduces the sample complexity. Our sample complexity results are consistent with our experiments which indicate increased predictive accuracy and decreased learning time for abstracted MIL compared with unabstracted MIL. This chapter supports Claim 1 of the thesis. In the rest of this thesis, any reference to Metagol refers to Metagol$_{AI}$.

Chapter 6

Metaopt

In the first half of this thesis, we supported Claim 1 by introducing techniques to efficiently learn programs. We now support Claim 2 by introducing techniques to learn efficient programs.

6.1 Introduction

As explained in the introduction, we often want to learn efficient programs, but existing program induction approaches cannot distinguish between the efficiencies of programs, and instead learn textually simple programs. To address this limitation, in this chapter, we introduce the *cost minimisation problem*, a general setting for learning efficient programs. We also introduce Metaopt, which extends Metagol by adding a general cost function into the meta-interpreter, where specific cost functions are provided as background knowledge. To learn minimal cost programs, Metaopt uses a search procedure called *iterative descent* which iteratively searches for more efficient programs, each time enforcing a tighter restriction on the hypothesis space. We show that given sufficient examples, Metaopt converges on minimal cost programs.

6.2 Cost minimisation problem

We now define the cost minimisation problem. We denote the power set of the set S as 2^S. We denote the Herbrand base of $B \cup E$ as $\mathcal{B}_{B,E}$. We first declare a general program cost function that measures the cost of a program with respect

to an atom:

Definition 36 (Program cost) A program cost function is of the form:

$$\Phi : \mathcal{H} \times 2^{\mathcal{B}_{B,E}} \rightarrow \mathbf{N}$$

We now define the input for the cost minimisation problem:

Definition 37 (Cost minimisation input) The cost minimisation input is a tuple (B, E, Φ, τ) where:

- B is a logic program representing background knowledge
- $E = (E^+, E^-)$ where E^+ and E^- are sets of facts representing positive and negative examples respectively
- Φ is a program cost function
- τ is a textual complexity function (Definition 20)

We measure the cost of a program as its worst-case cost over a set of examples:

Definition 38 (Worst-case program cost) Let E^+ be a set of positive examples and Φ be a program cost function. Then the worst-case cost of a program $H \in \mathcal{H}$ is defined as:

$$\Psi(\Phi, H, E^+) = max_{e \in E^+} \Phi(H, e)$$

We use the worst-case cost of a program to define an ordering which judges whether one program is more efficient than another:

Definition 39 (More efficient ordering $\preceq_{\Phi,\tau}$) Let (B, E, Φ, τ) be a cost minimisation input and $H_1, H_2 \in \mathcal{H}$. Then $H_1 \preceq_{\Phi,\tau} H_2$ iff either:

1. $\Psi(\Phi, H_1, E^+) < \Psi(\Phi, H_2, E^+)$
2. $\Psi(\Phi, H_1, E^+) = \Psi(\Phi, H_2, E^+)$ and $\tau(H_1) \leq \tau(H_2)$

We define the solution to the cost minimisation problem:

Definition 40 (Cost minimisation solution) A solution to the cost minimisation problem (B, E, Φ, τ) is a program $H \in \mathcal{VS}_{B,E}$ such that $H \preceq_{\Phi,\tau} H'$ for all $H' \in \mathcal{VS}_{B,E}$.

There can be multiple solutions to the cost minimisation problem. We call any solution a minimal cost program:

Definition 41 (Minimal cost program) Let H be a solution to the cost minimisation problem (B, E, Φ, τ). Then H is a minimal cost program.

6.3 Metaopt

Metaopt extends Metagol to support learning minimal cost programs. The two key extensions are (1) the addition of a general cost function into the meta-interpreter, and (2) the use of a procedure called iterative descent to search for efficient programs. We describe these extensions in turn.

Meta-interpreter

The key extension in Metaopt is the addition of a proof cost and program cost into the meta-interpreter:

```
learn(Pos,Neg,Prog):-
    prove(Pos,[],Prog,0,_),
    not(prove(Neg,Prog,Prog,0,_)).
prove([],Prog,Prog,C,C).
prove([Atom|Atoms],Prog1,Prog2,C1,C2):-
    prove_aux(Atom,Prog1,Prog3,C1,C3),
    prove(Atoms,Prog3,Prog2,C3,C2).
prove_aux(Atom,Prog,Prog,C1,C2):-
    prim(Atom),!,
    program_cost(Prog,[Atom],Cost),
    C2 is C1+Cost,
    get_max_cost(MaxCost),
    C2<MaxCost.
prove_aux(Atom,Prog1,Prog2,C1,C2):-
    interpreted((Atom:-Body)),
    prove(Body,Prog1,Prog2,C1,C2).
prove_aux(Atom,Prog1,Prog2,C1,C2):-
    member(MetaSub,Prog1),
    metarule(MetaSub,(Atom:-Body)),
    prove(Body,Prog1,Prog2,C1,C2).
prove_aux(Atom,Prog1,Prog2,C1,C2):-
    metarule(MetaSub,(Atom:-Body)),
    prove(Body,[sub(MetaSub)|Prog1],Prog2,C1,C2).
program_cost(Prog,Pos,ProgramCost):-
    assert_program(Prog),
    findall(C,(member(Atom,Pos),program_cost(Atom,C)),Costs),
```

```
    max_list(Costs,ProgramCost),
    retract_program(Prog).
```

This proof cost, denoted by the variables C_i, is used as follows. Given a set of atoms, Metaopt constructs a proof of the atoms. Whilst constructing the proof, when an atom is proven using compiled knowledge (Section 5.3), the cost of proving that atom is added to the overall proof cost. The cost of proving an atom is defined by a predicate called *program_cost/3*, which is defined in terms of a predicate called *program_cost/2*, supplied as background knowledge. During the proof, if the overall proof cost exceeds a bound (*MaxCost*), then the proof is terminated, as to ignore inefficient programs in the hypothesis space. The bound is determined by the iterative descent procedure, described in the next section. Once the proof is complete, a logic program is formed by projecting the meta-substitutions onto the metarules.

Iterative descent

The Metaopt meta-interpreter is controlled by the iterative descent algorithm:

```
metaopt(Pos,Neg):-
    learn(Pos,Neg,Prog),
    program_cost(Prog,Pos,Cost),
    is_better(Cost),
    set_max_cost(Cost),
    set_best_program(Prog),
    false.
metaopt(_,_):-
    get_best_program(Prog),
    pprint(Prog).
```

This algorithm works as follows. Starting at iteration 1, Metaopt uses iterative deepening on the number of clauses to find a textually minimal program H_1. The program H_1 is the quickest to learn because the hypothesis space is exponential in the number of clauses (Theorems 1 and Lemma 2). Metaopt then calculates the worst-case program cost $\Psi(H_1, E^+)$ of H_1 to derive an upper bound for subsequent iterations. In iteration $i > 1$, Metaopt searches for a program H_i, again with minimal textual complexity, but with a cost such that $\Psi(H_i, E^+) < \Psi(H_{i-1}, E^+)$. Metaopt continues to search until it cannot find a more efficient program.

We now prove convergence of iterative descent to minimal cost programs (Definition 41) given sufficiently large numbers of examples:

Theorem 6 (Iterative descent convergence) Assume E^+ consists of m positive examples drawn randomly and independently from instance distribution D. Without loss of generality consider the hypothesis space formed of two programs H_1 and H_2 such that $H_1 \preceq_{\Phi,\tau_c} H_2$ for an arbitrary cost function Φ and the clause complexity τ_c (Definition 21). Then in the limit, iterative descent will return H_1 in preference to H_2.

Proof 11 Assume false, which, because of Definition 39, implies that either (1) $\Psi(H_1, E^+) > \Psi(H_2, E^+)$, or (2) $\Psi(H_1, E^+) = \Psi(H_2, E^+)$ and $\tau_c(H_1, E^+) > \tau_c(H_2, E^+)$.

Case 1 With sufficiently large m there will exist an example e such that $\Phi(H_1, e) < \Phi(H_2, e)$ and $\Phi(H_2, e) > \Phi(H_2, e')$ for all other e' in E^+ and $\Phi(H_1, e) > \Phi(H_1, e')$ for all other e' in E^+. In this case $\Psi(H_1, E^+) < \Psi(H_2, E^+)$ and iterative descent returns H_1, which has the minimal program cost. This contradicts the assumption and we discard this case.

Case 2 Iterative descent performs iterative deepening search (IDS) on the number of clauses. From the optimality of IDS, iterative descent returns H_1, which has minimal clause complexity, i.e. $\tau_c(H_1) < \tau_c(H_2)$. This contradicts the assumption and we discard this case.

These two cases are exhaustive, and thus the proof is complete.

6.4 Summary

In this chapter, we have addressed Claim 2 by introducing the cost minimisation problem, a framework for learning efficient logic programs, where an efficient program is one that minimises an associated cost function. We have also introduced Metaopt, a MIL learner which finds solutions to the cost minimisation problem, which we call *minimal cost programs*. To find minimal cost programs, Metaopt uses a procedure called iterative descent which iteratively learns more efficient programs, each time further restricting the hypothesis space. We have shown (Theorem 6) that given sufficient training examples, Metaopt converges on minimal cost programs. In the following two chapters, we support Claim 2 by using Metaopt to learn efficient robot strategies (Chapter 7) and efficient time complexity logic programs (Chapter 8).

Chapter 7

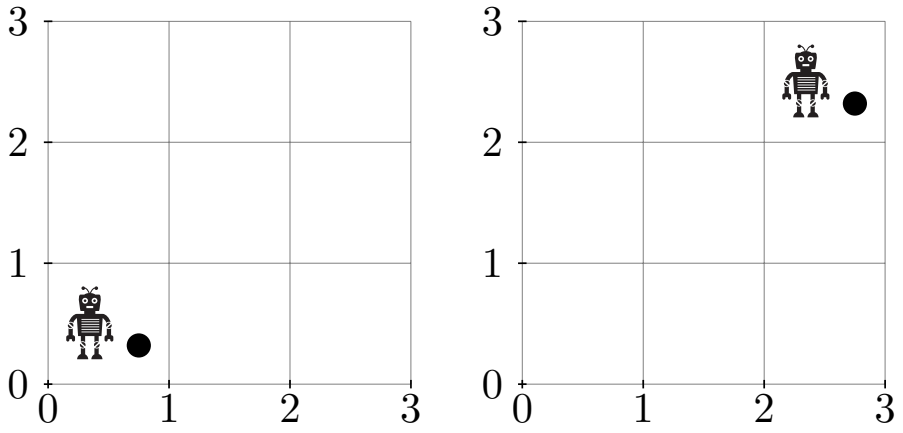
Learning efficient robot strategies

In this chapter, we support Claim 2 of this thesis by using Metaopt to learn minimal *resource complexity* robot strategies.

7.1 Introduction

Suppose we are machine learning robot plans from initial/final state examples. Figure 7.1 shows such a scenario, in which a robot is trying to move a ball in a two-dimensional space. The robot can perform four movement actions: *north/2*, *south/2*, *east/2*, and *west/2*, and two other actions: *grab/2* and *drop/2*. Figure 7.2 shows two programs for this problem, where Metagol learned program (a) and Metaopt learned program (b). Although both programs correctly transform the initial state to the final state and are equal in their textual complexity (Definition 21), the programs differ in efficiency. Program (a) is inefficient because the robot picks up the ball, moves north, moves east, drops the ball, and then repeats this procedure again, requiring two *grab/2* and two *drop/2* operations. By contrast, program (b) is efficient because it requires only one *grab/2* and one *drop/2* operation.

