Nonmonotonic Inductive Logic Programming as Abductive Search

Domenico Corapi

2011

Supervised by Dr. Alessandra Russo and Dr. Emil Lupu

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
Declaration

I herewith certify that all material in this dissertation which is not my own work has been properly acknowledged.

Domenico Corapi
Abstract

Inductive Logic Programming (ILP) is a machine learning technique that relies on logic programs as a representation language. Most of the effort in the field of ILP has concentrated on a restricted class of problems, providing solutions that do not fully support negation. On the other hand, integration of negation and nonmonotonic reasoning in logic programming is common and important for a number of problems.

This thesis presents an approach to nonmonotonic ILP that is based on a transformation of the original problem to a problem that can be solved by employing abductive reasoning. In particular we present a general framework for the transformation that can be used as reference for concrete implementations.

We instantiate the transformation to derive two alternative implementations that are rooted in the two dominant computational logic paradigms: Prolog and Answer Set Programming (ASP). In the first case, we derive an implementation, called TAL, that is based on the abductive proof procedure SLDNFA and uses a customisable best-first search on the space of abductive solutions. In the second case the transformation is further refined in order to exploit the computational properties of available ASP solvers. In the proposed system called ASPAL, a theory is constructed from a set of mode declarations and used to extend the search of the underlying solver, so enabling the derivation of inductive hypotheses.

We provide completeness and soundness results for the framework and the ILP systems presented and show how, as a consequence of this, it is possible to induce complex multi-predicate hypotheses involving negation, recursion and the definition of elements of the domain that are not directly observed.

We validate the framework on established ILP benchmark problems, on some nonmonotonic ILP problems proposed in the literature. Furthermore we demonstrate the approach on a novel application of nonmonotonic ILP to the revision of normative frameworks.
To my father, who introduced me to computer science,
and to the rest of my family.
Contents

Abstract 3

1 Introduction 13
   1.1 Motivation and Objectives .................................................. 14
   1.2 Contributions ................................................................. 16
      1.2.1 A generic sound and complete framework for nonmonotonic ILP .......... 16
      1.2.2 A top-directed system for nonmonotonic ILP ........................... 18
      1.2.3 Integration of Answer Set Programming and ILP ....................... 18
      1.2.4 Theory revision as nonmonotonic ILP and application to normative framework revision 19
   1.3 Thesis overview .............................................................. 19
   1.4 Publications .................................................................... 21

2 Logic Programming and preliminaries 23
   2.1 General notation ................................................................. 23
   2.2 First order logic ................................................................. 24
      2.2.1 Syntax ..................................................................... 24
      2.2.2 Semantics .................................................................. 25
   2.3 Logic programming ............................................................. 26
   2.4 Nonmonotonic Logic Programming ......................................... 28
      2.4.1 SLDNF and Completion semantics ....................................... 30
      2.4.2 Stable model semantics ................................................. 32
   2.5 Abductive Logic Programming .............................................. 33
   2.6 Which semantics? ................................................................. 34
   2.7 Program transformations ..................................................... 35
   2.8 Conclusions .................................................................... 36
5.4 Inductive extensions and TALH ........................................ 81
  5.4.1 Partial rules ...................................................... 82
  5.4.2 Extensions of the abductive procedure .............................. 86
5.5 Learning settings ..................................................... 90
5.6 Cover loop approach in TAL ........................................... 96
5.7 Implementation and evaluation ......................................... 98
5.8 Discussion and related work ......................................... 99
  5.8.1 Other nonmonotonic ILP systems ................................ 100
5.9 Conclusions .......................................................... 101

6 Abductive Learning in Answer Set Programming ..................... 104
  6.1 An encoding for ASP .................................................. 104
  6.2 Clause space generation ............................................... 107
    6.2.1 Skeleton rules .................................................... 109
  6.3 Algorithm ............................................................ 111
    6.3.1 Derivation of skeleton rules .................................... 112
  6.4 Experiments ........................................................ 118
  6.5 Related work ........................................................ 120
  6.6 Conclusion .......................................................... 122

7 Theory Revision as Nonmonotonic ILP ............................... 123
  7.1 Introduction ......................................................... 123
  7.2 Theory Revision ..................................................... 124
    7.2.1 The operator based approach ................................... 125
  7.3 Theory revision through nonmonotonic ILP .......................... 127
    7.3.1 Soundness, completeness and minimality ....................... 131
  7.4 Related work ....................................................... 134
    7.4.1 RUTH ............................................................ 135
    7.4.2 FORTE .......................................................... 135
    7.4.3 INTHELEX ....................................................... 136
  7.5 Normative framework design ......................................... 138
    7.5.1 Normative frameworks and formal model ....................... 139
    7.5.2 Methodology .................................................... 141
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.5.3 Case Study</td>
<td>142</td>
</tr>
<tr>
<td>7.5.4 Discussion</td>
<td>145</td>
</tr>
<tr>
<td>7.6 Conclusions</td>
<td>147</td>
</tr>
<tr>
<td>8 Conclusions</td>
<td>148</td>
</tr>
<tr>
<td>8.1 Future Work</td>
<td>150</td>
</tr>
<tr>
<td>Bibliography</td>
<td>151</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Inference operators in A-SYSTEM. .................................................. 77
5.2 Accuracy over a 10-fold cross-validation (and standard deviation) on the datasets for TAL, ALEPH and TOPLLOG ........................................ 99
6.1 Computation time and other information on the experiments. ............. 120
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Overview of the ILP and TR frameworks discussed in the thesis.</td>
<td>20</td>
</tr>
<tr>
<td>5.1</td>
<td>Full derivation tree for the “even-odd” example.</td>
<td>94</td>
</tr>
<tr>
<td>5.2</td>
<td>Selected states of the derivation tree.</td>
<td>103</td>
</tr>
<tr>
<td>6.1</td>
<td>ASP theory generated for Example 6.6.</td>
<td>119</td>
</tr>
<tr>
<td>7.1</td>
<td>Iterative design driven by use cases.</td>
<td>141</td>
</tr>
</tbody>
</table>
Acknowledgements

First and foremost I would like to express my deep and sincere gratitude to my supervisors Alessandra Russo and Emil Lupu for supporting and encouraging me during these years and for their outstanding guidance. I warmly thank all the members of the Policy group at Imperial College for providing an ideal work environment.

This thesis would have not been possible without the help of Daniel Sykes. His contribution to the completion of this thesis and help has been invaluable. I would also like to express my sincere gratitude to Jiefei Ma and to Dalal Alrajeh for their collaboration that has been essential for some parts of this thesis. I would like to thank all the PhD students and Research Assistants at Imperial College, in particular Robert Craven, Rudi Ball, Enrico Scalavino, Leonardo Mostarda, Changyu Dong, Anandha Gopalan, Srdjan Marinovic, Themis Bourdenas, Dimosthenis Pediaditakis and Luke Dickens.

I’m also highly indebted to Bashar Nuseibeh and all the people involved in the PRiMMA project for the useful discussions and the valuable experience I gained while working with them. During these years I have collaborated with many academics who truly enriched my experience as a PhD student. In particular, I owe my most sincere gratitude to Marina De Vos and Julian Padget for the fruitful collaborations. I would also like to express my gratitude to Katsumi Inoue, the members of the Inoue lab, Ken Satoh, Taisuke Sato and all the people that during my stay in Japan contributed to one of the most memorable experiences of my life.

A special word of thanks to all my friends Antonio, Giacomo, Alicia, Lyndsey, Daniel, and to all the others whom I cannot mention here individually, for staying close to me in both happy and gloomy times.

Finally, I owe my deepest thanks to my family, for their unconditional help through all these years. Without them, none of my achievements would have been possible nor worthwhile.
The reasonable man adapts himself to the world; the unreasonable one persists in trying to adapt the world to himself. Therefore all progress depends on the unreasonable man.

—George Bernard Shaw
1 Introduction

The idea of machines being able to synthesise some form of intelligence autonomously, by sensing the external world they operate in, is without doubt fascinating and has been the focus of research in Artificial Intelligence (AI) and more specifically in the field of Machine Learning over the last sixty years [Turing, 1950]. At the same time, AI has been influenced by logic, in particular by seeking the benefits of using formal logic both as a representation language and as a general framework for inference and reasoning.

Inductive Logic Programming (ILP) [Nienhuys-Cheng & de Wolf, 1997] is a technique that can be placed at the intersection of the two, as a machine learning technique that relies on computational logic to represent the knowledge involved in the process. The goal is undoubtedly ambitious: ILP aims at developing a technique that is able to process observations to produce a set of logic sentences that generalise them, called hypotheses. This closely resembles and abstracts human reasoning where inputs obtained by sensing the world are used to derive knowledge and ultimately affect our seemingly logical reasoning.

Despite the results achieved by the ILP community in a number of applications, notably in the biological domain [Muggleton, 2005], the field is still young. Plotkin’s work in the early ’80s can be considered the earliest contribution, defining some of the theoretical foundations of the field. The discipline acquired its name and became a branch of machine learning on its own in the ’90s with the appearance of the first ILP workshops [Muggleton, 1991], accompanied by increasing interest that led to the development of a multitude of techniques and implementations [Flach & Džeroski, 2001].

ILP has some distinctive features that differentiate it from other machine learning techniques. The outcome of the learning is a logic theory, a set of logic sentences or rules, which are understandable for domain experts and in general easy to translate into a convenient representation for humans. The format of the inputs is also advantageous as ILP systems naturally support multi-relational representations that model complex relationships amongst entities of the domain. A background knowledge can be provided in order to support the process with predefined domain concepts.

The two current directions of research in ILP are on one side the study of appropriate logical formalisms that are able to capture the intended solutions and on the other the study of techniques that improve the computation
of such solutions. The focus of this thesis is mainly, but not exclusively, on the former as we introduce a framework, based on a semantics-preserving transformation, that relates ILP to an equivalent *abductive reasoning* problem. Abductive reasoning and its realisation in logic programming, known as *Abductive Logic Programming* (ALP) [Kakas et al., 1992], are closely related to ILP. They both support reasoning from observations to possible explanations: in the case of abductive reasoning explanations are specific instances, while in the case of inductive reasoning they capture general patterns and regularities.

The aforementioned transformation is shown to be valuable for a number of reasons. Notably, it provides the theoretical basis for results on completeness and soundness of two novel ILP systems which we introduce in this thesis. Moreover this new framework is powerful enough to support a harder class of problems, which operate under nonmonotonicity of the inference and of the search, for which the fundamental assumptions of existing ILP implementations do not hold. In fact, under nonmonotonic inference, consequences of a certain theory can be retracted when the theory is extended. This complicates the search process that cannot deal sequentially with the induction of logic rules as it is usually done under the assumption of monotonicity.

In the following sections we review the motivations and objectives and elaborate more in detail on the main contributions of the thesis.