However, as already stated, existing program induction systems cannot distinguish between the efficiencies of programs. In this chapter, we use Metaopt to learn efficient robot *strategies* from initial/final state examples. In contrast to traditional AI planning [114], which involves the generation of a plan as a sequence of actions transforming a particular initial state to a particular final state, a strategy can be viewed as a potentially infinite set of plans, applicable to a class of initial/final state pairs [24].



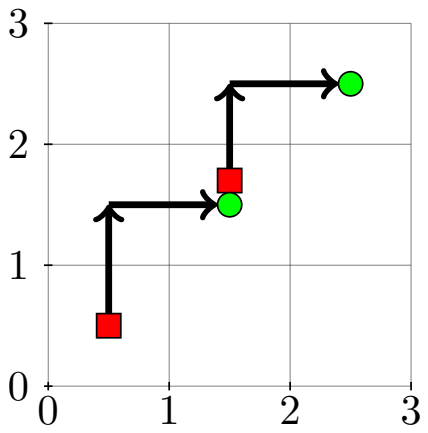
(a) Initial state (b) Final state

Figure 7.1: Robot planning initial/final state example. In the initial state (a) a robot and a ball are in position 1/1. In the final state (b) a robot and a ball are in position 3/3.

We focus on learning minimal *resource complexity* strategies. Resource complexity is a measure of the efficiency of a strategy, where energy and material consumption, such as solder, glue, or bricks, could be considered as resources. We represent the problem of learning minimal resource complexity strategies as a cost minimisation problem (Chapter 6). Our experiments on three robot problems (searcher, postman, sorter) show that Metaopt learns efficient strategies, in contrast to Metagol, which learns inefficient strategies. For instance, in the robot sorter problem, we show that Metaopt learns an efficient quick sort strategy, rather than an inefficient bubble sort strategy.

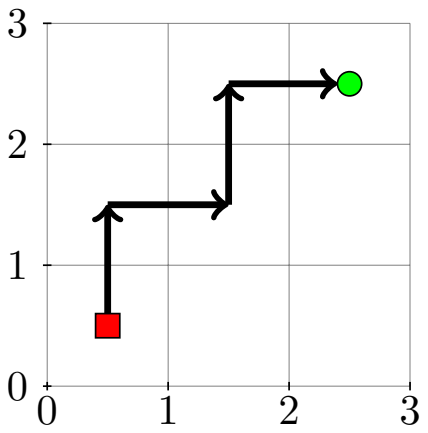
7.2 Related work

Classical planning typically focuses on efficiently learning plans [51]. However, we are often interested in plans that are optimal with respect to an objective function which measures the quality of a plan. A common objective function is the length of the plan [142], and existing systems can learn optimal plans based on this function [67, 97, 34]. Plan length alone is only one criterion. If executing actions is costly, we may prefer a plan which minimises the overall cost of the actions, e.g. to minimise the use of resources. The answer set programming




```
f(A,B):-f3(A,C),f3(C,B).
f3(A,B):-f2(A,C),drop(C,B).
f2(A,B):-grab(A,C),f1(C,B).
f1(A,B):-north(A,C),east(C,B).
```

(a) Inefficient program




```
f(A,B):-f3(A,C),drop(C,B).
f3(A,B):-grab(A,C),f2(C,B).
f2(A,B):-f1(A,C),f1(C,B).
f1(A,B):-north(A,C),east(C,B).
```

(b) Efficient program

Figure 7.2: Programs learned by Metagol (a) and Metaopt (b) for the planning example in Figure 7.1. A red square denotes a *grab/2* action and a green circle denotes a *drop/2* action.

(ASP) literature has started to address learning optimal plans by incorporating action costs into the learning [34, 56]. However, classical planning, including the aforementioned ASP work, focuses on finding a sequence of actions to get from one particular start state to one particular goal state [114], By contrast, we focus on learning strategies, where the goal is to induce a logic program that represents a potentially infinite set of plans, i.e. goes from a set of start states to a set of goal states, where both sets could be infinite. For instance, in Section 7.5.3, we describe experiments for learning sorting strategies, where the learned programs are able to generalise over any size of list.

Various machine learning approaches support constructing strategies, such as the SOAR architecture [64], action learning in ILP [85, 100], and reinforcement learning [131]. In this work, we aim to learn strategies with minimal cost, which is analogous to learning optimal policies for an markov deicison processes [105]. Q-learning [139] is a common approach to reinforcement learning which assigns values to state-action pairs and thus implicitly represents

policies. However, whereas a policy specifies what an agent should do in every state, a logic program strategy specifies what an agent should do in a small number of states – although strategies could be viewed as a deterministic special case of markov decision processes.

Relational reinforcement learning (RLL) [33] combines ILP and reinforcement learning, and is used to learn plans for a blocks world. The authors introduce P-functions, which are binary classifiers that deem whether an action and state pair are optimal, where the suggestion that this approach will allow for greater generalisation over Q-functions.

Our work differs from RLL in several ways. In RLL, the action costs are initially unknown to an agent (i.e. a learner). By contrast, in our work we are given action costs as part of the BK. In RLL, the learned policy is for a fixed number of states, known apriori, and it is unclear how the approach could generalise to instances beyond the training data, such as to learn minimal cost robot sorting strategies, where the size of the lists to be sorted in the test examples can greatly exceed those seen during training. Indeed, as the authors point out, for large state spaces, the probability of reaching the goal state by random exploration is extremely low. For instance, in the sorter experiment, for list of length 25, there at least 25! states, and that does not take into account other permutations of the state, such as the position of the robot, or whether the robot is holding an object, etc.

Overall, unlike these alternative approaches, in this thesis, we learn recursive logic programs that generalise beyond the training instances. These programs are also are optimal with respect to a program cost function (Definition 36). In addition, we use predicate invention for automatic problem decomposition.

7.3 Resource complexity

We now describe a MIL setting specific for learning minimal resource complexity robot strategies.

A state is a set of facts. Let (B, E) a MIL input, S be an enumerable set of states, and $\mathcal{P}$ be the predicate signature of (B, E). Then $\mathcal{P} = \mathcal{P}_a \cup \mathcal{P}_f$ where $\mathcal{P}_a$ and $\mathcal{P}_f$ are disjoint sets of action and fluent predicate symbols respectively. Actions are dyadic predicates which transform one state to another, i.e. each action $a \in \mathcal{P}_a$ is a function $a : S \to S$. Fluents are monadic predicates which apply to a state, i.e. each fluent $f \in \mathcal{P}_f$ is a function $f : S \to \{T, F\}$. We assume an action cost function $r : \mathcal{P}_a \times S \times S \to \mathbf{N}$ which defines the resources

consumed by calling an action $p \in \mathcal{P}_a$ to transform a state $s_1 \in S$ to a state $s_2 \in S$. A strategy $H \in \mathcal{H}$ is a definite program formed of actions and fluents.

We define a program cost function (Definition 36) to measure the resource complexity of a strategy:

Definition 42 (Resource complexity) Let $e = (s_1, s_2)$ be an example where $s_1, s_2 \in S$. Then the resource complexity $\Phi_{rc}(H, e)$ of a strategy $H \in \mathcal{H}$ is the sum of the action costs in applying H to e to transform s_1 to s_2.

We learn minimal resource complexity strategies by representing the problem as a cost minimisation problem (Chapter 6). The input is the tuple $(B, E, \Phi_{rc}, \tau_c)$ where Φ_{rc} is as stated in Definition 42 and τ_c is as stated in Definition 21. A solution to this cost minimisation problem is a minimal resource complexity strategy.

7.4 Implementation

We use Metaopt to learn minimal resource complexity strategies. Metaopt assumes a program cost function (Definition 36) as background knowledge. In the experiments in this chapter, we use the program cost function:

```
program_cost(Atom,Cost):-
    Atom=..[P,A,B],
    world_check(cost(E1),A),
    call(Atom),
    world_check(cost(E2),B),
    Cost is E2-E1.
```

This cost function gets the resource complexity of a strategy from the state. A robot action modifies the state. Each time a robot action is successfully executed, the cost of performing that action is added to the overall cost of the strategy, which is itself maintained in the state as the monadic fact *cost*.

Example 18 (Robot action cost) Suppose we have the dyadic action *move_right/2*, which increments a robot's position by one and has an action cost of two. Let the initial state be:

```
[robot_pos(0),cost(0)]
```

Then performing the action *move_right/2* will change the initial state to:

```
[robot_pos(1),cost(2)]
```

In Experiments 1, 2, and 3, Metaopt uses this implementation to learn minimal resource complexity programs. When supplied with this program cost function, we refer to Metaopt as Metaopt$_{rc}$.

7.5 Experiments

We now describe three experiments in learning robot strategies which compare Metaopt to Metagol. We investigate the null hypothesis:

Null hypothesis 1 Metaopt$_{rc}$ cannot learn lower resource complexity strategies than Metagol

Common materials We use the same representation in all the experiments in which there is humanoid robot in a one-dimensional space[1]. The robot can perform actions to transform the state. Several actions are available in all the experiments: *move_right/2*, *move_left/2*, *pick_up_left/2*, *pick_up_right/2*, *drop_left/2*, *drop_right/2*, *go_start/2*, and *go_end/2*. Some actions are defined in terms of others. For instance, the action *go_start/2* is defined in terms of *move_left/2*. The robot can check the state using fluents, for instance to perform equality checks on objects which it is holding. The robot can reason about its environment (the one-dimensional space), for instance to check its position using the fluents *at_start_position/1* and *at_end_position/1*. Because the robot knows the size of the space, it can move to the middle position using the action *go_middle/2*. The robot can manipulate its environment by changing the start and end positions using the actions *increment_start_position/2* and *decrement_end_position/2* respectively.

We compare strategies learned by Metaopt$_{rc}$ to strategies learned by Metagol. In all experiments, we train using positive examples only.

[1] A one-dimensional space is used for simplicity and the learner can handle any n-dimensional space

7.5.1 Learning robot librarian strategies

Materials Imagine a robot librarian learning to find books. In the initial state, there is an ordered list of d integers and the robot is at position 1 holding the value k in its left hand. In the final state, the robot is in position k and is holding the value k in both its left and right hands, i.e. the robot has found the value k and has picked it up. In addition to the common background knowledge, the robot can perform two actions: *set_start_current_if_left_bigger/2* and *set_end_current_if_left_smaller/2*. Performing comparisons has a cost of 1. All other actions have no cost. An efficient strategy is one that minimises comparisons. To evaluate whether Metaopt$_{rc}$ learns minimal cost strategies, we state the resource complexity of a minimal cost librarian strategy:

Proposition 4 (Minimal cost librarian strategy) Let d be the number of books. Then a minimal cost strategy involves $log\ d$ comparisons and the resource complexity is $O(log\ d)$.

Sketch proof 1 The robot can perform binary search requiring $O(log\ d)$ comparisons.

In addition, we supply Metagol and Metaopt with the *precon*, *chain*, and *tailrec* metarules (Figure 3.1).

Methods Examples are *searcher/2* atoms, where the first argument is the initial state and the second is the final state. To generate training examples, we select a random integer d from the interval $[1, 1000]$ representing the number of books[2]. We select a random integer k from the interval $[1, d]$ representing the position of the book to be found. To generate testing examples we repeat the aforementioned procedure but with a fixed number of books d for each value in the set {200,400,...,2000} to measure the resource complexity as d grows. We use 5 training examples and 10 testing examples. We measure mean resource complexities of learned strategies over 10 trials and limit the search to strategies of length five.

Results The log-lin plot in Figure 7.3 shows that Metaopt$_{rc}$ learns robot strategies with resource complexities that match the theoretical predictions stated in Proposition 4. By contrast, Metagol learns programs with non-minimal resource

[2] 1000 is an arbitrary limit and the learner handle any finite limit

complexities. This result refutes null hypothesis 1. We can explain these results by looking at the strategies learned by Metagol and Metaopt$_{rc}$:

Program 7 (Metagol librarian strategy)

```
librarian(A,B):-same(A),pick_up_right(A,B).
librarian(A,B):-move_right(A,C),librarian(C,B).
```

Program 8 (Metaopt$_{rc}$ librarian strategy)

```
librarian(A,B):-go_middle(A,C),librarian_1(C,B).
librarian(A,B):-librarian_1(A,C),librarian(C,B).
librarian_1(A,B):-go_middle(A,C),set_start_current_if_left_bigger(C,B).
librarian_1(A,B):-go_middle(A,C),set_end_current_if_left_smaller(C,B).
librarian_1(A,B):-same(A),pick_up_right(A,B).
```

In accordance with its Occamist bias, Metagol found a compact but inefficient sequential search strategy with resource complexity $O(d)$. By contrast, Metaopt$_{rc}$, having initially found the same sequential strategy, continued to search for a more efficient strategy, eventually learning a recursive binary search strategy with resource complexity $O(log\ d)$.

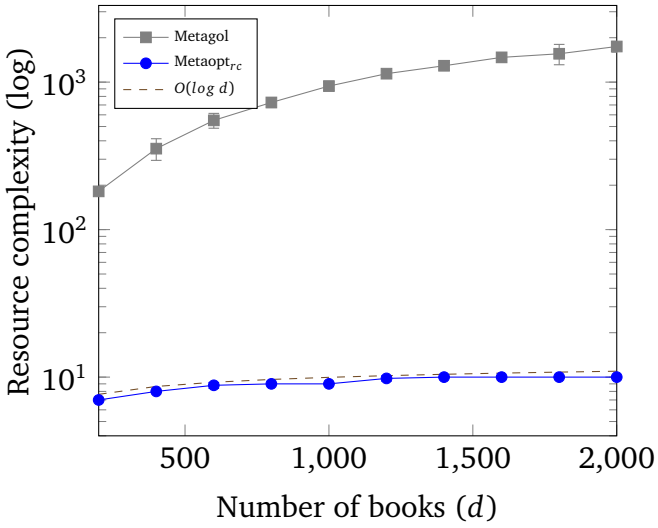
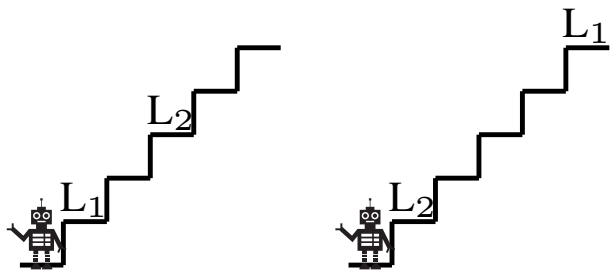


Figure 7.3: Log-lin plot of mean resource complexity of learned librarian strategies when varying the number of books (d).

7.5.2 Learning robot postman strategies

Materials Imagine a robot postman learning to collect and deliver letters. Figure 7.4 shows such a scenario, where the robot is on a hill with d houses (the size of the space). In the initial state, the robot is at position 1 and n letters are to be collected. In the final state, the robot is at position 1 and n letters have been delivered to their intended destinations. In addition to the common background knowledge, the robot can perform five actions: *take_letter/2*, *bag_letter/2*, *give_letter/2*, *find_next_sender/2*, and *find_next_recipient/2*. The robot can take and carry a single letter from a sender using the action *take_letter/2*. Alternatively, the robot can take a letter from a sender and place it a postbag using the action *bag_letter/2*, which allows the robot to carry multiple letters. The actions *move_left/2*, *move_right/2*, *take_letter/2*, and *bag_letter/2* have a cost of 1. All other actions have no cost. We now state the resource complexity of a minimal cost postman strategy:



(a) Initial state (b) Final state

Figure 7.4: Postman initial/final state examples for a route on a hill. In the initial states letters are to be collected. In the final states letters are at their intended destinations.

Proposition 5 (Minimal cost postman strategy) Let d be the number of houses and n be the number of letters to be collected and delivered. Then the resource complexity of a minimal cost strategy is $O(d+n)$.

Sketch proof 2 The minimal cost strategy involves the postman using the postbag to hold all n objects. This approach involves d steps for a single-pass traversal and n letter collections. The postman then needs to deliver each object to its destination, which again involves at most d steps for traversal and n

letter deliveries. Thus the overall complexity is bounded by $2(n+d)$, which is $O(n+d)$.

In addition, we use the *chain* and *tailrec* metarules (Fig 3.1).

Methods Examples are *pman/2* atoms. To generate training examples, we select a random integer d from the interval $[10, 25]$ representing the number of houses. We select a random integer n from the interval $[1, 5]$ representing the number of letters. For each letter we select random integers i and j from the interval $[1, d]$ representing the letter's start and end positions, such that $i \neq j$. To generate testing examples we repeat the aforementioned procedure but with a fixed number of letters n from the set $\{2, 4, \ldots, 20\}$ to measure the resource complexity as n grows. We use 5 training and 5 testing examples. We measure mean resource complexities of learned strategies over 10 trials and limit the search to strategies of length five.

Results Figure 7.5 shows that Metaopt$_{rc}$ learns strategies with resource complexities that match the theoretical predictions (Proposition 5), whereas Metagol learns non-minimal strategies. We can explain these results by looking at the strategies learned by Metagol and Metaopt$_{rc}$:

Program 9 (Metagol pman strategy)

```
pman(A,B):-pman_2(A,C),f(C,B).
pman(A,B):-pman_1(A,C),go_start(C,B).
pman_2(A,B):-pman_1(A,C),go_start(C,B).
pman_1(A,B):-find_next_sender(A,C),take_letter(C,B).
pman_1(A,B):-find_next_recipient(A,C),give_letter(C,B).
```

Program 10 (Metaopt$_{rc}$ pman strategy)

```
pman(A,B):-pman_2(A,C),pman_2(C,B).
pman_2(A,B):-pman_1(A,C),pman_2(C,B).
pman_2(A,B):-pman_1(A,C),go_start(C,B).
pman_1(A,B):-find_next_sender(A,C),bag_letter(C,B).
pman_1(A,B):-find_next_recipient(A,C),give_letter(C,B).
```

Both strategies handle any number of houses, any number of letters, and different start/end positions for the letters. However, although the strategies are equal in their textual complexity, they differ in their resource complexity. The strategy learned by Metaopt$_{rc}$ is more efficient than the one learned by Metagol because it uses the postbag to store letters.

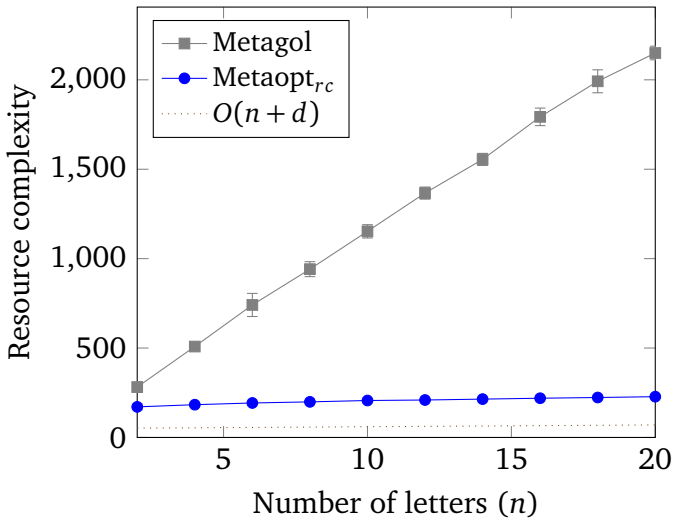


Figure 7.5: Mean resource complexity of learned postman strategies when varying the number of letters (n) for 50 places (d).

7.5.3 Learning robot sorting strategies

Materials Imagine a robot sorter where in the initial state there is an unsorted list of length d and in the final state there is a sorted list of length d. The robot can traverse the list moving sideways. The leftmost element represents the smallest element in the sorted list. We provide the robot with the actions to perform quick sort and bubble sort. We provide four complex actions: *compare_adjacent/2*, *split/2*, *combine/2*, and *go_start/2*. The action *compare/2* compares two adjacent elements and swaps them if the one to the right is smaller than the one to the left. The action *split/2* allows the robot to move through the list comparing each element with the element in the robot's left hand (the pivot). If an element is less than or equal to the pivot, then the item is placed in a left bag; otherwise it is placed in a right bag. Both bags are stacks.

The action *combine/2* empties the right bag, drops the pivot, and empties the left bag. The action *compare/2* has a cost of 1. All other actions have no cost. We now state the resource complexity of a minimal cost sorter strategy:

Proposition 6 (Minimal cost sorter strategy) Let d be the list length. Then the resource complexity of a minimal cost strategy is $O(log\ d)$.

Sketch proof 3 A minimal cost strategy involves the postman learning quick sort requiring (average case) $O(d\ log\ d)$ comparisons.

In addition, we use the *chain* and *tailrec* metarules.

Methods Examples are *sorter/2* atoms. To generate training examples we select a random integer d from the interval $[1, 25]$ representing the length of the list. We use Prolog's *randseq/3* predicate to generate a list of d unique random integers from the interval $[1, 100]$ representing the input list, with this list sorted representing the output list. To generate testing examples we repeat the aforementioned procedure but with a fixed list length d from the set $\{10, 20, \ldots, 100\}$ as to measure the resource complexity as d grows. We use 5 training and 10 testing examples. We measure mean resource complexities of learned strategies over 10 trials and limit the search to strategies of length four.

Results Figure 7.6 shows that Metaopt$_{rc}$ learns strategies with resource complexities that match the theoretical prediction (Proposition 6), whereas Metagol learns non-minimal strategies. We can explain these results by looking at the strategies learned by Metagol and Metaopt$_{rc}$:

Program 11 (Metagol sorter strategy)

```
sorter(A,B):-sorter_1(A,C),sorter(C,B).
sorter(A,B):-sorter_1(A,C),go_start(C,B).
sorter_1(A,B):-comp_adjacent(A,C),sorter(C,B).
sorter_1(A,B):-decrement_end(A,C),go_start(C,B).
```

Program 12 (Metaopt$_{rc}$ sorter strategy)

```
sorter(A,B):-sorter_1(A,C),sorter(C,B).
sorter(A,B):-decrement_end(A,C),combine(C,B).
sorter_1(A,B):-pick_up_left(A,C),split(C,B).
sorter_1(A,B):-combine(A,C),go_start(C,B).
```

Metagol learns a variation of bubble sort with a resource complexity close to the average-case expectations of bubble sort ($O(d^2)$). By contrast, Metaopt$_{rc}$ learns a variation of quicksort with a resource complexity $O(d\ log\ d)$.

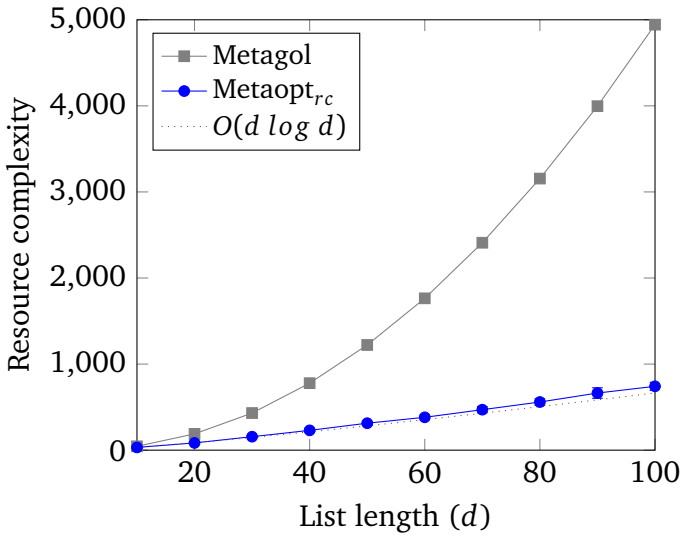


Figure 7.6: Mean resource complexity of robot sorter strategies when varying input lengths (d).