### 1.1 Motivation and Objectives

The motivation for our work stems both from practical and theoretical considerations. From a theoretical point of view, one line of research in ILP has tackled the problem of characterising completeness of existing systems in various ways. The work of Yamamoto [Yamamoto, 1997] studies a form of incompleteness in *inverse entailment* (IE), a technique used in what is considered the most successful ILP system *PROGOL* [Muggleton, 1995]. [Moyle, 2002] proposes a method that extends IE in order to support the learning of multiple clauses from a given seed example using *abductive reasoning*. [Ray, 2005] further extends the original *PROGOL* procedure providing a more comprehensive solution that uses abductive reasoning by developing a procedure called *HAIL*. [Yamamoto et al., 2010] solves the problem of enhancing *inverse subsumption*, a widely used generalisation technique, defining conditions for completeness. [Kimber et al., 2009] extends *HAIL*, in order to support so called *connected theories*. Although these various solutions proposed address and solve the original incompleteness of inverse entailment on known problematic instances, a general formal result on completeness for systems based on logic programming is still missing. This has been a central objective of the work presented in this thesis.
Nonmonotonic ILP is an emerging research area that aims at extending the applicability of ILP to learning under nonmonotonic inference which supports the use of negation and defeasible reasoning. Despite the existing body of work (e.g. [Sakama, 2005, Sakama, 2001b, Ray, 2009a, Otero, 2001, Ray et al., 2009]), for which we provide an overview here, the low maturity of this area demands both a deeper theoretical understanding and the design and development of algorithms and tools. The early work in nonmonotonic ILP concentrated on restricted classes of problems that can be supported by refinements of standard ILP techniques. For example, [Fogel & Zaverucha, 1998] define the conditions that permit the application of established techniques based on the sequential covering of the examples provided. [Dimopoulos & Kakas, 1995] separates the learning of default rules from the learning of exceptions to these rules. [Otero, 2001] draws attention to some peculiarities of nonmonotonic ILP, such as the possibility of using a different semantics – the stable model semantics – and consequently the necessity of handling multiple models. The work of Sakama [Sakama, 2005], further investigates the use of such semantics, developing the intuition, further investigated in this thesis, of using Answer Set Programming to support nonmonotonic inference within the learning process. One of the limitations, addressed in this thesis, is that the procedure is a best effort iterative process that considers the learning of a single rule from a single example.

More recently a nonmonotonic ILP system has been proposed, called XHAIL [Ray, 2009b], that addresses some of the shortcomings of earlier approaches to nonmonotonic ILP. In fact it is the first integrated approach that generalises a set of logic rules with respect to the whole set of examples provided, rather than iteratively constructing the rules from single examples. XHAIL has several limitations, in particular the lack of a formal result on completeness, addressed and discussed in detail in this thesis.

The solutions provided in this thesis approach the problem of nonmonotonic ILP in a radically novel way, replacing the IE framework used in XHAIL with the use of an “artificial theory”, constructed from a specification of the language bias, that constrains the search for inductive solutions. Lastly, none of the nonmonotonic ILP systems proposed in the literature is supported by studies on their applicability to real world problems. This is due, in part, to the fact that real world applications that benefit from nonmonotonic ILP have been proposed only recently ([Alrajeh et al., 2007, Alrajeh et al., 2009, Alrajeh et al., 2011, Ray & Bryant, 2008, Ray et al., 2009, Corapi et al., 2009, Corapi et al., 2011a]). Moreover, many solutions proposed in the literature are deliberately more of theoretical rather than practical interest.

From an application point of view, the problem of learning under nonmonotonic inference applies to an increasingly large set of domains, as shown by the application scenarios emerging in parallel to this research and in the years immediately before (such as [Alrajeh et al., 2011, Ray & Bryant, 2008]). However, the ILP community has been focusing mostly on monotonic inference and on restricted, more tractable representations.
Besides, most systems favour performance in the search, sacrificing thoroughness, whilst most often in non-
monotonic problems the interest is on finding all the possible solutions (e.g. when a supervisor must choose one
of possible correct candidate hypotheses) and thus completeness plays a central role. In fact, one of our goals
is to provide a theoretically sound and complete framework where the space of possible solutions is restricted
by user bias or search heuristics and not by inherent incompleteness. Therefore one of the main objectives of
this thesis is to devise new techniques that support richer representations and at the same time guarantee that
all the solutions of a given problem can be derived by such techniques. Due to the low maturity of the field
and the lack of an established benchmark we cannot provide a conclusive empirical evaluation. Despite this,
in this thesis we also investigate possible enhancements that broaden the concrete applicability of the proposed
framework, including the use of search heuristics and the support for noisy examples.

Starting from the aforementioned objectives, in this thesis we set forth and explore the hypothesis that non-
monotonic ILP can be formulated as an abductive reasoning problem and benefit from this formulation. Besides
this, our approach provides a number of valuable features examined more in details in Section 1.2.

1.2 Contributions

Placed in the research scenario illustrated above, we briefly discuss the main contributions of this thesis.

1.2.1 A generic sound and complete framework for nonmonotonic ILP

Abductive Logic Programming (ALP) is a logic programming technique that supports abductive reasoning, i.e.
the reasoning from observations to possible explanations. In this thesis we argue that also (nonmonotonic)
ILP, under certain plausible assumptions is a form of abductive reasoning. In particular whenever the space of
possible inductive hypotheses is constrained using a set of mode declarations [Muggleton, 1995], it is possible
to perform abductive inference on a “meta-domain” of the structure of the possible inductive hypotheses. To
achieve this we introduce a general mapping from (nonmonotonic) ILP tasks to ALP tasks. This enables the
use of established ALP techniques and tools to derive inductive hypotheses. We observe that a very general
class of inductive problems can be handled, including problems like multi-predicate learning that require ad
hoc extensions in other ILP systems.

**Example 1.1** We present through this example the intuition behind the ILP techniques introduced in this thesis.
Consider the following theory:

\[ B = \begin{cases} 
    a(X) \leftarrow \text{not } b(X) \\
    c(1). 
  \end{cases} \quad (1.1) \]

We want to find an additional theory \( H \) such that \( B \cup H \models \text{not } a(1) \). Without delving into the details of the semantics adopted, intuitively \( a(1) \) is a consequence of \( B \) since \( b(1) \) is false. In a nonmonotonic framework a new theory \( H \) can be found such that \( a(1) \) is no longer a consequence. Let us consider the mode declaration set \( M = \{ \text{mode}(b(\text{any})), \text{mode}(\text{not } c(\text{any})) \} \). We formally introduce mode declarations in Chapter 3. For the sake of understanding in this simple example the mode declarations instantiate a space of possible solutions that includes the two rules \( b(X) \) and \( b(X) \leftarrow c(X) \). Now the main idea is that we can use these mode declarations to create an additional theory that captures the meta-level representation of inductive hypothesis accepted by the mode declaration, and that can be used to perform abductive reasoning to solve the original inductive problem. In this example we can use the following theory \( \top \):

\[ \top = \begin{cases} 
    b(X) \leftarrow \text{rule}(r_1). \\
    b(X) \leftarrow \text{not } c(X), \text{rule}(r_2). 
  \end{cases} \quad (1.2) \]

Despite the simplicity of the example we can observe that by making the rules in the search space explicit we can translate the problem into an equivalent abductive reasoning problem. Note that in (1.2) we do this by enumerating all the rules in \( \top \) extensionally but the mapping proposed in this thesis constructs generic theory that is an intensional representation of the rules. Given the theory \( \top \) we can apply abductive reasoning to derive ground facts, like \( \text{rule}(r_1) \), that can be reinterpreted as a logic rule. In this particular example both the facts \( \text{rule}(r_1) \) and \( \text{rule}(r_2) \) are abductive solutions (i.e. for \( \Delta = \{ \text{rule}(r_1) \} \) and \( \Delta = \{ \text{rule}(r_2) \} \), \( B \cup \top \cup \Delta \models \text{not } a(1) \)). In a post-processing step we can easily derive that both the candidate rules are solutions for the problem \((H = \{ b(X) \} \) and \( H = \{ b(X) \leftarrow \text{not } c(X) \} \)). In fact, if \( \text{rule}(r_1) \) is true, then we can simplify the first of the two rules in \( \top \) and assert \( b(X) \). Similarly, if \( \text{rule}(r_2) \) is true we can simplify the second rule and assert \( b(X) \leftarrow \text{not } c(X) \). Assuming a more constraining requirement \( B \cup H \models \text{not } a(1), a(2) \), it is easy to verify that there is only one abductive solution \( \text{rule}(r_2) \), and consequently the solution for this new problem is \( H = \{ b(X) \leftarrow \text{not } c(X) \} \).

Although this is a very simple example, it gives an intuition on how this idea enables ILP in a nonmonotonic context and, contrarily to the majority of the existing ILP solutions, it is able to generate solutions both from positive and negative examples symmetrically.
The completeness problem is a much-debated topic in ILP [Yamamoto, 1997], and it has been addressed without a definitive solution in a number of works. The only ILP system that provides a completeness result is [Inoue, 2004] but it deals with classical (monotonic) first-order inference, leaving open the problem of a complete ILP system that supports nonmonotonic inference. The framework proposed provides results on soundness and completeness. In particular the transformation preserves the original semantics, thus relating the soundness and completeness of the inductive problem to the properties of the underlying abductive system used.

Besides the ILP systems derived from the use of the transformation, the mapping has a more general significance as it can be used to relate the body of work on ALP to the derivation of hypotheses structured as logic rules. As an example, in [Corapi et al., 2011b], we applied a simplified transformation to the learning of probabilistic rules, extending the work of [Inoue et al., 2009] and [Gutmann et al., 2008] from the derivation of probabilistic abductive explanations under distribution semantics [Sato, 1995] to the construction of structured probabilistic rules under the same semantics.

1.2.2 A top-directed system for nonmonotonic ILP

We have designed and implemented a nonmonotonic ILP system, called TAL (Top-directed abductive learning), that stems from the theoretical notions introduced in this thesis. TAL uses the mapping of the ILP problem to an ALP task and employs a goal driven procedure that supports customisable search strategies to derive hypotheses. The system can be applied to challenging problems that involve predicate invention, multi-predicate, recursive and non-observational [Moyle, 2002] learning.

We also show that under certain restrictions, the computation results in a process that resembles other ILP systems. The sequential covering approach used in many other ILP systems (e.g. PROGOL) can be used also in TAL, producing results on well known ILP problems that are comparable to more established ILP systems in terms of predictive accuracy.

1.2.3 Integration of Answer Set Programming and ILP

The generality of the framework introduced in this thesis is demonstrated also by its use within a different computational logic paradigm. Answer Set Programming is an emerging logic programming paradigm that supports nonmonotonic reasoning. The work of [Sakama, 2001a] and [Ray, 2009b] investigate the use of ASP within ILP. We propose an alternative method for the induction of hypotheses under ASP, using the abductive framework given in this thesis.

We describe ASPAL (ASP Abductive Learning), an ILP system that adapts our abductive framework in order
to exploit the features of ASP solvers. In ASPAL the search of inductive solutions is indeed delegated to an
ASP solver. We also discuss how the shift to harder problems (e.g. where the space of possible solutions is par-
ticularly large) affects the computation and address the problem of finding effective computation strategies. As
the overall computation time is affected mostly by the grounding [Gebser et al., 2009], we employ an iterative
technique that increasingly extends the maximum number of literals in the rules of the hypotheses. We finally
provide an empirical evaluation of the ASPAL system on a set of nonmonotonic ILP problems.

1.2.4 Theory revision as nonmonotonic ILP and application to normative framework revision

The problem of Theory Revision (TR) [Wrobel, 1996], i.e. finding a minimal revision of a given initial theory
that correctly entails a set of examples, has been addressed in the literature by approaches based on a search
over a space of possible repeated applications of revision operators like deleting a clause or adding a condition
to an existing rule.

The development of a generic nonmonotonic ILP framework has also made possible the realisation of an
approach for TR based on inductive learning. Specifically, we show that TR can be approached as a particular
task of nonmonotonic ILP, providing an operational description of the link between the two. In other words it
is possible to rewrite a TR problem into a nonmonotonic ILP problem avoiding the use of an ad-hoc solution
for the revision.

We also introduce a novel applications of ILP in which we revise a normative framework, i.e. regulations
that control the behaviour of set of agents, formalised as a normal logic program. This application is inherently
nonmonotonic and substantiates many distinctive features of the technique proposed here.

1.3 Thesis overview

Figure 1.1 outlines some of the main conceptual steps described in this thesis. On the upper side of the figure it
is shown how ILP can be performed by applying abductive reasoning on a pre-processed theory, obtained from
a set of mode declarations. The result of the abductive reasoning task is then processed into the final hypothesis
for the ILP task. The bottom side of the figure shows the steps involved in the approach to TR discussed in this
thesis. Also in this case a set of mode declarations, this time together with a revisable theory are pre-processed
into a theory given as input for an ILP task. The hypothesis generated by the ILP task is then processed into
a revised theory. Note that although the overall process resembles the one previously described, in this case
different types of processing and postprocessing are performed. Also, the actual search is performed by an
abductive reasoning system in the first case, and by an ILP system in the second, where mode declarations are
used to define the space of possible revisions in addition to the usual hypothesis space for new rules.

The two pre-processing phases transform the given logic theory into a semantically equivalent one that can be used to solve the original task. The post-processing phases also preserve the semantics and are used to transform a partial output into the output required for the given task.

![Diagram](image)

**Figure 1.1:** Overview of the ILP and TR frameworks discussed in the thesis.

In *Chapter 2* we summarise preliminary notions and notations used in this thesis. We give an overview of logic and logic programming and discuss nonmonotonic semantics. We also define the task of ALP and compare alternative semantics, discussing classes of programs for which they coincide in order to abstract from particular cases and provide a more unifying vision of the context of this thesis.

In *Chapter 3* we present the relevant ILP background. In particular we define the setting and discuss some of the main techniques, focusing on top-down approaches. We survey the most important ILP systems, presenting more in detail four systems that are used for comparison.

In *Chapter 4* we detail an original framework that is based on the formalisation of an ILP task into an ALP task, based on the declaration of the language bias through mode declarations. We discuss the properties of the mapping and the semantic equivalence between the original ILP task and the abductive problem generated by our approach.

In *Chapter 5* we use the transformation presented in *Chapter 4* to design a nonmonotonic ILP system called TAL. We first discuss a basic implementation that applies the SLDF proof procedure as a procedural counterpart of the transformation. We prove the soundness and completeness of the resulting system. We then present control mechanisms that allow the use of search heuristics and the reduction of the search space by excluding redundant parts of the search space. We introduce a mode of execution that operates using the standard sequential covering approach and discuss the similarities to more established ILP systems. We conclude with some notes on the implementation, an empirical evaluation on four well-known ILP problems and discussion
on related work.

In Chapter 6 we show how the framework presented in Chapter 4 can be further refined in order to adapt to the different mode of computation and semantics of ASP. We discuss the implementation of a nonmonotonic ILP system called ASPAL. We finally evaluate the system over a set of nonmonotonic ILP problems and show in particular its application to the domain of normative systems.

In Chapter 7 we introduce a general methodology that supports TR through nonmonotonic ILP. We give an account of the main computational steps and compare it to other TR systems.

In Chapter 8 we review the contributions and conclude the thesis discussing possible future directions of research.

The comparison with other ILP solutions and in general the discussion on related work is distributed over Chapters 3, 4, 5 and 6. In Chapter 3, we review some of the reference ILP systems, discussing the procedures employed and highlighting the limitations of such systems. In Chapter 4, we examine approaches proposed in literature that integrate abduction within ILP. In Chapter 5, we investigate nonmonotonic ILP systems in relation to our TAL system. In Chapter 6, we compare and review existing approaches to ILP that are based on ASP.

1.4 Publications

Some parts of the work presented in this thesis have appeared in following publications:

  Description of the integration of ILP an ASP and of the ASPAL system (Chapter 6).

  The transformation technique presented in Chapter 4 is used to perform learning of rules with associated probability, under distribution semantics. This is not discussed in this thesis, although it provides an application of the transformation.

2011.
Application to revision of normative framework (Chapter 7) and preliminary version of ASPAL (Chapter 6).

  Application of TAL to the revision of Business Process models. Not discussed in this thesis.

  Application to the learning of rules for normative framework design.

  Application scenario where TAL is integrated in a collaborative privacy policy authoring system. Not discussed in this thesis.

  Preliminary version of the transformation in Chapter 4 and design of the TAL system (Chapter 5).

  General technique for TR based on nonmonotonic ILP (Chapter 7) and preliminary application to the learning of rules modelling mobile phone user behaviour.
2 Logic Programming and preliminaries

In this chapter we introduce some preliminary notions used in this thesis. The exposition will concentrate on the essential definitions and results. We refer to [Lloyd, 1984], [Nienhuys-Cheng & de Wolf, 1997] and [Hogger, 1990] for further details.

First-order logic is a well studied formal language that, amongst other uses, supports automatic reasoning and provides a general framework for knowledge representation. It extends propositional logic by introducing entity of a domain and expressing relationships amongst these entities. For example “3 is greater than 2” and “6 is greater than 3” can only be expressed as two different propositions $p$ and $q$ in propositional logic. The information that both propositions refer to instances of a general relation “greater” and to the same domain entities cannot be captured by propositional logic. First-order logic can be used to introduce the relation greater and instantiate it over the entities $1, 2, 3, \ldots$ making use of variables, thus realising a more powerful and expressive language.

We are interested in the computational formalism called logic programming that uses a computationally tractable subset of first-order logic and combines it with computational mechanisms used for inference.

2.1 General notation

Throughout this thesis we assume the usual set notation [Halmos, 1960]. We recall the meaning of some of the symbols used in this thesis here. A set is represented as $S = \{e_1, \ldots, e_n\}$ (or intensionally as $\{e : p(e)\}$, where $p(e)$ is a proposition on $e$ over the universe of discourse) and its cardinality as $|S|$. An $n$-tuple is denoted as $\langle e_1, \ldots, e_n \rangle$. For convenience we will also use the name list to denote tuples and adopt the alternative notation $[e_1, \ldots, e_n]$ or simply $e_1, \ldots, e_n$. Also, with abuse of notation we will say that an element $e$ is in a list $l$, denoted $e \in l$ whenever $e$ is in the image of $l$. We use the usual notation $f : A \to B$ to denote a function with domain $A$ and codomain $B$, writing $b = f(a)$ to indicate that $f$ maps $a$ to $b$. 
2.2 First order logic

2.2.1 Syntax

First-order logic is a formal language. Syntactically, it defines a class of languages that are characterised by an alphabet $\Sigma$ consisting of the following symbols:

- the logical symbols "\(\land\)" (conjunction), "\(\lor\)" (disjunction), "\(\neg\)" (negation), "\(\leftarrow\)" (implication), "\(\leftrightarrow\)" (equivalence), "\(\exists\)" (existential quantifier) and "\(\forall\)" (universal quantifier);

- the punctuation symbols "(" and ")";

- a set $\Sigma^p$ of predicate symbols with specified arities;

- a set $\Sigma^f$ of functor symbols with specified arities;

- a set $\Sigma^v$ of variable symbols.

A predicate or function $p$ with arity $n$ is denoted $p/n$. Functor symbols with arity 0 are also called constants. We follow the “Prolog convention” [Bratko, 1990] for the notation$^1$. In the following, variables are denoted by strings of letters and digits that start with an uppercase letter (e.g. $X$, $Body$, $Number$, $A$). Predicate, functor and constant symbols are denoted by strings of letters and digits that start with a lowercase letter or a number (e.g. $t$, 12, $body$, $user4213$). Whenever we refer to generic terms we use lower case letters of the type $t$. Tuples of terms $t_1, ..., t_n$ are abbreviated in the following as $\{t\}$. Whenever the terms are all variables, in order to simplify the reading, we use $X_1, ..., X_n$, abbreviated as $\{X\}$.

Symbols defined in the alphabet are used to construct syntactical structures. A term is either a constant, a variable or a function symbol $f/n$ applied to a tuple of $n$ terms $t_1, ..., t_n$, denoted $f(t_1, ..., t_n)$ and called an $n$-ary term. An atomic formula or atom is a predicate symbol $p$ applied to a tuple of terms $t_1, ..., t_n$, denoted $p(t_1, ..., t_n)$.

A well-formed formula (or formula) is either an atomic formula or, given that $\alpha_1$ and $\alpha_2$ are formulas, one of the following: $\alpha_1; (\neg \alpha_1); (\forall X. \alpha_1); (\exists X. \alpha_1); (\alpha_1 \land \alpha_2); (\alpha_1 \lor \alpha_2); (\alpha_1 \leftarrow \alpha_2); (\alpha_1 \leftrightarrow \alpha_2)$. The previous list also defines the priority and disambiguates the reading of a formula where the brackets are omitted.

For a given alphabet $\Sigma$, a first-order logic language $L_\Sigma$ is the set of all well-formed formulas that can be constructed from the symbols in $\Sigma$.

Given a formula $\forall X. \alpha$ (respectively $\exists X. \alpha$), $\alpha$ is the scope of the quantifier $\forall X$ (respectively $\exists X$). A variable $X$ is bound in a formula if it occurs in the scope of a quantifier that is immediately followed by $X$. A formula

$^1$Although this is not a conventional choice for first-order languages, we prefer to uniform the notation with logic programs, which are the main focus of this thesis.
where all the variables are bound is called a *closed formula*. A formula $f$ is *closed* if none of the variables in $f$ is free. Given a formula $f$, let $X_1, ..., X_n$ be all the distinct variables that occur in $f$ (denoted in the following as $\text{vars}(f)$), we denote by $\overline{\forall}.f$ the closed formula $\forall X_1...\forall X_n.f$. Closed formulae of the type $\forall X.f$ and $\exists X.f$ are called respectively *universally* and *existentially quantified* formulae. A *literal* is an atomic formula $\alpha$ or a negated atomic formula $\neg \alpha$. A *ground* term (resp. formula) is a term (resp. formula) containing no variables.

A *first order logic theory* is a set of formulae.

A *substitution* $\theta$ is a finite set of the form $\{X_1/t_1, ..., X_n/t_n\}$ where each $X_i$ is a different variable and each $t_i$ is a term distinct from $X_i$. We call each element of the set a *binding*. $\theta$ is called a ground substitution if all $t_i$ are ground terms. Given a formula $f$ and a substitution $\theta = \{X_1/t_1, ..., X_n/t_n\}$, the instantiated formula $f\theta$ is obtained by replacing each variables $X_i$ with the term $t_i$. A variable renaming is an invertible substitution, i.e. a substitution $\theta$ such that for some substitution $\theta^{-1}$, $\theta \circ \theta^{-1} = \theta^{-1} \circ \theta = \emptyset$ (the composition of the two substitutions produces the empty substitution). Formulae obtained by applying a variable renaming are equivalent (*up to variable renaming*). For example the clauses $p(X) \leftarrow q(X, Y), r(Y)$ and $p(Y) \leftarrow q(Y, X), r(X)$ are considered equivalent.

### 2.2.2 Semantics

The semantics of first-order logic can be defined through *model theory* for which we provide here a simplified introduction. We refer to [Lloyd, 1984] for a deeper explanation.