7.6 Future work

In the robot strategies considered, we have assumed positive action costs. In future work, we would like to consider negative action costs. i.e. benefits. For example, to consider an action which recharges the robot's battery, or actions in which the robot collects other resources, such as glue or bricks, or recruits other robots to help in a task.

Two of our experiments (postman and sorter) involve strategies where objects are composed by storing objects in containers (such as the postbag). In this setting, object composition increases the efficiency of the learned strategy. We intend to explore this notion of object composition and investigate whether inventing new objects can reduce the resource complexity of a problem. For instance, in the postman example, we provide a postbag in the background knowledge but we would like to develop methods for the learner to invent such

an object. One idea is to treat object invention as constant invention, and to automatically introduce new constant symbols to the constant signature, similar to how we introduce new predicate symbols (Section 3.2).

7.7 Summary

In this chapter, we have investigated learning minimal resource complexity robot strategies, described as logic programs. We have represented this problem as an instance of the cost minimisation problem. Our experiments show that Metaopt$_{rc}$ learns minimal resource complexity strategies, in contrast to Metagol, which learns non-minimal strategies. To our knowledge, this is the first demonstration of an algorithm proven to learn resource minimal logical strategies. This chapter supports Claim 2 of the thesis.

Chapter 8

Learning efficient logic programs

In this chapter, we support Claim 2 of this thesis by using Metaopt to learn minimal time complexity logic programs.

8.1 Introduction

Given a goal G, the time complexity of a logic program P is a function of the size of the SLD-tree searched to find an SLD-refutation of $P \cup \{G\}$. To learn minimal time complexity programs, we introduce a cost function called *tree complexity* which measures the size of a SLD-tree at the point of which a goal is proved by a logic program. The tree complexity of a logic program is different from resource complexity of a robot strategy because:

- Resource complexity only measures dyadic actions costs. By contrast, tree complexity takes into account the costs of non-dyadic predicates.
- Resource complexity measures the proof cost, and thus ignores costs involved in backtracking. By contrast, tree complexity ascribes costs to the whole search for a proof, and thus includes backtracking steps.
- Resource complexity requires that users provide resource costs as background knowledge. By contrast, tree complexity is defined in terms of the SLD-tree size of a goal with respect to a program, and thus does not require user-provided costs.

We support Claim 2 of this thesis by conducting experiments which show that Metaopt learns minimal tree complexity programs, which correspond to minimal time complexity programs.

8.2 Related work

Kaplan [54] described a method for estimating the average-case complexity of deterministic logic programs. However, in contrast to functional and imperative programs, logic programs can be non-deterministic, i.e. a logic program may return multiple solutions. Multiple solutions to a logic program are found by searching a SLD-tree for a SLD-refutation, and then backtracking to find other SLD-refutations. In [31], the authors introduced a semi-automatic method to estimate the worst-case complexity of deterministic and non-deterministic logic programs. However, this approach required meta-information, such as mode declarations and type information. In contrast to these approaches, we introduce a cost function which estimates the worst-case complexity of deterministic and non-deterministic logic programs by measuring the size of the SLD-tree searched to find a SLD-refutation of a goal, which does not require meta-information. In addition, the aforementioned approaches did not consider how to machine learn efficient programs.

In this chapter, we use iterative descent (Chapter 6) to learn minimal tree complexity programs. This approach reduces the hypothesis space by restricting the number resolutions allowed to find a hypothesis. Our approach is similar to one proposed by Blum and Blum [8] which Shapio [119] later adapted. Shapio used the notion of *h-easy* functions to limit the search for a hypothesis, where an atom A is *h-easy* with respect to a logic program P if there exists a derivation of A from P using at most h resolution steps. Shapiro's approach measures the number of resolutions in the derivation of A from P, and thus ignores backtracking steps, which is also the case with resource complexity. By contrast, our approach uses the notion of *tree complexity* to measure the total number of resolutions required to find a SLD-refutation of a goal with respect to a program, i.e. tree complexity includes backtracking steps.

8.3 Framework

In computer science, time complexity refers to the time an algorithm needs to perform some computation (Section 2.1). In logic programming, computation is formalised by means of SLD-resolution (Section 3.1.1). Given a logic program H and a goal G, computation involves finding a SLD-refutation of $H \cup \{G\}$. A SLD-refutation is found by searching a SLD-tree, which contains all possible SLD-derivations, and thus all possible SLD-refutations. Prolog searches for SLD-refutations using a depth-first search [129]. Therefore, we can measure the runtime (time complexity) of a Prolog program as a function of the size of the SLD-tree that it is being searched. Specifically, we measure the size of the leftmost branch of the SLD-tree in which the first SLD-refutation is found, i.e. the leftmost successful branch (Definition 10):

Definition 43 (Tree complexity) Let H be a logic program, G a goal, T a SLD-tree for $H \cup \{G\}$, and L be the leftmost successful branch of T. Then the tree complexity $\Phi_{tc}(H, G)$ is the number of resolutions prior to and including L within the depth-first enumeration of T.

We learn minimal tree complexity programs by representing the problem as a cost minimisation problem (Chapter 6). The input is the tuple $(B, E, \Phi_{tc}, \tau_c)$ where Φ_{tc} is as stated in Definiton 43 and τ_c is as stated in Definition 21. A solution to this cost minimisation problem is a minimal tree complexity program. Our experiments in Section 8.5 show that minimal tree complexity programs correspond to minimal time complexity programs.

8.4 Implementation

Metaopt assumes a program cost function (Definition 36) as background knowledge. To learn minimal tree complexity programs, we use the program cost function:

```
program_cost(Atom,Cost):-
    statistics(inferences,I1),
    call(Atom),
    statistics(inferences,I2),
    Cost is I2-I1-1.
```

This predicate uses a feature of SWI-Prolog to measure the number of logical inferences needed to prove a goal, where an inference is defined as a call or redo on a predicate[1]. In the experiments in this chapter, we supply Metaopt with this predicate. When supplied with this predicate, we refer to Metaopt as Metaopt$_{tc}$.

8.5 Experiments

We now describe three experiments to evaluate Metaopt$_{tc}$. To do so, we compare programs learned by Metaopt$_{tc}$, Metagol, and Metaopt$_{rc}$ (Metaopt supplied with the resource complexity function). To be clear, we compare three learning systems:

- **Metagol**
- **Metaopt$_{tc}$**: Metaopt supplied with the tree complexity cost function.
- **Metaopt$_{rc}$**: Metaopt supplied with the resource complexity cost function, where each dyadic background predicate has a cost of 1 and all non-dyadic predicates have no cost.

8.5.1 Experiment 1: convergence on minimal cost programs

This experiment revisits the *find duplicate* problem from Section 1. The aim is to see whether Metaopt converges on minimal cost programs given sufficient training examples (Theorem 6). In particular, we want to see how many examples are required in practice for convergence. We test the null hypothesis:

Null hypothesis 1 Metaopt cannot learn minimal cost programs without large numbers of training examples.

Materials To refute null hypothesis 1, we must identify a minimal cost program in the hypothesis space, which, in this experiment, is a minimal tree complexity program. We provide Metaopt$_{tc}$ with the *ident*, *chain*, and *tailrec* metarules (Figure 3.1) and four background predicates: *mergesort/2*, *tail/2*,

[1] http://www.swi-prolog.org/pldoc/man?predicate=statistics/2

head/2, and *element/2*. We limit the search to programs with four clauses. The minimal cost program in the hypothesis space is[2]:

Proposition 7 (Find duplicate minimal cost program) Let n be the list length. Then the tree complexity of a minimal cost program is $O(n\ log\ n)$.

Sketch proof 4 The minimal cost program involves first sorting the list and then passing through the list checking whether any two adjacent elements are the same. Thus the overall cost is $O(n\ log\ n)$.

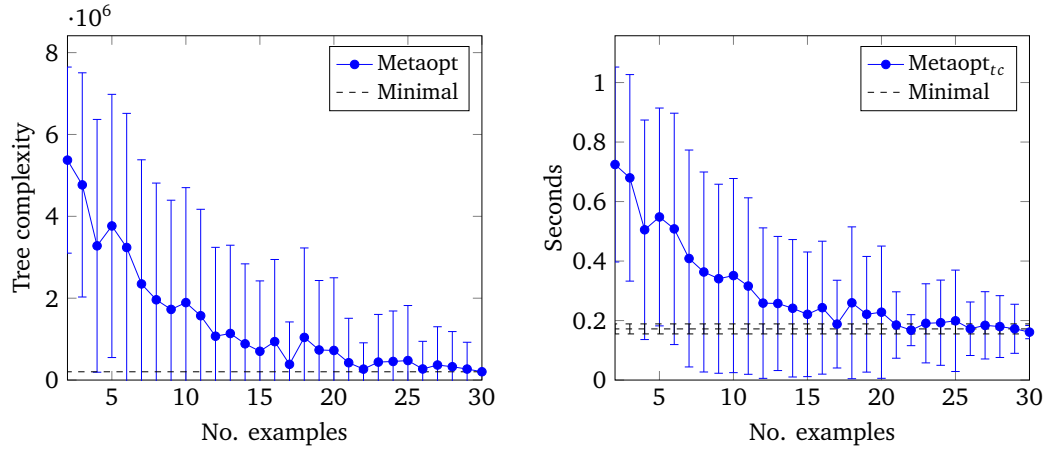
Although this result gives the order of the minimal cost program, the actual program will have many operations which affect the resulting cost. Therefore, we compare the results with the target minimal cost program:

```
f(A,B):-msort(A,C),f2(C,B).
f2(A,B):-f1(A,C),head(C,B).
f2(A,B):-tail(A,C),f1(C,B).
f1(A,B):-head(A,C),tail(C,B).
```

Method Examples are *fdup/2* atoms. We generate training examples as follows. For the first argument, the list which contains a duplicate, we select a random integer k from the interval $[5, 100]$ and generate an ordered sequence from $1 \ldots k$. We select a random integer j from the interval $[1, k]$ as the duplicate element and append it to the list. Finally, we randomly shuffle the list. The second argument is the duplicate element j. To generate testing examples we repeat the aforementioned procedure but for the fixed list size of 5000 to measure the learned program cost. We use m training examples from the interval $[1, 30]$ and 20 testing examples. We measure mean tree complexities and running times over 100 trials. We enforce a 10-minute timeout.

Results Figure 8.1a shows that Metaopt$_{tc}$ learns programs with lower costs given more training examples. After approximately 25 examples, Metaopt$_{tc}$ converges on the minimal cost program, refuting null hypothesis 1. Figure 8.1b shows similar results when measuring the runtimes of learned programs. These results show that the tree complexity of a program corresponds to the time complexity of a program.

[2]One could find the duplicate in time O(n) using a hash table but this solution is not in the hypothesis space, so could not be found by Metaopt.



(a) Tree complexities (b) Program runtimes

Figure 8.1: Figure (a) shows the tree complexities of programs learned by Metaopt$_{tc}$ when varying the number of training examples. After approximately 25 examples, Metaopt$_{tc}$ converges on the minimal cost program. Figure (b) shows the corresponding runtimes.

8.5.2 Experiment 2: comparison with other systems

This experiment again revisits the *find duplicate* problem from Section 1. The aim is to compare the efficiency of programs learned by Metaopt$_{tc}$, Metagol, and Metaopt$_{rc}$. We test the null hypothesis:

Null hypothesis 2 Metaopt$_{tc}$ cannot learn programs with lower costs and running times than Metagol and Metaopt$_{rc}$.