An interpretation relates elements of the logic language to a certain domain. More formally, an *interpretation* $I$ of a first order language $L_\Sigma$ consists of a non-empty set $D$, called the domain of the interpretation and the following assignments:

- an assignment to each constant $c$ in $\Sigma$ to an element $c^I$ in $D$;
- an assignment to each n-ary function symbol $g$ in $\Sigma$ of a function $g^I$ from $D^n$ to $D$;
- an assignment to each n-ary predicate $p$ in $\Sigma$ of a function $p^I$ from $D^n$ to $\{\text{true, false}\}$;

Given an arbitrary clause we want to establish the *truth value* of the clause, true or false, with respect to the given interpretation $I$. The atom $p(t_1, ..., t_n)$ is *true* (*false*) iff the value of $p^I(t_1^I, ..., t_n^I)$ is *true* (*false*). We omit additional notions on how to derive the truth value of logic formulae given a interpretation (e.g. the formula $a \land b$ is true in an interpretation $I$ iff both $a$ and $b$ are true in $I$). We assume the reader is familiar with logic connectives and quantifiers and refer to [Lloyd, 1984] for a thorough exposition.

An interpretation $I$ is a *model* for a closed formula $f$ if $f$ is true with respect to $I$. Similarly an interpretation $I$ is a model for a set of closed formulae if each formula is true with respect to $I$ (equivalently, $I$ is a model
for the conjunction of the formulae). A formula (theory) is consistent (satisfiable) iff it has at least one model, inconsistent (unsatisfiable) if it has no models, valid if every interpretation is a model.

Let $S$ be a set of closed formulae and $f$ be a closed formula. We say $f$ is a logical consequence of $S$, denoted $S \models f$, if for every interpretation $I$, $I$ is a model for $S$ implies that $I$ is a model for $f$. In this case we also say that $S$ (logically) entails $f$. In other words $S \models f$ if and only if the set of the models for $S$ is a subset of the models for $f$. For any set of closed formulae $S$, we denote the set of all the consequences as $Cn(S)$. In addition, let $S$ be a set of closed formulae and $f$ a closed formula, then $S \models f$ iff $S \cup \neg f$ is unsatisfiable [Lloyd, 1984]. This is used as a fundamental result for the inference methods used in logic programming, described later.

### 2.3 Logic programming

The characterisation given so far of first-order logic can be restricted to a computationally convenient language, that of logic programming, the discipline that, in the widest sense, deals with the computational techniques that support inference, i.e. the derivation of logical conclusions from premises.

A clause is a formula of the form $\forall (l_1 \lor \ldots \lor l_n)$ where each $l_i$ is a literal. A definite clause, also called a (definite) rule is a clause of the type $\forall (h \lor \neg b_1 \lor \ldots \lor \neg b_n)$ for which a different notation is used in the context of logic programming:

$$h \leftarrow b_1, \ldots, b_n$$

(2.1)

where $h$ is called the head of the clause and the rest is called the body. Each of the literals in the body is called condition. Given a rule $r$, head$(r)$ and body$(r)$ denote respectively the head of $r$ and the set of all the conditions in $r$.

A definite clause with empty body is called a fact, denoted as $h$. A goal or integrity constraint is a clause that contains one or more conditions and no head, denoted as $\leftarrow g_1, \ldots, g_n$. A query is given by the negation of a goal $g_1, \ldots, g_n$. A Horn clause is either a definite clause or a goal.

The conjunction of (definite) clauses $c_1 \land \ldots \land c_n$ is denoted by the set of these clauses $\{c_1, \ldots, c_n\}$ and is called (definite) program or (definite) theory. In this thesis we also use the following notation to represent a program $P = \{c_1, \ldots, c_n\}$:

$$P = \begin{cases} c_1 \\ \vdots \\ c_n \end{cases}$$

(2.2)
In order to avoid ambiguity with the conjunction in the body, clauses are sometimes separated by “.". For a given predicate $p$ we call the set of all the clauses with the predicate $p$ in the head in a theory $P$ the definition of $p$ in $P$.

We restrict our attention to a particular domain called Herbrand domain (Herbrand universe), the set of all ground terms that can be constructed using elements that occur in a given theory. A Herbrand domain (or universe) of a logic language $\mathcal{L}$, denoted as $\mathcal{U}_\mathcal{L}$, is the set of all the ground terms which can be formed using the constant symbols and function symbols in $\mathcal{L}$. A Herbrand base $\mathcal{B}_\mathcal{L}$ is the set of all the ground atoms constructed using predicate symbols in $\mathcal{L}$ and elements in $\mathcal{U}_\mathcal{L}$ as arguments of the predicates. An interpretation based on a Herbrand base is called a Herbrand interpretation.

Also, we will refer for convenience to interpretations and models of a theory $P$, referring to the underlying language from which the clauses in $P$ are formed. Interpretations and models will refer to Herbrand interpretations and Herbrand models. This means we can restrict the previous definitions of model, consistency, entailment etc. to Herbrand interpretations rather than general interpretations.

Under these assumptions, we can represent interpretations of a theory $P$ as subsets $I$ of $\mathcal{B}_P$. Elements of the Herbrand base that are true are included in $I$. All the remaining atoms are considered false for the interpretation $I$.

If we consider only definite clauses, we can simplify the definition of model of a theory as follows: an interpretation $I$ is a model for a theory $P$ if and only if for all clauses $h \leftarrow b_1, \ldots, b_n \in P$, for all ground substitutions $\theta$, $\{b_1\theta, \ldots, b_n\} \theta \subseteq I$ implies $h \in I$. It is also a well known result that there is a unique least (Herbrand) model of $P$ that is the intersection of all models of $P$ and is also the least fixed point of the immediate consequence operator $T_P$ defined as follows: $T_P(I) = \{ h : h \leftarrow b_1, \ldots, b_n \in P \text{ and } b_1, \ldots, b_n \in I \}$ [Lloyd, 1984]. Note that these characterisations of the semantics of definite programs coincide with the general semantics of first-order logic theories.

We briefly introduce a common approach to the procedural semantics of definite logic programs. Given a formula $f$ and a definite program $P$ it is of interest to know how to determine whether $P \models f$. Traditionally, in Prolog and more in general in logic programming, this is verified applying a deductive inference rule called resolution [Robinson, 1965]. This could be achieved by using the model theoretical semantics introduced, but this would involve the computation of all the (possibly infinite) models.

Resolution makes use of unifiers. A unifier for two terms or literals $p_1$ and $p_2$, used within a resolution step, is a substitution $\theta$ such that $p_1\theta = p_2\theta$. If such $\theta$ exists we say that the two terms or literals are unifiable. A most general unifier (m.g.u.) is substitution $\theta$ such that there is no other unifier $\theta'$ for which the term $p_1\theta'$ is

---

2 Given a theory $P$, $P$ has a model iff $P$ has a Herbrand model [Nienhuys-Cheng & de Wolf, 1997]
more general than $p_1 \theta$. We denote the most general unifier of two literals $p$ and $p'$ as $\text{m.g.u.}(p, p')$.

Given two definite clauses $h \leftarrow b_1, \ldots, b_n$ and $g \leftarrow c_1, \ldots, c_m$ the resolution operator infers the resolvent $h \leftarrow b_1, \ldots, b_{i-1}, b_{i+1}, \ldots, b_n, c_1, \ldots, c_m \theta$ where $\theta = \text{m.g.u.}(b_i, g)$. $b_i$ is called the selected literal. Note that the variables in the clauses are renamed so that none of them appear in both clauses. When the operator is applied more than once, we talk about resolution derivation or proof. When a clause $c$ is derived by a resolution proof using clauses from a theory $T$ we write $T \vdash_{\text{res}} c$. We also denote the set of resolvents of resolution on clauses $c_1$ and $c_2$ on some unspecified literal as $C = \text{res}(c_1, c_2)$. Resolution is sound, i.e. if $T \vdash_{\text{res}} c$ then $T \models c$, but not complete, i.e. it is not true that whenever $T \models c$ then $T \vdash_{\text{res}} c$. In order to establish if $T \models c$, resolution can be used to prove that $T \cup \{\neg c\}$ is unsatisfiable, i.e. to prove $c$ by refutation. Resolution is refutation complete [Lloyd, 1984].

Linear Resolution with a Selection function for Definite clauses (SLD) is at the base of all Prolog implementations [Bratko, 1990]. SLD is an instantiation of resolution where one of the two clauses that are resolved is always a denial (either the initial goal or the result of a previous resolution step) and the other is a clause in the theory. The derivation can thus be described as a sequence of goals $[g_0, g_1, \ldots, g_k]$ in which $g_0$ is the initial goal and each goal $g_{i+1}$ is derived by an SLD-resolution step from $g_i$ using clauses in $P$. An SLD-refutation is a SLD-derivation which terminates in a finite number of steps in the empty clause, i.e. a successful derivation. An SLD-refutation collects the m.g.u. for each step as the list $[\theta_1, \ldots, \theta_k]$. Let $\theta^*$ denote the composition of all the m.g.u. in the list. The restriction of $\theta^*$ to the variables that appear in the query is called the answer substitution. SLD is sound and complete with respect to logical entailment for definite programs [Lloyd, 1984], thus can be used as operational counterpart of the natural semantics of definite programs.

Prolog (PROgramming in LOGic), was developed as computational counterpart of logic programs. Prolog originally addressed definite programs and was later extended under the principle that it could be used as a full fledged programming language. This has led to extensions that favoured efficiency and control over the derivation to declarative style [Bratko, 1990].

2.4 Nonmonotonic Logic Programming

Logical entailment is monotonic, defined as the property under which, given the sets of formulae $S \subseteq T$, $\text{Cn}(S) \subseteq \text{Cn}(T)$. In other words whenever we add a new sentence to an existing program, the set of consequences of the new program can only grow. Definite programs encode positive information. Knowledge can be represented as logic facts or as rules. In this context, it is never possible to derive negative information.

Negative information can be explicitly stated in general first-order non-clausal representations, but this is
often not a good solution since the computational properties of clauses are lost and in most cases the amount of negative information can be unmanageable. Consider for example a knowledge base representing all the flights for a given company. Together with the list of flights, this approach would require the explicit representation of all the flights that do not take place, a high majority of the cartesian product of the involved airports. In cases like these it is much better to adopt an implicit representation of the negative information, i.e. all the flights that do not appear in the knowledge base do not take place. More in general, what is not a consequence of a theory is considered false. This assumption is known as Closed World Assumption (CWA) [Minker, 1982].

**Example 2.1** Consider the following theory:

\[
T = \begin{cases} 
\text{flight}(\text{london}, \text{paris}). \\
\text{flight}(\text{london}, \text{rome}). \\
\text{flight}(\text{new\_york}, \text{london}). 
\end{cases}
\] (2.3)

Whenever this knowledge base is associated to a classical first-order logic semantics the only conclusion that we can derive are the three stated facts. Under CWA we can derive, for example, \text{not flight}(\text{new\_york}, \text{rome}), that is intuitively what we expect from the given theory.

CWA causes nonmonotonicity. In fact in the previous example, if we add the fact \text{flight}(\text{new\_york}, \text{rome}) to the theory, \text{not flight}(\text{new\_york}, \text{rome}) is no longer a consequence. **Negation as failure** (NAF) interprets the negation operator as failure to prove, thus it relates to the operational semantics that supports the inference. This type of negation, implemented in most realisations of logic programming, is not a logical concept but a metaconcept: it defines what is true on the basis of the provability of some logic atoms. Note that we use a different notation “\text{not}” for this type of negation that distinguishes it from the classical \(\neg\). A normal clause is a clause of the type:

\[ h \leftarrow b_1, \ldots, b_m, \text{not } c_1, \ldots, \text{not } c_n \]

where each \(b_i, c_j, i = 1, \ldots, m, j = 1, \ldots, n\) is an atom. We call \(b_1, \ldots, b_m\) positive literals and \(\text{not } c_1, \ldots, c_n\) negative literals. We will refer to the conjunction of normal clauses, as normal theories or normal (logic) programs. Also, if not otherwise stated, we always refer implicitly to normal theory and normal clause whenever we write “theory” (or “program”) and “clause” (or “rule”).

In this thesis we use two of the most widespread semantics, the completion semantics [Clark, 1977] and stable model semantics [Gelfond & Lifschitz, 1988] for normal programs, presented in the next sections.
2.4.1 SLDNF and Completion semantics

SLDNF is an extension of SLD that implements NAF. Whenever a literal of the type not \( l \) is selected the procedure verifies that the goal \( \leftarrow l \) finitely fails, i.e. it does not contain successful derivations. We sketch here the procedure. An SLDNF-tree is constructed with the initial goal \( g \) as root. Let \( \leftarrow l_1, \ldots, l_i, \ldots, l_n \) be a node in the tree where \( l_i \) is the selected literal:

- \( l_i \) is a positive literal and let \( h \leftarrow b_1, \ldots, b_n \) be a variant of a clause in the program, and \( \theta \) be the m.g.u. of \( l_i \) and \( h \). Then the node has a child \( \leftarrow (l_1, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n, b_1, \ldots, b_n) \theta \)

- \( l_i \) is a negative literal \( l_i = \text{not } g \). If there is a finitely failed SLDNF-tree with initial goal \( g \) then the node has a child \( \leftarrow (l_1, \ldots, l_{i-1}, l_{i+1}, \ldots, l_n) \)

The procedure only selects negated literals if ground in order to avoid floundering [Lloyd, 1984], a source of incompleteness for SLDNF.

The completion semantics [Clark, 1977] is the first attempt to define a declarative semantics for SLDNF and one of the most influential approaches to the definition of a semantics for normal logic programs. It relies on a transformation of the original normal theory into a first order theory whose semantics define the one of the original program. Intuitively, the transformed theory groups the definition of predicates (all the rules that have a certain predicate in the head) into “if and only if” rules, thus asserting that the rules in the program are the only way to make a certain atom true.

Let us denote by \( \text{rules}(p, T) \) the set of all rules in \( T \) having an atom with predicate \( p \) in the head.

**Definition 2.1** Let \( T \) be a theory and \( p \) a predicate. If \( \text{rules}(p, T) = \emptyset \) then \( \text{comp}(p, T) \) contains the fact

\[ \forall X. \neg p(X) \]

If \( \text{rules}(p, T) = \{ r_1, \ldots, r_n \} \), where each \( r_i = p(t_i) \leftarrow \overline{b_i}, i = 1, \ldots, n \) then \( \text{comp}(p, T) \) contains the formula

\[ \forall X. p(X) \leftrightarrow (X = t_1 \land \exists Y_1. \overline{b}_1) \lor \ldots \lor (X = t_n \land \exists Y_n. \overline{b}_n) \]

where \( Y_i, 1 \leq i \leq n \) is the list of variables that occur in the body but not in the head of \( p(t_i) \leftarrow \overline{b}_i \). The completion \( \text{comp}(T) \) is given by the set of all the theories denoted \( \text{comp}(p, T) \) for each predicate \( p \) appearing in \( T \) together with the theory \( ET \), called equality theory. \( ET \) contains general axioms that define equality (see [Clark, 1977]).

Intuitively, supported models are those models that have some support in the logic program. Each atom that
is true in a supported model is justified by rules in the program. For example, given the theory \( \{ p \leftarrow \neg q \} \), 
\( q \) is a model but it is not supported, because the model makes the rule true but the truth of \( q \) is not supported 
in the theory. As shown in [Apt & van Emden, 1982], the completion of a normal program characterises the 
supported models, thus in this thesis we will refer to models of \( \text{comp}(T) \) as supported models.

**Example 2.2** Consider the following definite theory \( T \):

\[
\begin{align*}
p(f(X)) & \leftarrow p(X) \\
p(a) & \leftarrow r \\
p(b) & \leftarrow \neg r
\end{align*}
\]

Using the above definition we derive the following completion:

\[
\text{comp}(T) = \left\{ \begin{array}{l}
\forall X. p(X) \leftrightarrow (\exists X_1.X = f(X_1) \land p(X_1)) \lor (X = a \land r) \lor (X = b \land \neg r) \\
\neg r
\end{array} \right. 
\]

This theory together with ET logically entails for example \( p(b) \) and \( p(f(b)) \) but also \( \neg p(a) \) and \( \neg p(f(a)) \), 
and matches the expected conclusions derived by NAF on \( T \).

A completion theory in general can be inconsistent, as shown in Example 2.3. Also, in some cases \( \text{comp}(P) \) 
may not entail either \( f \) or \( \neg f \).

**Example 2.3** Consider the following theory \( T \):

\[
p \leftarrow \neg p.
\]

\( \text{comp}(T) \) is represented by the following formula \( p \leftrightarrow \neg p \). The formula is inconsistent and thus any theory 
containing \( T \) would be meaningless.

Despite these issues the completion semantics represents the semantics counterpart of widely used proof 
procedures for a large class of programs (see [Clark, 1977, Fages, 1994]).

The 3-valued completion semantics [Kunen, 1987] provides a solution to cases similar to Example 2.3. Under 
these semantics the program \( \{ p \leftarrow \neg p \} \) has a model \( \{ p^u \} \), where \( p^u \) means that the truth value of \( p \) is 
undefined. By adding this third truth value the program remains consistent, thus it can be used for meaningful 
inferences. The 3-valued completion semantics can be seen as a “weakening” of the 2-valued semantics (and 
also of the stable model semantics), in the sense that if \( M \) is a model under the stronger semantics, then it is 
also a model under the 3-valued completion semantics.
We omit the definition of the 3-valued completion semantics referring the reader to [Kunen, 1987] for further details. For some classes of logic programs the 3-valued and the previously described (2-valued) completion semantics coincide (see Section 2.6).

2.4.2 Stable model semantics

Stable models [Gelfond & Lifschitz, 1988] provide an alternative semantics for normal logic programs. We define the grounding of a program as \( \text{ground}(P) = \cup_{r \in P} \text{ground}(r) \), where \( \text{ground}(r) \) is obtained by substituting in all possible ways the variables occurring in \( r \) with elements in the Herbrand universe of \( P \) [Gebser et al., 2007]. \( \text{body}^+(r) \) and \( \text{body}^-(r) \) denote respectively the set of the positive literals and the set of the negative conditions in the body of \( r \). Let \( P \) be a normal program and \( S \) a set of ground atoms. Then the reduct of \( P \) relative to \( S \) is defined as \( P^S = \{ \text{head}(r) \leftarrow \text{body}^+(r) | r \in \text{ground}(P) \text{ and } \text{body}^-(r) \cap S = \emptyset \} \)

A set of ground atoms \( S \) is a stable model of a program \( P \) if \( S \) is the least Herbrand model of \( P^S \).

**Example 2.4** Consider the following program \( P \):

\[
p \leftarrow \neg q. \\
q \leftarrow \neg p.
\]

(2.6)

There are four possible interpretations as shown in the table below. Two of them are also least Herbrand models of the reduct and thus are also stable models for \( P \).

<table>
<thead>
<tr>
<th>( S )</th>
<th>( P^S )</th>
<th>( HM(P) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</td>
<td>{ p } \cup { q }</td>
<td>{ p, q }</td>
</tr>
<tr>
<td>{ p }</td>
<td>{ p }</td>
<td>{ p }</td>
</tr>
<tr>
<td>{ q }</td>
<td>{ q }</td>
<td>{ q }</td>
</tr>
<tr>
<td>{ p, q }</td>
<td>{ p, q }</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>

In general a normal theory can have more than one stable model. Consequences of a theory can be derived according to a brave semantics (true in at least one stable model) and credulous semantics (true in all stable models) [Bonatti, 2002].

Stable models are also called answer sets. The concept of answer set is the theoretical foundation of Answer Set Programming [Lifschitz, 2008]. Like Prolog, ASP is a knowledge representation technique that is oriented towards declarative problem solving. It combines a rich modelling language with powerful solving mechanisms based on satisfiability testing. ASP, unlike Prolog, resulted from the line of research regarding stable models.
The logic programming language used in this context is called *AnsProlog* (PROgramming in LOGic with ANswer sets) [Baral, 2003a].

Despite the overlap in the semantics of Prolog and ASP the two techniques greatly differ. In the first case a solution is derived as failure or success of a given query, together with a substitution of the variables in the query. In ASP a solution is encoded by a model of the specification.

An ASP program consists of a normal logic program plus some ASP constructs, like aggregates and optimisation statements [Gebser et al., 2011]. ASP solvers usually include or make use of a grounder that derives the grounding of the input theory. Usually the grounding process involves optimisation techniques that reduce the grounding instantiation, maintaining logical equivalence with the original program. The solver computes all the answer sets for the input theory. The usual workflow involves the processing of such answer sets from which it is possible to extract the predicates of interest and the desired answers for the problem modelled.

### 2.5 Abductive Logic Programming

Abduction is a recognised reasoning paradigm in AI with a wide range of applications such as fault diagnosis, legal reasoning and natural language processing. In particular, Abductive Logic Programming (ALP) [Kakas et al., 1992], the realisation of abductive reasoning in logic programming, has been successfully used to deal with incompleteness, common sense and default reasoning. ALP has also been used to characterise the nonmonotonic semantics of logic programs [Mancarella et al., 2002]. This makes ALP suitable for extensions of logic programming paradigms with a nonmonotonic reasoning component.

ALP addresses the problem of deriving an explanation, within a set of allowed explanations, that together with a given background theory entails some given observations. Additionally the newly derived explantions must respect some integrity constraint. We introduce the definitions used in this thesis to address an ALP problem (we refer to [Denecker & Kakas, 2002] and [Kakas et al., 1992] for more general definitions).

**Definition 2.2** An abductive logic program is a tuple \( \langle T, A, I \rangle \) where \( T \) is a normal logic program, \( A \) is set of atoms called abducibles and \( I \) is a set of integrity constraints.

We specify the set \( A \) either extensionally, by enumerating the elements, or by specifying a set of predicates. In this second case all the atoms that are constructed using predicates in \( A \) are abducibles.

**Definition 2.3** Given an abductive logic program \( \langle T, A, I \rangle \), an abductive logic programming task is a tuple \( \langle g, T, A, I \rangle \), where \( g \), called the goal, is a conjunction of literals. \( \Delta \) is an abductive explanation for the task if there exists a substitution \( \theta \) such that \( T \cup \Delta \models g\theta \); \( T \cup \Delta \models I \); and \( T \cup \Delta \) is consistent.

\[^3\text{In general *AnsProlog* supports disjunction in the head of clauses. In this thesis we restrict to normal theories.}\]
Both the notion of consistency and the entailment relation used are dependent on the particular semantics associated with the task.

A number of ALP systems have been proposed that address the problem of deriving an abductive explanations (see [Kakas et al., 1992] for a review). In this thesis we employ two practical solutions. The first is rooted in the completion semantics and relies on the ALP proof procedure SLDNFA (see Section 5.2). The second relies on the use of an ASP solver (see Section 6.3.1).