Materials We provide all three systems with the background knowledge used in Experiment 1.

Method Examples are *fdup/2* atoms. We generate training examples in the same way as in Experiment 1. To generate testing examples we repeat the aforementioned procedure but for fixed list sizes from the set $\{1000, 2000, \ldots, 10000\}$ to measure efficiency as the input grows. We use 20 training and 20 testing examples. We measure mean program costs and running times over 40 trials. We limit the search to programs of length four and enforce a 10-minute timeout.

Results The log-lin plots in Figure 8.2 show that Metaopt$_{tc}$ learns programs with lower tree complexities and lower runtimes than Metagol and Metaopt$_{rc}$. Therefore, null hypothesis 2 is refuted in terms of tree complexities and running times. There is a standard error on the Metaopt$_{tc}$ line because in one of the 40 trials, Metaopt$_{tc}$ learned a non-minimal cost program. In all the other trials, Metaopt$_{tc}$ learned the minimal cost program. Example programs learned by Metagol, Metaopt$_{rc}$, and Metaopt$_{tc}$ are shown below:

Program 13 (Metagol and Metaopt$_{rc}$ fdup program)

```
fdup(A,B):-head(A,C),fdup_1(C,B).
fdup(A,B):-tail(A,C),fdup(C,B).
fdup_1(A,B):-tail(A,C),member(B,C).
```

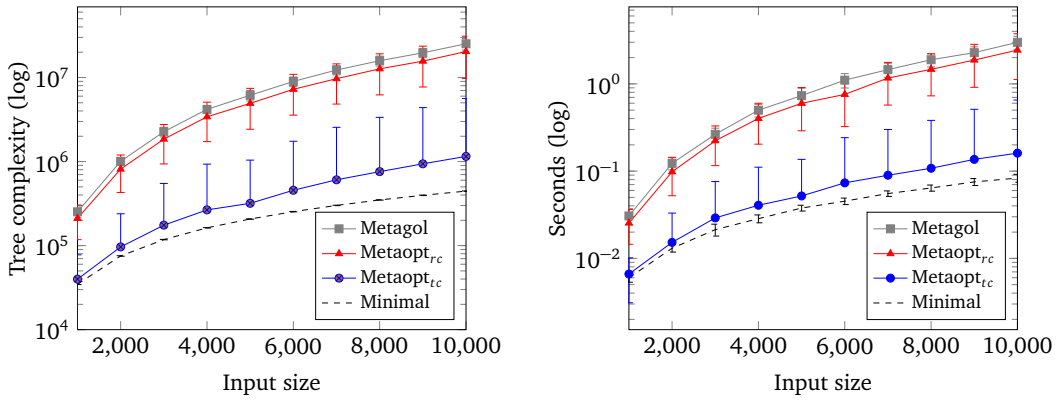
Program 14 (Metaopt$_{tc}$ fdup program)

```
fdup(A,B):-msort(A,C),fdup_1(C,B).
fdup_1(A,B):-head(A,C),fdup_2(C,B).
fdup_1(A,B):-tail(A,C),fdup_1(C,B).
fdup_2(A,B):-tail(A,C),head(C,B).
```

8.5.3 Experiment 3: string transformations

In [72] the authors evaluate Metagol on 17 real-world string transformation problems. Figure 8.3 shows problem *p01* where the goal is to learn a program to extract names from inputs. This experiment explores whether Metaopt$_{tc}$ can learn minimal cost programs for these problems.

Materials We provide Metaopt$_{tc}$, Metagol, and Metaopt$_{rc}$ with the *curry* and *chain* metarules (Figure 3.1) and the background predicates: *is_letter/1*, *not_letter/1*, *is_uppercase/1*, *not_uppercase/1*, *is_number/1*, *not_number/1*, *is_space/1*, *not_space/1*, *tail/2*, *dropLast/2*, *reverse/2*, *filter/3*, *dropWhile/3*, and *takeWhile/3*. The setup for Metaopt$_{tc}$, Metaopt$_{rc}$, and Metagol is the same as in Experiment 2, i.e. the aim is to learn programs with minimal tree complexities and thus minimal time complexities.



(a) Tree complexities (b) Program runtimes

Figure 8.2: Figure (a) shows the tree complexities of learned *fdup/2* programs. The costs of programs learned by Metaopt$_{tc}$ match those of the minimal cost program and are of order $O(nlogn)$. By contrast, the programs learned by Metagol and Metaopt$_{rc}$ are of order $O(n^2)$. Figure (b) shows the corresponding runtimes.

Input	**Output**
My name is John.	John
My name is Bill.	Bill
My name is Josh.	Josh
My name is Albert.	Albert
My name is Richard.	Richard

Figure 8.3: Examples for the *p01* string transformation problem.

Method The dataset from [72] contains five examples of each problem. We perform leave two out (keep three in) cross validation. We measure mean program costs and running times over all trials. We limit the search to programs with six clauses and enforce a 5-minute timeout.

Results Out of the 17 problems, Metagol, Metaopt$_{rc}$, and Metaopt$_{tc}$ learned different programs for 8 of them. Figure 8.4 shows the tree complexities for the 8 problems, where Metaopt$_{tc}$ learns programs with lower tree complexities in all cases, again refuting null hypothesis 2. For problem *p01*, the cost of the program learned by Metaopt$_{tc}$ (31) is half of that learned by Metagol (67) and Metaopt$_{rc}$ (86). Example programs learned by the systems for problem *p01*

are:

Example 19 (Metagol *p01* program)

```
p01(A,B):-tail(A,C),p01_1(C,B).
p01_1(A,B):-dropLast(A,C),p01_2(C,B).
p01_2(A,B):-dropWhile(A,B,not_uppercase).
```

Example 20 (Metaopt$_{rc}$ *p01* program)

```
p01(A,B):-p01_1(A,C),p01_4(C,B).
p01_1(A,B):-p01_2(A,C),p01_3(C,B).
p01_2(A,B):-filter(A,B,is_letter).
p01_3(A,B):-dropWhile(A,B,is_uppercase).
p01_4(A,B):-dropWhile(A,B,not_uppercase).
```

Example 21 (Metaopt$_{tc}$ *p01* program)

```
p01(A,B):-tail(A,C),p01_1(C,B).
p01_1(A,B):-p01_2(A,C),dropLast(C,B).
p01_2(A,B):-p01_3(A,C),p01_3(C,B).
p01_3(A,B):-tail(A,C),p01_4(C,B).
p01_4(A,B):-p01_5(A,C),p01_5(C,B).
p01_5(A,B):-tail(A,C),tail(C,B).
```

Although textually more complex, the program learned by Metaopt$_{tc}$ is more efficient because it successively applies the *tail/2* predicate until it reaches the first letter of the name. By contrast, Metagol learns a program which uses the higher-order *dropWhile/3* predicate to recursively check whether the head symbol is uppercase, and if not it drops the head element, which requires twice the work. Because Metaopt$_{rc}$ only associates costs with dyadic predicates, it found a program which does not directly use any primitive dyadic predicates, and thus has a resource cost of 0, yet is less efficient than the one found by Metaopt$_{tc}$.

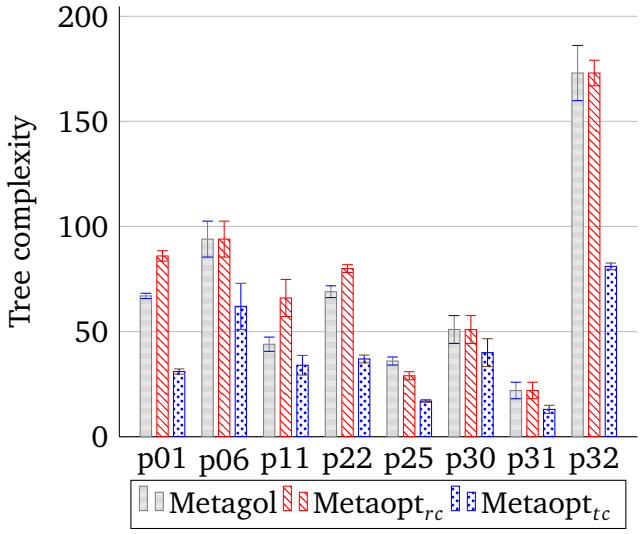


Figure 8.4: Tree complexities of learned string transformation programs where Metaopt$_{tc}$ learns more efficient programs than Metagol and Metaopt$_{rc}$ in all cases.

8.6 Future work

Our result that Metaopt learns minimal cost programs (Theorem 6) assumes sufficient training examples. The *find duplicate* experiment showed that Metaopt can learn minimal cost programs with small numbers of examples. However, future work should further test this claim on other domains, such as learning from visual data. We also want to see whether Metaopt can learn minimal space complexity programs.

Metaopt$_{tc}$ calculates the tree complexity of a program during the learning. To reduce learning times, we would like to investigate approximating the tree complexity by approximating the SLD-tree size [58].

8.7 Summary

In this chapter, we have investigated learning minimal tree complexity programs, and thus minimal time complexity programs. We have introduced a program cost function called tree complexity (Definition 43), which is based on the size of a SLD-tree at the point of which a goal is proved by a logic program. Our

experiments on the *find duplicate* problem and real-world string transformations show that given small numbers of examples, Metaopt$_{tc}$ learns minimal tree complexity programs, and thus minimal time complexity programs. To our knowledge, this is the first demonstration of an algorithm proven to learn efficient logic programs. This chapter supports Claim 2 of the thesis.

Chapter 9

Conclusions and future work

We now conclude by reviewing the contributions and discussing future work.

9.1 Conclusions

We started this thesis by claiming:

- **Claim 1:** we can *efficiently* machine learn programs
- **Claim 2:** we can machine learn *efficient* programs

To support these claims, we have introduced techniques to efficiently learn efficient programs. Specifically, we have made the following contributions:

Contribution 1: metarules A MIL learner takes metarules as part of the background knowledge. In Chapter 4, we explained that selecting which metarules to use is a trade-off between efficiency and expressivity: the hypothesis space increases given more metarules, so we wish to use fewer metarules, but if we use too few metarules then we lose expressivity. To alleviate this trade-off, we used Plotkin's reduction algorithm to logically reduce sets of metarules. Using this approach, we found that only two metarules are necessary to entail all metarules in the H^{2*}_m fragment of logic. Our experiments showed that, compared to learning with non-minimal sets of metarules, learning with minimal sets of metarules improves predictive accuracies and reduces learning times. This contribution supports Claim 1.

Contribution 2: higher-order programs In addition to metarules, a MIL learner takes clauses as part of the background knowledge. In previous work on MIL, and most work on ILP, these clauses were first-order, and were used to learn first-order programs. In Chapter 5, we extended MIL to support learning higher-order programs by using higher-order definitions as background knowledge. We showed that learning higher-order programs reduces the textual complexity required to express target classes of programs which in turn reduces the hypothesis space. Our sample complexity results show that learning higher-order programs reduces (1) the number of examples required to reach high predictive accuracies, and (2) learning times (Proposition 3). To learn higher-order programs, we introduced Metagol$_{AI}$, a MIL learner which also supports higher-order predicate invention, such as inventing predicates for the higher-order abstractions *map/3* and *reduce/4*. Our experiments showed that, compared to learning first-order programs, learning higher-order programs improves predictive accuracies by up to 50% and reduces learning times by four orders of magnitude. This contribution supports Claim 1.

Contribution 3: cost minimisation problem In Chapter 6, we introduced the *cost minimisation problem*, a framework for learning efficient programs, where specific program cost functions are provided as input. We also introduced Metaopt, a MIL learner which solves the cost minimisation problem. Metaopt uses a new search procedure called *iterative descent* to iteratively search for more efficient (lower cost) programs, each time further restricting the hypothesis space. We showed that, given sufficient training examples, Metaopt converges on minimal cost programs (Theorem 6). This contribution supports Claim 2.

Contribution 4: learning efficient robot strategies In Chapter 7, we used Metaopt to learn minimal resource complexity robot strategies, where resource complexity is a user-defined measure of the resources necessary for a robot to perform certain actions. Our experiments showed that, whereas Metagol learns non-minimal cost strategies, Metaopt learns minimal cost strategies. For instance, the sorter experiment (Section 7.5.3) showed that Metaopt learns an efficient quicksort strategy, whereas Metagol learns an inefficient bubble sort strategy. This contribution supports Claim 2.

Contribution 5: learning efficient logic programs In Chapter 8, we used Metaopt to learn minimal time complexity logic programs. To learn such

programs, we introduced a cost function called *tree complexity* which is based on the size of a SLD-tree at the point of which a goal is proved by a logic program. In contrast to resource complexity, tree complexity takes into account backtracking when measuring the cost of a program, which is necessary to measure the efficiency of non-deterministic programs. In addition, because tree complexity is based on the size of a SLD-tree, user-defined cost functions are not required. Our experiments showed that when supplied with this cost function, Metaopt learns minimal tree complexity programs which correspond to minimal time complexity programs, such as learning efficient programs for programming puzzles and real-world string transformations. This contribution supports Claim 2.

9.2 Future work

We now outline potential future work to address limitations of this thesis.

9.2.1 Background knowledge

Metarules In Chapter 4, we used logical minimisation techniques to show that only two metarules are necessary to entail all metarules in the H^{2*}_m fragment of logic, and that using these two metarules improves learning performance compared to using more metarules. However, we have not used this minimal set in the rest of this thesis.

One reason for not using this minimal set is that it is restricted to dyadic logic programs, whereas in the rest of the thesis we have used non-dyadic background knowledge, such as the predicates *is_uppercase/1* and *is_number/1* when learning string transformations (Chapter 8). Although the class of dyadic logic programs with one function symbol has Universal Turing Machine (UTM) expressivity [132], some concepts are not easily expressed in this class, such as when learning triadic target predicates. In the case of learning triadic predicates, one would need to change the representation to compose multiple terms into compound terms, and similarly decompose terms into multiple terms. Such a representation change would likely increase the size of the target program, which would increase the size of hypothesis space (Theorem 1 and Lemma 2) which would in turn lead to longer learning times. An increase in program size may also lead to a reduction in the readability of the program [117].

In future work, we want to use similar techniques to those used in Chapter 4 to reduce broader classes of metarules, such as those containing monadic and triadic predicates, which preliminary work suggests is possible. By having a complete (i.e. sufficient to entail all hypotheses) set of metarules, we would also like the prove both the soundness and completeness of Metagol, both of which should follow from the soundness and refutation completeness of SLD-resolution [74].

Another reason for not using the minimal set is learning efficiency. In Chapter 4, our experiments showed that learning with minimal sets of metarules reduces learning times compared to learning with the maximal sets of metarules. We also explored randomly sampling metarules from the maximum set, where the experiments again showed no benefit in using more metarules. However, this is not always the case. For example, when Metagol was used to learn dyadic string transformations [72], the learned programs only used the *chain* metarule. In this case, the optimal set of metarules is a subset of the minimal set. In future work, we want to investigate learning optimal sets of metarules. Such an approach could use never-ending learning [82] to count the frequency of use of metarules. We could then select metarules based on their frequency of use.

By combining a never-ending learning with dependent learning [72], we could also learn metarules over time. In this approach, we could supply Metagol with a full enumeration of metarules for a fixed language. In this approach, we could supply Metagol with a full enumeration of metarules for a fixed language. We could then learn relatively simple clauses, which we could unfold to derive new clauses, which could be uninstantiated to generate metarules. For instance, reconsider the learned string transformation program from Section 8.5.3:

```
p01(A,B):-tail(A,C),p01_1(C,B).
p01_1(A,B):-dropLast(A,C),p01_2(C,B).
p01_2(A,B):-dropWhile(A,B,not_uppercase).
```

Having learned this program, Metagol could unfold the program to remove unnecessary invented predicates to form the new clause:

```
p01(A,B):-tail(A,C),dropLast(C,D),dropWhile(D,B,not_uppercase).
```

We could then uninstantiate the bound variables to form the metarule:

```
P(A,B):-Q(A,C),R(C,D),S(D,B,F).
```

Background knowledge reduction The hypothesis space of a MIL learner is a function of the number of metarules and the number of background predicates, i.e. the size of the background knowledge (Theorems 1 and Lemma 2). In Chapter 4, we used Plotkin's reduction algorithm to find redundant metarules. In future work, we want to see whether we could use similar techniques to minimise background knowledge with respect to metarules. We briefly mentioned this form of reduction in Section 4.5.2 when learning robot strategies. Specifically, we described the case when we could remove the *move_right/2* predicate from the background knowledge, which Metagol could replace by combining the *inverse* metarule with a *move_left/2* predicate. This idea of purposely removing background predicates is analogous to dimensionality reduction, widely used in other forms of machine learning [122], but which has been under used in ILP [41]. Our preliminary experiments seem to indicate that this reduction is possible.

Higher-order definitions In Chapter 5, we introduced Metagol$_{AI}$ which supports learning higher-order programs by using higher-order definitions as background knowledge. This approach led to the invention of functional constructs for use in higher-order predicates, such as *map/3* and *until/4*. In future work, we want to investigate the use of relational constructs. For instance, consider this higher-order definition of a closure:

closure(P,A,B) ← P(A,B)
closure(P,A,B) ← P(A,C), closure(P,C,B)

We could use this definition to learn compact abstractions of relations, such as:

ancestor(A,B) ← closure(parent,A,B)
lessthan(A,B) ← closure(increment,A,B)
subterm(A,B) ← closure(headortail,A,B)

Moreover, the issue of how metarules might themselves be learned could be treated in a similar fashion using higher-order programs, such as:

chain(P,Q,R,A,B) ← Q(A,C), R(C,B)
inverse(P,Q,A,B) ← Q(B,A)

Acquisition of background knowledge As with most work in ILP and program induction, we have assumed background knowledge as input. However, as with determining which metarules to use, determining which background knowledge to use is non-trivial. If given insufficient background knowledge, then Metagol can potentially invent the necessary background knowledge [22], such as inventing missing kinship relations (Section 4.5.1) or missing robot actions (Section 4.5.2).

However, reinventing missing background knowledge may be intractable. For instance, when learning the *find duplicate* program in Section 8.5.1, we provided the *msort/2* predicate as background knowledge. If not given this predicate, then Metagol could potentially learn it, but would require learning a larger program. In the extreme case of being given no background knowledge, and thus no inductive bias, Metagol, as with any program induction approach, would be equivalent to universal induction methods, and the learning would thus be intractable.

To avoid the intractability of learning with no background knowledge, ILP has traditionally relied on domain experts to craft the background knowledge, which is a problem because (1) background knowledge can be difficult or expensive to obtain, and (2) if the domain changes or shifts, then the background knowledge must be altered. To address this limitation, future work should explore learning background knowledge over time. This idea was initially explored in [72], where the authors used Metagol to learn a collection of string transformation programs, whereby learned programs are saved so that they can be reused in future learning. This approach allowed for Metagol to learn large and complex programs by first learning programs for sub-programs of tasks. However, although this approach worked in the short-term, in the long-term the learning efficiency would decline because the size of the background knowledge would grow monotonically. FOIL suffered from similar issues when it was used to learn programs over time [106]. Therefore, to efficiently learn over time, a program induction system must be able to forget background knowledge, which is a topic for future work.

9.2.2 Efficient programs

Efficiency vs accuracy trade-off Theorem 6 shows that Metaopt learns minimal cost programs given sufficient training examples. The *find duplicate* experiment (Section 8.5.1) supported this result and showed that Metaopt can learn minimal cost programs with small numbers of examples (<25). Future

work should further test this result on other domains, such as learning from visual data. We also want to use Metaopt to learn minimal space complexity programs.

Program complexity analysis Metaopt uses iterative descent to continually reduce the hypothesis space to prune programs that are less efficient than already learned ones. However, this approach is inefficient when the first found program (in the first iteration of iterative descent) has a prohibitively high cost. For instance, suppose you are learning to sort lists and that the shortest program in the hypothesis space is permutation sort. Then in the first iteration of iterative descent, Metaopt would find permutation sort, which would require $O(n!)$ time. If the examples are large, then this approach would be impractical. To overcome this issue, iterative descent could start with a low program cost bound and then iteratively relax this bound until the first program is found. Once a program has been found, iterative descent could then work as it does now and search for more efficient programs by continually restricting the hypothesis space. Alternatively, we could estimate the tree complexity of a program by approximating the SLD-tree size [58].

Algorithm discovery We have used Metaopt to learn efficient programs, such as an efficient quicksort robot strategy and an efficient *find duplicate* program. However, although the learning techniques are novel, the learned programs are not, i.e. we have learned programs that we already knew about. In future work, we want to use Metaopt for *algorithm discovery*, where the goal is to learn programs that are useful and novel. One criterion to determine whether a learned program is useful and novel is whether it is worthy of publication in an algorithms journal. In other words, we want to use Metaopt to discover a publishable algorithm.

9.2.3 Meta-interpretive learning

We now discuss the limitations of using MIL for program induction.

Program size In our experiments, we have learned complex but compact programs, which rarely contain more than 6 clauses. As shown in Theorem 1 and Lemma 2, the hypothesis space of a MIL learner is exponential in the size of the target program, which explains the difficulty in learning programs with

a large number of clauses (e.g. more than 10). To overcome this limitation, Metagol could use a greedy search strategy, which was briefly explored in [97], in which Metagol finds the smallest program that covers one example and adds the program to the background knowledge which can then be used to help cover the rest of the examples. Because fewer clauses need to be learned for each example, the search space is smaller than when performing non-greedy learning. This greedy approach was shown to reduce learning times by two orders of magnitude. However, this approach could lead to overly specific programs. In the worst-case, this approach could learn a specific program for each example, offering no generalisation. The aforementioned greedy approach does not help in the case of learning from one example. In this case, one approach is to decompose the single example into granular parts, and to then greedily learn programs for the individual parts, which preliminary work suggests is possible.

Noise handling In all of our experiments, we have assumed noise-free examples, which means that a learned program must be consistent with all examples. This assumption restricts MIL from being applied to noisy problems. To address this limitation, we could relax the requirement that a program must be consistent with all examples. One idea is to repeatedly learn programs from random subsets of the examples, and to then calculate confidence levels of the learned programs based on the size of the subsets and the number of repetitions.

Probabilities We have also assumed noise-free background knowledge. For instance, in Chapter 7, we assumed that robot actions always succeed. However, real-world robotic systems are fallible, and actions may be probabilistic. To overcome this limitation, we would need a probabilistic MIL setting. In contrast to MetaBayes [95], a version of MIL that assigns probabilities to hypotheses, we would need a probabilistic version of MIL that assigns probabilities to clauses in a program. Such an approach could be based on stochastic logic programs [91] or Problog [110].

Negation We have learned definite programs. However, definite programs are not expressive enough to represent incomplete knowledge [116]. In addition, we sometimes want to learn programs with negative literals, i.e. normal programs [99]. For instance, in Chapter 5, we used Metagol$_{AI}$ to learn higher-order programs by using higher-order definitions as background knowledge. One such definition was *until/4* (Example 12) in which the second clause contained

negation in the body. We used negation as failure to learn programs which used this definition. However, our approach is limited because we can only negate already defined predicates, and thus we cannot invent predicates to be used as conditions in *until/4*. Therefore, future work should explore adding negation into MIL.