2.6 Which semantics?

In this chapter, we have introduced three different semantics, one for definite programs and two for normal programs. All of them are defined in terms of models and define a notion of entailment $\models$ that depends on them. So in principle, every time we use the symbol $\models$ we need to specify the associated semantics. The approach we take here is that whenever we do not specify a semantics, then the particular definition or result is independent of the semantics. For example, in Definition 2.3, the definition of an abductive task can be instantiated to the particular semantics of interest.

In other cases, for example when reporting results on completeness and soundness, a particular semantics is naturally associated with the computational infrastructure used. In these situations we specify the particular semantics involved. We provide some results that define classes of (normal) programs where the alternative semantics provided coincide so that the reader can abstract from the particular cases that make the different formulations of the semantics required and focus on the main results provided in this thesis.

For definite programs, the least Herbrand model naturally defines the semantics and stable model semantics and completion semantics also have a least model that coincides with it. In general for normal programs this is not the case as models do not agree on an intersection being also a model.

Most definitions for classes of normal programs are based on a notion of dependency amongst predicates, defined as follows:

**Definition 2.4** (Dependency, from [Apt & Bol, 1994]). The dependency graph $D_P$ for a normal program $P$ is a directed graph with signed positive and negative edges. For every rule in $P$ with the predicate $p$ in the head, there is a positive (resp. negative) edge from $p$ to $q$ in $D_P$ for each predicate $q$ in a positive (respectively negative) literal in the body.

The predicate $p$ depends positively (resp. negatively) on $q$ if there is a path in $D_P$ from $p$ to $q$ with only positive edges (resp. at least one negative edge). $p$ depends evenly (resp. oddly) on $q$ if there is a path in $D_P$ from $p$ to $q$ with an even (resp. odd) number of negative edges.
Definition 2.5 Let $P$ be a normal program and $q$ a goal:

- (Call-consistency). $P$ is call-consistent if no predicate in it depends oddly on itself.

- (Strictness). $P$ is strict with respect to $q$ if no predicate in $q$ depends both evenly and oddly on a predicate in $P$.

- (Stratification). $P$ is stratified (respectively locally stratified) if $D_P$ (resp. $D_{\text{ground}(P)}$, where $\text{ground}(P)$ denotes the ground instantiation of $P$) has no cycle with a negative edge in its dependency graph.

- (Tightness). $P$ is tight if the subgraph of $D_P$ that only contains positive edges is acyclic.

Acyclic programs are a subclass of locally stratified programs [Apt & Bezem, 1991]:

Definition 2.6 (Acyclicity). Let $P$ be a program. $P$ is acyclic if there exists a function $|| \cdot || : \mathcal{B}_P \rightarrow \mathbb{N}$ from ground literals to natural numbers, such that $||\text{not } a|| = ||a||$ for all $a \in \mathcal{B}_P$ and for every $h \leftarrow b_1, \ldots , b_n \in P$, $||h|| > ||b_i||, 1 \leq i \leq n$.

The following result establishes a class of problems for which the 3-valued completion and 2-valued completion coincide:

Theorem 2.1 ([Apt & Bol, 1994]) Let $P$ be a call-consistent and strict with respect to a query $q$. Then the 3-valued completion of $P$ logically entails $q$ iff the 2-valued completion of $P$ entails $q$.

The following theorems are useful to establish equivalence between stable and completion semantics [Fages, 1994, Apt & Bol, 1994]:

Theorem 2.2 Let $P$ be a normal program:

- If $M$ is a stable model for $P$ then $M$ is a minimal supported model for $P$.

- If $P$ is locally stratified, $P$ has a unique stable model.

- If $P$ is tight, every supported model of $P$ is a stable model of $P$.

2.7 Program transformations

It is out of the scope of this thesis to define a framework for program transformation. However some of the proofs here are based on transformations that preserve the set of models of a theory; in particular, we use a transformation called unfolding [Aravindan & Dung, 1995]. Having presented two different semantics for
normal programs, the completion semantics and the stable model semantics, we want to characterise the effects of the transformations we use on the intended model on the first case, and on all the models in the second.

Unfolding is a basic transformation that consists of applying a resolution step to a certain rule on a selected atom in all possible ways, replacing the original rule.

**Definition 2.7** Let \( P \) be a theory, \( h \leftarrow b_1, \ldots, b_n \in P \) a rule \( r \) and \( b_i, 1 \leq i \leq n \) a positive literal. We say we apply an unfolding transformation to \( r \) on \( b_i \) if \( P' \) is the theory obtained from \( P \) by replacing rule \( r \) with the \( p \) rules

\[
(h \leftarrow b_1, \ldots, b_{i-1}, b_{i+1}, b_n, a_1^{j}, \ldots, a_m^{j})\theta_j
\]

such that for each \( j = 1, \ldots, p \), the rule \( h_j \leftarrow a_1^{j}, \ldots, a_m^{j} \) is a variant of a clause in \( P \) with no variables in common with \( r \), and \( \theta_j \) is the m.g.u. of \( b \) and \( h_j \).

The sequence of programs \( P_0, \ldots, P_z, z > 0 \) is called a transformation sequence starting from \( P_0 \), where \( P_{k+1} \) is a program obtained from \( P_k \), \( 0 < k < z \) by applying unfolding.

**Theorem 2.3** (From [Aravindan & Dung, 1995]). Let \( P_0, \ldots, P_n \) be a transformation sequence, \( 0 < i \leq n \):

1. The (3-valued) completion semantics of \( P_n \) in an unfolding transformation sequence where no rule unfolds itself, is identical to that of \( P_0 \).\(^4\)
2. The stable model semantics of \( P_n \) is identical to that of \( P_0 \).

Thus the program obtained after a transformation sequence has the same semantics as the original program, i.e. it has the same stable and completion semantics as the original program. We say in this case a semantics-preserving transformation has been applied.

We introduce for convenience the following notation: let \( L \) be a set of atoms. Two logic programs \( P \) and \( Q \) are equivalent on \( L \), denoted \( P \equiv_{L} Q \), iff for each model \( M \) of \( P \), there exists one and only one model \( N \) of \( Q \) such that \( M \cap L = N \cap L \), i.e. the two models agree on the subset \( L \).

### 2.8 Conclusions

We have introduced the main concepts and the notation used in this thesis, giving an overview of the main results of logic programming and its nonmonotonic extension. Based on their respective theoretical foundations we

---

\(^4\)In [Shepherdson, 1992], the result is stated for the 2-valued completion. But, as also reported in [Bossi & Etalle, 1994], it is straightforward to verify that the proof also supports the 3-valued case.
have introduced the two currently dominant computational frameworks: resolution-based logic programming and ASP.

The main advantage of ASP is that the language is fully declarative, whereas the computation mechanism in Prolog is such that the relative order of clauses in the program and of literals in clauses can influence not only the performance, but also the final outputs (see also [Truszczyński, 2004]). Furthermore, Prolog support for negation is limited compared to AnsProlog [Baral, 2003b]. On the other side, computation in AnsProlog is based on the initial grounding of the given theory, while Prolog operates on the non-ground given theory. Prolog is thus able to operate on infinite domains and resolution is able to perform inference on the non-ground program. In general the type of inference is different: ASP encodings usually assume the existence of multiple models, while in Prolog using programs with multiple models is not common.
3 Inductive Logic Programming

The purpose of deductive reasoning is to derive consequences from premises. This kind of reasoning cannot lead directly to new knowledge, as it merely uses the premises to derive something that is already embedded in them. Inductive reasoning instead is used to generate hypotheses and is essential for scientific discovery performing the inference of general laws from particular instances. For example, deriving that customers that buy pasta are likely to buy tomato sauce from the observation of customer’s behaviour is a process of inductive generalisation. Inductive Logic Programming [Muggleton, 1995], [Lavrac & Dzeroski, 1994] is a machine learning technique that relies on logic programming as representation language, inheriting its theoretical basis. Its goal is to derive a logic theory from a set of given examples, thus realising inductive reasoning in a logic programming context. ILP addresses some limitations of machine learning techniques based on propositional representations, by including logical inference in the learning process and by using a more expressive representation language based on first-order logic.

3.1 Inductive Logic Programming and Machine Learning

The field of machine learning studies algorithms that are able to learn. The process of learning can be seen as the extraction of knowledge in some form from experience in order to improve the performance of a program with respect to a given task [Mitchell, 1997]. For example, improving the playing strategy at chess from past games is a machine learning problem. The knowledge in this case is a representation of the strategy, the experience comes from the previously played games, the task is playing chess and the performance can be measured as the ratio of won games.

Here we focus on supervised learning, and in particular on inductive concept learning. In the supervised learning setting the training data is represented by a set of examples \( e \in E \) that are labelled with a target value \( c(e) \). The learner tries to find a hypothesis function \( h \) such that a certain \( \text{loss}(h, E) \) function that measures the quality of the hypothesis with respect to \( E \) is minimised [Raedt, 2008]. In particular in concept learning \( c(e) \) assigns a class to the given example. In this case a possible loss function would count the number of errors, i.e. the number of examples \( e \) such that \( h(e) \neq c(e) \). The final hypothesis derived can be used for prediction.
of future observations or as a description of a model that underlies the observed data. As both predictive and
descriptive tasks share the same foundations and are closely related, the notions discussed in this thesis can
be applied to both. In ILP the target values are given as truth values and the function $h$ that is computed is
represented by a logic program.

ILP has two main distinctive features that differentiate it from other machine learning techniques. The
hypothesis is encoded as a logic program and the classification of the examples also relies on an additional
domain-dependent pre-established logic program that defines relevant concepts for the task. This has two
immediate advantages. Firstly it can be applied to complex domains where entities are to be modelled using
multi-relational representations. Secondly, the outcome of the learning is human readable and can be interpreted
by domain experts or by end users. Imagine the case of learning a set of rules of a firewall, that permit or deny
the access to a network. We can argue that predictive accuracy can be sacrificed in order to obtain rules that are
more readable and thus better controlled by a human supervisor.

ILP has been also used as a knowledge engineering tool. Often the task at hand is not the classification of
unobserved instances of a certain concept but rather the improvement of incomplete knowledge with respect to
some examples on which the conclusions of the current theory differ from the expected ones. In this case ILP
can be seen as an extension to structured hypotheses of ALP, where hypotheses are derived as explanations for
a certain scenario of the domain modelled. The nonmonotonic extension, addressed in this thesis, goes in both
these directions: the support of a more expressive and compressive representation language for the hypotheses
in machine learning tasks and the application of ILP to complex knowledge engineering tasks.

### 3.2 Inductive Logic Programming Problem Statement

Consider the following theory:

$$B = \begin{cases} 
  male(bob) \\
  female(eve) \\
  female(alice) \\
  female(sharon) \\
  father(bob, sharon) \\
  mother(eve, sharon) \\
  parent(X, Y) \leftarrow father(X, Y) \\
  parent(X, Y) \leftarrow mother(X, Y) 
\end{cases} \quad \text{(3.1)}$$
The theory includes a definition of *parent* and facts that model properties of entities of the domain, e.g. *bob* is *male* and *sharon* is female, and relationships between entities, e.g. *bob* is a *father* of *sharon*.

Suppose that the current theory must be extended because a new relationship is *observed*: \( \text{child}(\text{sharon}, \text{eve}) \), i.e. *sharon* is a *child* of *eve*. One possible approach consists of adding the new relationship to the theory. This would not be an acceptable solution if the knowledge must be used to predict further instances of the *child* relationship, e.g. whether \( \text{child}(\text{alice}, \text{eve}) \) is true. This diverges even more from the intuitive expected behaviour if the underlying semantics employs the CWA, since in this case the new theory would state that any other instance of *child* is false.