9.3 Summary

To conclude, the techniques introduced in this thesis open new avenues of research in computer science and raise the potential for algorithm designers to discover novel efficient algorithms, for software engineers to automate the building of efficient software, and for AI researchers to machine learn efficient robot strategies.

Bibliography

[1] Hilde Adé, Luc De Raedt, and Maurice Bruynooghe. Declarative bias for specific-to-general ILP systems. *Machine Learning*, 20(1-2):119–154, 1995.

[2] Alfred V Aho and Jeffrey D Ullman. *Foundations of computer science*. Computer Science Press, 1992.

[3] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 934–950, 2013.

[4] Duangtida Athakravi, Domenico Corapi, Krysia Broda, and Alessandra Russo. Learning through hypothesis refinement using answer set programming. In Gerson Zaverucha, Vítor Santos Costa, and Aline Paes, editors, *Inductive Logic Programming - 23rd International Conference, ILP 2013, Rio de Janeiro, Brazil, August 28-30, 2013, Revised Selected Papers*, volume 8812 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2013.

[5] Margherita Berardi and Donato Malerba. Learning recursive patterns for biomedical information extraction. In *Inductive Logic Programming, 16th International Conference, ILP 2006, Santiago de Compostela, Spain, August 24-27, 2006, Revised Selected Papers*, pages 79–93, 2006.

[6] Hendrik Blockeel, Luc Dehaspe, Bart Demoen, Gerda Janssens, Jan Ramon, and Henk Vandecasteele. Improving the efficiency of inductive logic programming through the use of query packs. *J. Artif. Intell. Res.*, 16:135–166, 2002.

[7] Hendrik Blockeel, Luc De Raedt, Nico Jacobs, and Bart Demoen. Scaling up inductive logic programming by learning from interpretations. *Data Min. Knowl. Discov.*, 3(1):59–93, 1999.

[8] Lenore Blum and Manuel Blum. Toward a mathematical theory of inductive inference. *Information and Control*, 28(2):125–155, 1975.

[9] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Learnability and the vapnik-chervonenkis dimension. *J. ACM*, 36(4):929–965, 1989.

[10] Ivan Bratko. Discovery of abstract concepts by a robot. In *Discovery Science - 13th International Conference, DS 2010, Canberra, Australia, October 6-8, 2010. Proceedings*, pages 372–379, 2010.

[11] Ivan Bratko and Donald Michie. A representation for pattern-knowledge in chess endgames. *Advances in Computer Chess*, 2:31–56, 1980.

[12] Richard P. Brent. Recent progress and prospects for integer factorisation algorithms. In Ding-Zhu Du, Peter Eades, Vladimir Estivill-Castro, Xuemin Lin, and Arun Sharma, editors, *Computing and Combinatorics, 6th Annual International Conference, COCOON 2000, Sydney, Australia, July 26-28, 2000, Proceedings*, volume 1858 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2000.

[13] Krysia Broda, Keith Clark, Rob Miller, and Alessandra Russo. *SAGE: a logical agent-based environment monitoring and control system*. Springer, 2009.

[14] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, 1985.

[15] Gregory J. Chaitin. Algorithmic information theory. *IBM Journal of Research and Development*, 21(4):350–359, 1977.

[16] Alonzo Church. A note on the entscheidungsproblem. *J. Symb. Log.*, 1(1):40–41, 1936.

[17] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[18] K.L. Clark. Negation as failure. In M. L. Ginsberg, editor, *Readings in Nonmonotonic Reasoning*, pages 311–325. Kaufmann, Los Altos, CA, 1987.

[19] William W Cohen. Rapid prototyping of ilp systems using explicit bias. In *Proceedings of the 1993 IJCAI Workshop on Inductive Logic Programming, Chambery, France*, 1993.

[20] William W. Cohen. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artif. Intell.*, 68(2):303–366, 1994.

[21] Domenico Corapi, Alessandra Russo, and Emil Lupu. Inductive logic programming as abductive search. In Manuel V. Hermenegildo and Torsten Schaub, editors, *Technical Communications of the 26th International Conference on Logic Programming, ICLP 2010, July 16-19, 2010, Edinburgh, Scotland, UK*, volume 7 of *LIPIcs*, pages 54–63. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[22] Andrew Cropper and Stephen Muggleton. Can predicate invention compensate for incomplete background knowledge? In Slawomir Nowaczyk, editor, *Thirteenth Scandinavian Conference on Artificial Intelligence - SCAI 2015, Halmstad, Sweden, November 5-6, 2015*, volume 278 of *Frontiers in Artificial Intelligence and Applications*, pages 27–36. IOS Press, 2015.

[23] Andrew Cropper and Stephen H. Muggleton. Logical minimisation of meta-rules within meta-interpretive learning. In Jesse Davis and Jan Ramon, editors, *Inductive Logic Programming - 24th International Conference, ILP 2014, Nancy, France, September 14-16, 2014, Revised Selected Papers*, volume 9046 of *Lecture Notes in Computer Science*, pages 62–75. Springer, 2014.

[24] Andrew Cropper and Stephen H. Muggleton. Learning efficient logical robot strategies involving composable objects. In Qiang Yang and Michael Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 3423–3429. AAAI Press, 2015.

[25] Andrew Cropper and Stephen H. Muggleton. Learning higher-order logic programs through abstraction and invention. In Subbarao Kambhampati,

editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 1418–1424. IJCAI/AAAI Press, 2016.

[26] Andrew Cropper and Stephen H. Muggleton. Metagol system. https://github.com/metagol/metagol, 2016.

[27] Andrew Cropper and Stephen H. Muggleton. Learning efficient logic programs. *Machine Learning*, 2018. In Press.

[28] Andrew Cropper, Alireza Tamaddoni-Nezhad, and Stephen H. Muggleton. Meta-interpretive learning of data transformation programs. In Katsumi Inoue, Hayato Ohwada, and Akihiro Yamamoto, editors, *Inductive Logic Programming - 25th International Conference, ILP 2015, Kyoto, Japan, August 20-22, 2015, Revised Selected Papers*, volume 9575 of *Lecture Notes in Computer Science*, pages 46–59. Springer, 2015.

[29] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[30] Alexander Munro Davie and Andrew James Stothers. Improved bound for complexity of matrix multiplication. *Proceedings of the Royal Society of Edinburgh: Section A Mathematics*, 143(02):351–369, 2013.

[31] Saumya K. Debray, Pedro López-García, Manuel V. Hermenegildo, and Nai-Wei Lin. Lower bound cost estimation for logic programs. In *Logic Programming, Proceedings of the 1997 International Symposium, Port Jefferson, Long Island, NY, USA, October 13-16, 1997*, pages 291–305, 1997.

[32] Luc Dehaspe and Luc De Raedt. Parallel inductive logic programming. In *Proceedings of the MLnet familiarization workshop on statistics, machine learning and knowledge discovery in databases*, volume 1, page 5, 1995.

[33] Sašo Džeroski, Luc De Raedt, and Kurt Driessens. Relational reinforcement learning. *Machine learning*, 43(1-2):7–52, 2001.

[34] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. Answer set planning under action costs. *J. Artif. Intell. Res.*, 19:25–71, 2003.

[35] Werner Emde, Christopher Habel, and Claus-Rainer Rollinger. The discovery of the equator or concept driven learning. In Alan Bundy, editor, *Proceedings of the 8th International Joint Conference on Artificial Intelligence. Karlsruhe, FRG, August 1983*, pages 455–458. William Kaufmann, 1983.

[36] Colin Farquhar, Gudmund Grov, Andrew Cropper, Stephen Muggleton, and Alan Bundy. Typed meta-interpretive learning for proof strategies. In Katsumi Inoue, Hayato Ohwada, and Akihiro Yamamoto, editors, *Late Breaking Papers of the 25th International Conference on Inductive Logic Programming, Kyoto University, Kyoto, Japan, August 20th to 22nd, 2015.*, volume 1636 of *CEUR Workshop Proceedings*, pages 17–32. CEUR-WS.org, 2015.

[37] Cao Feng and Stephen Muggleton. Towards inductive generalization in higher order logic. In Derek H. Sleeman and Peter Edwards, editors, *Proceedings of the Ninth International Workshop on Machine Learning (ML 1992), Aberdeen, Scotland, UK, July 1-3, 1992*, pages 154–162. Morgan Kaufmann, 1992.

[38] Andreas Fidjeland, Wayne Luk, and Stephen Muggleton. Scalable acceleration of inductive logic programs. In *Proceedings of the 2002 IEEE International Conference on Field-Programmable Technology, FPT 2002, Hong Kong, China, December 16-18, 2002*, pages 252–259, 2002.

[39] Pierre Flener and Serap Yilmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. *J. Log. Program.*, 41(2-3):141–195, 1999.

[40] Martin Fürer. Faster integer multiplication. *SIAM J. Comput.*, 39(3):979–1005, 2009.

[41] Johannes FÃijrnkranz. Dimensionality reduction in ilp: A call to arms. In *Proceedings of the IJCAI-97 Workshop on Frontiers of Inductive Logic Programming*, pages 81–86, 1997.

[42] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A userâĂŹs guide to gringo, clasp, clingo, and iclingo. 2008.

[43] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[44] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 317–330, 2011.

[45] Sumit Gulwani. Applications of program synthesis to end-user programming and intelligent tutoring systems. In *Genetic and Evolutionary Computation Conference, GECCO '14, Vancouver, BC, Canada, July 12-16, 2014, Companion Material Proceedings*, pages 5–6, 2014.

[46] Sumit Gulwani. Example-based learning in computer-aided STEM education. *Commun. ACM*, 57(8):70–80, 2014.

[47] Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H. Muggleton, Ute Schmid, and Benjamin G. Zorn. Inductive programming meets the real world. *Commun. ACM*, 58(11):90–99, 2015.

[48] Larry Harris. The heuristic search and the game of chess. a study of quiescence, sacrifices, and plan oriented play. In *Computer Chess Compendium*, pages 136–142. Springer, 1988.

[49] David Hilbert and Wilhelm Ackermann. *Principles of mathematical logic*, volume 69. American Mathematical Soc., 1950.

[50] Geoffrey E Hinton. Learning distributed representations of concepts. In *Proceedings of the eighth annual conference of the cognitive science society*, volume 1, page 12. Amherst, MA, 1986.

[51] Jörg Hoffmann. The metric-ff planning system: Translating "ignoring delete lists" to numeric state variables. *J. Artif. Intell. Res.*, 20:291–341, 2003.

[52] Marcus Hutter. Universal learning theory. In *Encyclopedia of Machine Learning and Data Mining*, pages 1295–1304. 2017.

[53] Katsumi Inoue, Andrei Doncescu, and Hidetomo Nabeshima. Completing causal networks by meta-level abduction. *Machine Learning*, 91(2):239–277, 2013.

[54] S. Kaplan. Algorithmic complexity of logic programs. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes)*, pages 780–793. MIT Press, 1988.

[55] Susumu Katayama. Efficient exhaustive generation of functional programs using monte-carlo search with iterative deepening. In *PRICAI 2008: Trends in Artificial Intelligence, 10th Pacific Rim International Conference on Artificial Intelligence, Hanoi, Vietnam, December 15-19, 2008. Proceedings*, pages 199–210, 2008.

[56] Piyush Khandelwal, Fangkai Yang, Matteo Leonetti, Vladimir Lifschitz, and Peter Stone. Planning in action language BC while learning action costs for mobile robots. 2014.

[57] Jörg-Uwe Kietz and Stefan Wrobel. Controlling the complexity of learning in logic through syntactic and task-oriented models. In *Inductive logic programming*. Citeseer, 1992.

[58] Philip Kilby, John K. Slaney, Sylvie Thiébaux, and Toby Walsh. Estimating search tree size. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, pages 1014–1019, 2006.