Inductive reasoning can be applied in this case in order to abstract a general rule, rather than just passively accepting the new fact. Assuming that the new information is correct, we know that a set of rules \( H \) is a possible candidate iff

\[
B \cup H \models \text{child}(\text{sharon}, \text{eve})
\]

otherwise \( H \) would not be able to classify correctly the only observation provided. The single rule set \( H_1 = \{ \text{child}(X,Y) \leftarrow \text{father}(Y,X) \} \) does not respect (3.2). The rules \( H_2 = \{ \text{child}(X,Y) \leftarrow \text{parent}(Y,X) \} \) and \( H_3 = \{ \text{child}(X,Y) \} \) instead respect (3.2) and are thus considered *inductive hypotheses* for the given observation. It is worth noticing the duality with deductive reasoning where \( H_2 \) or \( H_3 \) would be part of the input and \( \text{child}(\text{sharon}, \text{bob}) \) an output of the reasoning. The decision on which of the two solutions is more appropriate is also part of the learning process and can be based, for example, on prior likelihood or on human supervision. We can formalise this scenario as follows:

**Definition 3.1** An ILP task is defined as \( \langle E, B, \mathcal{L}_H \rangle \) where \( E \) is a set of positive and negative literals, called examples, \( B \) is a logic program called the background theory and \( \mathcal{L}_H \) is a set of logic theories called the language bias.

Based on the context we will use \( E \) to denote both the set of examples \( \{ e_1, ..., e_m, \text{not } e_{m+1}, ..., \text{not } e_n \} \) and also their conjunction. In general different constraints can be imposed on the language of the background knowledge adopted and on the language of the examples. The prevalent setting requires that the background knowledge is a definite theory and the examples are positive and negative ground literals. In this case the examples are usually provided as two sets \( E^+ \) and \( E^- \) containing positive literals, called respectively positive and negative examples\(^1\).

\(^1\)The particular setting presented here is called *learning from entailment*, and will be the focus of this thesis. Alternatively in *learning*
The definition of solution for an ILP task depends on the particular requirements of the problem being represented. The following definition gives a common characterisation:

**Definition 3.2** $H$ is a hypothesis for a given ILP task $\langle E, B, \mathcal{L}_H \rangle$ iff $H \in \mathcal{L}_H$, and $B \cup H \models E$.

Note that the above definition includes also the common case where $B$ and $H$ are definite theories and $E$ is a pair of sets of positive and negative examples, given that $B \cup H \models E$ iff $B \cup H \models e$ for each $e \in E^+$ and $B \cup H \models \text{not } e$, for each $e \in E^-$. Given a set of rules $H$, we say that $H$ covers an example $e \in E$ or not $e \in E$ iff $B \cup H \models e$.

We want to stress the similarities with the ALP task previously introduced (Definition 2.3). If $\mathcal{L}_H$ is constrained to be a set of atoms (rather than a set of rules), and the integrity constraints in Definition 2.3 is empty, then the definitions we have given of the two tasks are equivalent.

In practice, most ILP systems implicitly rely on a different definition of hypothesis that takes into account a quantitative measure of its quality. A convenient, simplifying way to see ILP tasks is as a search for one hypothesis or for a subset of hypotheses in $\mathcal{L}_H$. Either implicitly or explicitly, these hypotheses are associated with a quality measure that, in this thesis, is codified as a loss.

**Definition 3.3** $H$ is an optimal hypothesis for a given ILP task $\langle E, B, \mathcal{L}_H \rangle$ with respect to some loss function $l$ iff $H \in \mathcal{L}_H$, and $H \in \arg\min_H \{l(E, B, H')\}$.

In ASP it is expected that a certain solution is associated with more than one model. In these cases, rather than reasoning only on the hypothesis and the entailment relation with respect to the examples, it can be more appropriate to define the quality of the solution as a function of its models.

**Definition 3.4** Let $\langle E, B, \mathcal{L}_H \rangle$ be an ILP task and $\Gamma$ a function over answer sets. $H$ is a hypothesis with loss $l(E, B, H) = \Gamma(E, B, H, I_i)$ if $H \subseteq \mathcal{L}_H$, and there exists a non-empty set of answer sets $I_1, \ldots, I_n$ of $B \cup H$ such that $\Gamma(E, B, H, I_j) \leq \Gamma(E, B, H, I_i)$ for all $j = 1, \ldots, n$.

In other words we associate a loss with each hypothesis $H$, and the optimal hypothesis is the one with the minimum loss over all the answer sets of $B \cup H$. The type of learning inference considered here is brave [Sakama & Inoue, 2008], since a hypothesis has a certain loss value if there exists a model over which the hypothesis has that loss. However, in many cases, when $B \cup H$ belongs to the class of locally stratified programs (see Section 2.6), each hypothesis is associated with a single answer set.

We consider here two alternative loss functions:

---

1. from interpretation [Raedt, 2008], a less conventional setting, examples are given as Herbrand interpretations.
2. Also note that under the given assumptions and both completion and stable models semantics (that coincide in this case) $B \cup H \models \text{not } e$ iff $B \cup H \not\models e$ under logical entailment.
\[ l^{\text{complexity}}(E, B, H) = \begin{cases} 
\sum_{r \in H} |r| & \text{if } B \cup H \models E \\
+\infty & \text{otherwise} 
\end{cases} \]

\[ l^{\text{progol}}(E, B, H) = +\sum_{r \in H} |r| - |\{e \in E : B \cup H \models e\}| + |\{\text{not } e \in E : B \cup H \models e\}| \]

where \(|r|\), the complexity of the rule \(r\), is the number of literals in \(r\). The functions can be easily reported to the case of answer set semantics. For example, we can define the function

\[ \Gamma^{\text{progol}}(E, B, H, I) = +\sum_{r \in H} |r| - |\{e \in E : e \in I\}| + |\{\text{not } e \in E : e \in I\}| \]

for the loss function \(l^{\text{progol}}(E, B, H)\).

The first function counts the total number of literals in the rules of the hypothesis if all positive examples are in the answer set \(I\), it returns positive infinity otherwise. The second function resembles the scoring function in [Muggleton, 1995]: it is the sum of number of negative examples minus number of positive examples that are covered by \(H\), plus the number of literals in the hypothesis. The choice of the loss function depends on the nature of the task. \(l^{\text{complexity}}\) can be used for tasks of hypothesis finding, where the goal is to find all those hypotheses that cover all the positive and none of the negative examples, while \(l^{\text{progol}}\) function can be used in general for prediction tasks and tasks with noisy examples.

In general the function \(l\) can take into account the number of examples that are correctly classified by the current hypothesis and properties of the hypothesis as in the loss functions shown, but it can also be based on probabilistic semantics like [Sato, 1995].

Given the definition of an ILP task, it is common to reason on the soundness and completeness properties of systems employed to derive (optimal) solutions. Intuitively, an ILP system is sound if each of the hypotheses derived are a hypothesis, as defined in Definition 3.3. Conversely a system is complete if for all the hypotheses of the given ILP task, the system is able to derive them. We will instantiate these properties, considering specific assumptions, for the two systems presented in this thesis and discuss cases of incompleteness for other systems.

### 3.3 The language bias

The space of possible solutions is inherently large for all meaningful applications so different levels of constraints need to be imposed to restrict the search for hypotheses. When possible, besides the background knowledge about the modelled world, some a priori knowledge about the structure of the hypothesis can be employed to impose an instance-specific language bias \(\mathcal{L}_H\). In practice this is provided in the form of specification languages. The most common specification language is mode declarations, described in detail in the
Whenever the language bias does not require constraints on subsets of the hypothesis, but just on the structure of the rule, we can model the space based on the rules that are allowed by the language bias. The rule space $\mathcal{R}$ is the set of rules that can appear in a hypothesis, i.e. $H \in \mathcal{L}_H \iff H \subseteq \mathcal{R}$.

### 3.3.1 Mode declarations

*Mode declarations* are a common tool to specify a language bias, first introduced in [Muggleton, 1995].

**Definition 3.5** A mode declaration is either a head or body declaration, respectively $mode_h(s)$ and $mode_b(s)$ where $s$ is called a schema. A schema $s$ is a ground positive or negative literal that contains one or more placemarkers. A placemaker is a ground 1-ary function whose functor is one of the three symbols ‘+’ for input placemarkers, ‘−’ for output placemarkers, ‘#’ for constant placemarkers. The argument of the function is a constant called type.

Compared to the original definition, we extended the notion of mode declaration to the case of normal programs; thus a schema can be in general a ground positive or negative literal.

**Definition 3.6** Let $m$ be a mode declaration and $l$ a literal. $l$ is compatible with $m$ iff $l$ corresponds to the schema of $m$ where all the input placemarkers are replaced with variables; all output placemarkers are replaced with different variables; constant placemarkers are replaced with constants; variables replacing output placemarkers do not also replace input placemarkers.

Whenever literals are associated with mode declarations we will refer to variables (terms) that replace input and output placemarkers respectively as *input* and *output variables (terms)*. Also, the set of head mode declarations in $M$ is written $M^+$ and the set of body mode declarations is written as $M^−$.

**Definition 3.7** Let $r$ be a rule $h \leftarrow b_1, \ldots, b_n$ with a specific total order on the literals (ordered rule) in the body and $M$ be a set of mode declarations. $r$ is compatible with $M$ on $[m_h, m_{b_1}, \ldots, m_{b_n}]$ iff the following conditions are met: (i) $m_h = mode_h(s) \in M$ and $h$ is compatible with $m_h$ (ii) for each $b_i, i = 1, \ldots, n$, $m_{b_i} = mode_b(s) \in M$ and $b_i$ is compatible with $m_{b_i}$; (iii) every input variable in any of the literals is either an input variable in $h$ or an output variable in some literal $b_j, j < i$. The set of all compatible ordered rules for $M$ is denoted $\mathcal{R}_M^o$.

---

[3]In this thesis we do not discuss the use of output placemarkers in head mode declarations. We made this choice for three reasons. Firstly, such case would complicate some of the abstractions and notions used in the presentation but does not represent a challenging technical issue (the actual implementations of the systems presented in this thesis support it). Secondly, the use of output placemarkers in the head can be avoided by refactoring the set of mode declarations provided. Thirdly, our presentation follows the original definition of mode declarations that does not consider this case.
Literals in a compatible rule are always associated with a mode declaration that implicitly defines the type of the variables. Variables $X$ are denoted as $X^+$ whenever they replace input placemarkers and $X^-$ whenever they replace output placemarkers.

**Example 3.1** Consider the following three mode declarations

\[
M = \begin{cases} 
  m_1 : \text{modeh}(p(+\text{any})) \\
  m_2 : \text{modeb}(r(+\text{any}, \#\text{any})) \\
  m_3 : \text{modeb}(q(+\text{any}, -\text{any})) 
\end{cases}
\]

The ordered rule \( p(X^+) \leftarrow q(X^+, Y^-), r(Y^+, a) \) is compatible with \( M \). It is worth noticing that the rule \( p(X^+) \leftarrow r(Y^+, a), q(X^+, Y^-) \), semantically equivalent to the previous, is not compatible with \( M \).