[59] Tim Kimber, Krysia Broda, and Alessandra Russo. Induction on failure: Learning connected horn theories. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings*, volume 5753 of *Lecture Notes in Computer Science*, pages 169–181. Springer, 2009.

[60] Emanuel Kitzelmann. Data-driven induction of functional programs. In *ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings*, pages 781–782, 2008.

[61] S.C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112:727–742, 1936.

[62] Andrei N Kolmogorov. Three approaches to the quantitative definition of information. *Problems of information transmission*, 1(1):1–7, 1965.

[63] Robert A. Kowalski and Donald Kuehner. Linear resolution with selection function. *Artif. Intell.*, 2(3/4):227–260, 1971.

[64] John E. Laird. Extending the soar cognitive architecture. pages 224–235, 2008.

[65] J. Larson and Ryszard S. Michalski. Inductive inference of VL decision rules. *SIGART Newsletter*, 63:38–44, 1977.

[66] Tessa A. Lau, Pedro M. Domingos, and Daniel S. Weld. Learning programs from traces using version space algebra. In *Proceedings of the 2nd International Conference on Knowledge Capture (K-CAP 2003), October 23-25, 2003, Sanibel Island, FL, USA*, pages 36–43, 2003.

[67] Mark Law, Alessandra Russo, and Krysia Broda. Inductive learning of answer set programs. In *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*, pages 311–325, 2014.

[68] Gregor Leban, Jure Zabkar, and Ivan Bratko. An experiment in robot discovery with ILP. In *Inductive Logic Programming, 18th International Conference, ILP 2008, Prague, Czech Republic, September 10-12, 2008, Proceedings*, pages 77–90, 2008.

[69] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[70] Leonid Anatolevich Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973.

[71] Henry Lieberman. Your wish is my command: Programming by example, 2001.

[72] Dianhuan Lin, Eyal Dechter, Kevin Ellis, Joshua B. Tenenbaum, and Stephen 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)*, pages 525–530, 2014.

[73] John W. Lloyd. Practical advtanages of declarative programming. In *1994 Joint Conference on Declarative Programming, GULP-PRODE'94 Peñiscola, Spain, September 19-22, 1994, Volume 1*, pages 18–30, 1994.

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

[75] J.W. Lloyd. *Logic for Learning*. Springer, Berlin, 2003.

[76] Alain-Pierre Manine, Érick Alphonse, and Philippe Bessières. Extraction of genic interactions with the recursive logical theory of an ontology. In *Computational Linguistics and Intelligent Text Processing, 11th International Conference, CICLing 2010, Iasi, Romania, March 21-27, 2010. Proceedings*, pages 549–563. 2010.

[77] Zohar Manna and Richard J. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.

[78] John McCarthy. *Programs with common sense*. RLE and MIT Computation Center, 1960.

[79] John McCarthy. Making robots conscious of their mental states. In *Machine Intelligence 15, Intelligent Agents [St. Catherine's College, Oxford, July 1995]*, pages 3–17, 1995.

[80] Donald Michie. Machine learning in the next five years. In Derek H. Sleeman, editor, *Proceedings of the Third European Working Session on Learning, EWSL 1988, Turing Institute, Glasgow, UK, October 3-5, 1988*, pages 107–122. Pitman Publishing, 1988.

[81] Tom M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997.

[82] Tom M. Mitchell, William W. Cohen, Estevam R. Hruschka Jr., Partha Pratim Talukdar, Justin Betteridge, Andrew Carlson, Bhavana Dalvi Mishra, Matthew Gardner, Bryan Kisiel, Jayant Krishnamurthy, Ni Lao, Kathryn Mazaitis, Thahir Mohamed, Ndapandula Nakashole, Emmanouil Antonios Platanios, Alan Ritter, Mehdi Samadi, Burr Settles, Richard C. Wang, Derry Tanti Wijaya, Abhinav Gupta, Xinlei Chen, Abulhair Saparov, Malcolm Greaves, and Joel Welling. Never-ending learning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 2302–2310, 2015.

[83] Stephen Moyle. Using theory completion to learn a robot navigation control program. In Stan Matwin and Claude Sammut, editors, *Inductive*

Logic Programming, 12th International Conference, ILP 2002, Sydney, Australia, July 9-11, 2002. Revised Papers, volume 2583 of *Lecture Notes in Computer Science*, pages 182–197. Springer, 2002.

[84] Stephen Moyle and Stephen Muggleton. Learning programs in the event calculus. In Nada Lavrac and Saso Dzeroski, editors, *Inductive Logic Programming, 7th International Workshop, ILP-97, Prague, Czech Republic, September 17-20, 1997, Proceedings*, volume 1297 of *Lecture Notes in Computer Science*, pages 205–212. Springer, 1997.

[85] Stephen Moyle and Stephen Muggleton. Learning programs in the event calculus. In *Inductive Logic Programming, 7th International Workshop, ILP-97, Prague, Czech Republic, September 17-20, 1997, Proceedings*, pages 205–212, 1997.

[86] Stephen Muggleton. Inductive logic programming. *New Generation Comput.*, 8(4):295–318, 1991.

[87] Stephen Muggleton. Inverse entailment and progol. *New Generation Comput.*, 13(3&4):245–286, 1995.

[88] Stephen Muggleton. Alan turing and the development of artificial intelligence. *AI Commun.*, 27(1):3–10, 2014.

[89] Stephen Muggleton and Christopher H. Bryant. Theory completion using inverse entailment. In James Cussens and Alan M. Frisch, editors, *Inductive Logic Programming, 10th International Conference, ILP 2000, London, UK, July 24-27, 2000, Proceedings*, volume 1866 of *Lecture Notes in Computer Science*, pages 130–146. Springer, 2000.

[90] Stephen Muggleton and Wray L. Buntine. Machine invention of first order predicates by inverting resolution. In *Machine Learning, Proceedings of the Fifth International Conference on Machine Learning, Ann Arbor, Michigan, USA, June 12-14, 1988*, pages 339–352, 1988.

[91] Stephen Muggleton et al. Stochastic logic programs. *Advances in inductive logic programming*, 32:254–264, 1996.

[92] Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In *Algorithmic Learning Theory, First International Workshop, ALT '90, Tokyo, Japan, October 8-10, 1990, Proceedings*, pages 368–381, 1990.

[93] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *J. Log. Program.*, 19/20:629–679, 1994.

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

[95] Stephen H. Muggleton, Dianhuan Lin, Jianzhong Chen, and Alireza Tamaddoni-Nezhad. Metabayes: Bayesian meta-interpretative learning using higher-order stochastic refinement. In *Inductive Logic Programming - 23rd International Conference, ILP 2013, Rio de Janeiro, Brazil, August 28-30, 2013, Revised Selected Papers*, pages 1–17, 2013.

[96] Stephen H. Muggleton, Dianhuan Lin, Niels Pahlavi, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning: application to grammatical inference. *Machine Learning*, 94(1):25–49, 2014.

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

[98] Claire Nédellec, Céline Rouveirol, Hilde Adé, Francesco Bergadano, and Birgit Tausend. Declarative bias in ilp. *Advances in inductive logic programming*, 32:82–103, 1996.

[99] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. *Foundations of Inductive Logic Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[100] Ramón P. Otero. Induction of the indirect effects of actions by monotonic methods. In Stefan Kramer and Bernhard Pfahringer, editors, *Inductive Logic Programming, 15th International Conference, ILP 2005, Bonn, Germany, August 10-13, 2005, Proceedings*, volume 3625 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 2005.

[101] Andrea Passerini, Paolo Frasconi, and Luc De Raedt. Kernels on prolog proof trees: Statistical learning in the ILP setting. 2005.

[102] Charles Sanders Peirce. *Collected papers of charles sanders peirce*, volume 2. Harvard University Press, 1974.

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

[104] Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 107–126, 2015.

[105] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[106] J. Ross Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[107] Luc De Raedt. *Logical and relational learning*. Cognitive Technologies. Springer, 2008.

[108] Luc De Raedt. Declarative modeling for machine learning and data mining. In *Algorithmic Learning Theory - 23rd International Conference, ALT 2012, Lyon, France, October 29-31, 2012. Proceedings*, page 12, 2012.

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

[110] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2462–2467, 2007.

[111] Oliver Ray. Nonmonotonic abductive inductive learning. *J. Applied Logic*, 7(3):329–340, 2009.

[112] Scott E. Reed and Nando de Freitas. Neural programmer-interpreters. *CoRR*, abs/1511.06279, 2015.

[113] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[114] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, New Jersey, 2010. Third Edition.

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

[116] Chiaki Sakama. Nonmonotomic inductive logic programming. In Thomas Eiter, Wolfgang Faber, and Miros law Truszczyński, editors, *Logic Programming and Nonmotonic Reasoning: 6th International Conference, LPNMR 2001 Vienna, Austria, September 17–19, 2001 Proceedings*, pages 62–80. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[117] Ute Schmid, Christina Zeller, Tarek Besold, Alireza Tamaddoni-Nezhad, and Stephen Muggleton. How does predicate invention affect human comprehensibility? In *ILP*, volume 10326 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 2016.

[118] Peter Schüller and Mishal Kazmi. Best-effort inductive logic programming via fine-grained cost-based hypothesis generation. *arXiv preprint arXiv:1707.02729*, 2017.

[119] E.Y. Shapiro. *Algorithmic program debugging*. MIT Press, 1983.

[120] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 634–651, 2012.

[121] Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[122] David Skillicorn. *Understanding complex datasets: data mining with matrix decompositions*. CRC press, 2007.

[123] Ray J. Solomonoff. A formal theory of inductive inference. part I. *Information and Control*, 7(1):1–22, 1964.

[124] Ray J. Solomonoff. A formal theory of inductive inference. part II. *Information and Control*, 7(2):224–254, 1964.

[125] J Michael Spivey and JR Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, 1992.

[126] A. Srinivasan. A study of two probabilistic methods for searching large spaces with ilp. Technical Report PRG-TR-16-00, Oxford University Computing Laboratory, Oxford, 2000.

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

[128] Irene Stahl. The appropriateness of predicate invention as bias shift operation in ILP. *Machine Learning*, 20(1-2):95–117, 1995.

[129] Leon Sterling and Ehud Y Shapiro. *The art of Prolog: advanced programming techniques*. MIT press, 1994.

[130] Phillip D. Summers. A methodology for LISP program construction from examples. *J. ACM*, 24(1):161–175, 1977.

[131] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press, 1998.

[132] Sten-Åke Tärnlund. Horn clause computability. *BIT*, 17(2):215–226, 1977.

[133] Ken Thompson. Retrograde analysis of certain endgames. *ICCA journal*, 9(3):131–139, 1986.

[134] Alan M Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[135] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[136] Leslie G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, 1984.

[137] S. Vera. Induction of concepts in the predicate calculus. In *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, September 3-8, 1975*, pages 281–287, 1975.

[138] David H. D. Warren, Luis M. Pereira, and Fernando Pereira. Prolog - the language and its implementation compared with lisp. *SIGART Newsletter*, 64:109–115, 1977.

[139] Christopher J. C. H. Watkins and Peter Dayan. Technical note q-learning. *Machine Learning*, 8:279–292, 1992.

[140] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 364–374, 2009.

[141] Bo Wu and Craig A. Knoblock. An iterative approach to synthesize data transformation programs. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1726–1732, 2015.

[142] Zhao Xing, Yixin Chen, and Weixiong Zhang. Optimal STRIPS planning by maximum satisfiability and accumulative learning. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling, ICAPS 2006, Cumbria, UK, June 6-10, 2006*, pages 442–446, 2006.

[143] Wojciech Zaremba, Tomas Mikolov, Armand Joulin, and Rob Fergus. Learning simple algorithms from examples. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 421–429, 2016.