Definition 3.7 is the original definition given in [Muggleton, 1995]. Note that the definition refers to ordered rules. We relate the definition to (non-ordered) rules as follows:

**Definition 3.8** A rule \( r \) is compatible with a set of mode declarations \( M \) iff there exists an ordered rule \( r' \in \mathcal{R}_M \) that has the same head and body conditions as \( r \).

The set of all the (non-ordered) rules compatible with a given set of mode declarations \( M \) is denoted as \( \mathcal{R}_M \) and defines the rule space.

**Definition 3.9** Let \( s \) be a schema. \( s^* \), called the variabilisation of \( s \), is the literal obtained from \( s \) by replacing all placemarkers with different variables.

**Example 3.2** Let \( m = \text{modeb}(\text{load}(\#\text{id}, +\text{car}, -\text{shape}, -\text{int})) \). Then \( m^* = \text{load}(W, X, Y, Z) \).

### 3.4 Generalisation and specialisation

Most ILP systems perform an informed exploration of the space of candidate solutions according to some generality notion on the rules. Some approaches that operate by specialising an overly general initial theory are called **top-down**, others where the initial theory is overly specific and generalised during the search are called **bottom-up**.

**Definition 3.10** A set of clauses \( H_1 \) is more general than another set of clauses \( H_2 \), denoted \( H_1 \leq H_2 \) if and only if all the examples covered by \( H_2 \) are also covered by \( H_1 \).
The definition of a generality relationship highly increases the effectiveness of the search since, as described in [Raedt, 2008], whenever a refinement operator is used that only generalises or specialise a current solution and the loss function associated with a certain hypothesis is a monotonic function of the generality, then a high portion of the search space can be pruned.

Let us focus for the moment on the most common setting, where the theories involved are definite. In this case, the generality notion can be captured by logical implication. Thus $H_1 \preceq H_2$ if and only if $H_1 \models H_2$. Whenever a background theory $B$ is involved we say $H_1$ is more general than $H_2$ relative to the background theory $B$, written $H_1 \preceq_B H_2$, if and only if $B \land H_1 \models H_2$. Thus in order to specialise a hypothesis $H$ (the reasoning for generalisation is dual), we could derive all the elements $H'$ in $L_H$ such that $H \preceq_B H'$. This would result in a very inefficient strategy, since it would derive theories $H''$, such that there exists a $H'$ and $H \preceq_B H' \preceq_B H''$. This is not desirable because $H''$ would also be a specialisation of $H'$ and thus considered twice during the search. In practice, most ILP systems perform a search over the rule space employing so called refinement operators that given a rule $r$ produce a refined rule $r' \in \rho(r)$. They are either specialisation $\rho^s$ or generalisation $\rho^g$ refinement operators. Focusing on specialisation (the generalisation case is dual), in order to avoid redundancy as explained before, it is usually required that $\rho^s(r)$ only derives those elements that are the most general specialisations i.e. $\rho^s(r) = \text{min}(\{r' \in \mathcal{R} : r \prec r'\})$, where $\text{min}(S)$ denotes all the elements $s$ in $S$ such that there is no $s' \in S, s' \prec s$. An additional requirement is that $\rho^s(r)$ is finite and computable. A refinement operator that satisfies both these requirements is called ideal [Torre & Rouveiroi, 1997].

ILP systems are characterised by the particular refinement operators employed. We refer to [Muggleton, 1995] and [Raedt, 2008] for an overview of specialisation and generalisation operators. Most ILP systems use a common notion of generality on clauses, called $\theta$-subsumption.

**Definition 3.11** Let $c$ and $d$ be clauses. We say $c$ $\theta$-subsumes $d$ if there is a substitution $\theta$ such that $c\theta \subseteq d$, denoted $c \preceq_\theta d$.

$\theta$-subsumption introduces a syntactic notion of generality, in the sense that it is only based on the literals in the clause, and is thus computationally easy to verify. $\theta$-subsumption satisfies the following property (soundness):

**Proposition 3.1** For all clauses $c, d$, $c \preceq_\theta d$ implies $c \models d$.

It is not complete in the sense that there may be a more specific clause that is not subsumed. In particular it is not complete with respect to self recursive clauses [Raedt, 2008].
3.5 Monotonicity and sequential covering

Traditionally, ILP addresses the particular case where the entailment notion used is monotonic [Lavrač & Džeroski, 1994]. In this case, the background knowledge is a definite program, the examples are positive and negative ground literals and the language bias only includes definite theories. The entailment notion used in this context is classical entailment.

This assumption is at the base of the sequential covering approach, shown in Algorithm 1.

**Algorithm 1** SEQUENTIAL-COVERING

**Inputs:** $E$ examples, $\mathcal{L}_\mathcal{H}$ language bias, $B$ background knowledge

**Outputs:** $H$ hypothesis

1. $E_{\text{cur}} = E$
2. $H = \emptyset$
3. while generalization stopping criterion is satisfied do
4. $c = h \leftarrow$
5. while specialization stopping criterion is satisfied do
6. $c = \text{REFINE}(c, \mathcal{L}_\mathcal{H})$
7. end while
8. $H = H \cup c$
9. $E_{\text{cov}} = \{e \in E_{\text{cur}} : B \cup H \models e\}$
10. $E_{\text{cur}} = E_{\text{cur}} \setminus E_{\text{cov}}$
11. end while
12. return $H$

The algorithm performs a search in the space of possible rules, usually adding conditions starting from a base atomic rule. Adding a condition to the rule restricts the set of examples covered by the current hypothesis. So based on this, rules resulting from adding a condition to the current one are further refined based on heuristics related to the loss. The rule is then added to the current hypothesis. In the algorithm, the choice of the refinement is not deterministic and can be a backtracking point. In general, different algorithms implement different types of search based not only on the particular refinement operator employed, but also on the choice of a refinement amongst alternative refinements of a certain rule. The monotonicity assumption is crucial because whenever a new rule is added (line 8), we know that we can safely remove all the examples that are entailed from the examples that are considered in a subsequent iteration (line 10), since they cannot be retracted as a consequence of new rules.

This critical assumptions that permit the use of a sequential covering approach no longer hold in the case of nonmonotonic entailment. Consider the theory $B = \{p \leftarrow \text{not } q, r\}$ and the examples $E = \{p, q\}$. Using the sequential covering method a first rule that covers $p$ can be derived as $r$. In fact $B \cup \{r\} \models p$. When deriving a rule for $q$, the only candidate is adding $q$ itself, since $B \cup \{r\} \cup \{q\} \models q$. However, because of non-monotonicity $B \cup \{r\} \cup \{q\} \not\models p$. From this simple example it is clear that the sequential covering method
cannot be applied to nonmonotonic problems as we cannot rule out examples that are covered in a previous iteration (in the example, \( p \) must be taken into account when deriving a rule for \( q \)).

The monotonicity of the entailment relationship is heavily exploited in most of the top-down ILP systems, relying on the fact that rules can be added sequentially to the hypothesis, under the assumption that the set of the examples entailed by the current hypothesis \( H \) will still be entailed for any hypothesis \( H' \supseteq H \). More in general, the monotonicity of the entailment enables the learning process to be monotonic in the sense of the following definitions:

**Definition 3.12 (Monotonicity of the iteration).** Let \( \langle E, B, \mathcal{R}_M \rangle \) be an ILP task. The search is monotonic for the ILP task iff for all \( c \in \mathcal{R}_M, H \subseteq \mathcal{R}_M \) and \( e \in E \), if \( B \cup H \models e \) then \( B \cup H \cup \{c\} \models e \).

Under monotonicity of the iteration, examples that are currently entailed by the current hypothesis can be safely discarded (line 10 in Algorithm 1). This is ensured under the assumptions defined by the following theorem:

**Theorem 3.2 (From [Fogel & Zaverucha, 1998]).** Let \( P \) be a normal theory, \( q(\overline{u}) \leftarrow \overline{b} \) a normal clause and \( p(\overline{t}) \) a ground atom: if \( P \cup \{q(\overline{u}) \leftarrow \overline{b}\} \) is strict and call-consistent and \( p \) depends evenly on \( q \) in \( D_P \), then \( P \models p(\overline{t}) \) implies \( P \cup \{q(\overline{u}) \leftarrow \overline{b}\} \models p(\overline{t}) \).

In the previous example, \( p \) does not depend evenly on \( q \). Thus when \( q \) is added to the theorem does not guarantee that \( p \) is still entailed. Note that the monotonicity of the entailment relation is a sufficient but not necessary condition for the use of a sequential covering approach. In general, under the restriction of Theorem 3.2, a sequential covering approach can still be employed. Relating it to an ILP task, the theorem requires that the examples use predicates that depend evenly on the predicates defined by the rules allowed in the rule space.

Within the **REFINE** procedure in Algorithm 1, the algorithm searches for a rule that together with the partial hypothesis derived so far minimises the loss function associated with the ILP task or correctly entails the examples. Often the algorithm specialises an initial overly general hypothesis. The following properties are used to prune part of the search space:

**Definition 3.13 (Monotonicity of the specialisation).** Let \( \langle E, B, \mathcal{R}_M \rangle \) be an ILP task. The specialisation is monotonic for the ILP task iff for all \( c, d \in \mathcal{R}_M \) such that \( c \preceq_\theta d \) and \( e \in E \):

- if \( P \models e \) and \( P \cup \{c\} \models e \) then \( P \cup \{d\} \models e \);
- if \( P \not\models e \) and \( P \cup \{c\} \not\models e \) then \( P \cup \{d\} \not\models e \).
In other words, under monotonicity of the specialisation, if an example is not covered at some point during the search, specialising a rule will not result in the example being covered. If the example is covered then specialising a rule will not result in the example not being covered. These conditions enable the use of pruning techniques based on the bounds on the maximum number of positive and negative examples that can be covered after refinements of a given rule, like in PROGOL.

The following theorem clarifies the conditions under which the specialisation under $\theta$-subsumption is monotonic:

**Theorem 3.3** (From [Fogel & Zaverucha, 1998]). Let $P$ be a normal theory, $c$ and $d$ be normal clauses such that $c \preceq_\theta d$ and $e$ be a ground atom. Suppose $P \cup \{d\}$ is strict and call-consistent. If $P \models e$ then $P \cup \{c\} \models e$ implies $P \cup \{d\} \models A$. If $P \not\models e$ then $P \cup \{c\} \not\models e$ implies $P \cup \{d\} \not\models e$.

Both strictness and call-consistency are ensured in the case of definite programs, while additional conditions are required whenever a nonmonotonic semantic is involved.

### 3.6 Classes of problems

The ILP task is a very general one. For example, characterising the direction of a train, westbound or eastbound, based on the size, number, position and other properties of the carriages (also known as the Michalski train problem [Michalski & Stepp, 1983]) is indeed very different from completing an Event Calculus [Kowalski & Sergot, 1986] formalisation with initiation and termination rules by observing the fluents and the events over time [Alrajeh et al., 2011]. In this section we define three notable features that characterise the type of problems that can be solved by ILP systems.

#### 3.6.1 Multi-predicate and recursive concept learning

In the most common case ILP systems (e.g. PROGOL and FOIL) address problems that involve a set of atoms of a single predicate $p$ as examples and the rule space $\mathcal{R}$ only includes rule that have atoms of such predicate $p$ as the head. In multi-predicate learning the examples and the rule space refer to a set of different predicates $p_1, \ldots, p_n$. In the general case it has shown that learning multiple predicates is harder than learning a single predicate [Raedt & Lavrac, 1996] in the sense that systems used for single predicate learning cannot be used also for this setting. This is intuitive since, in addition to the single predicate case, an ILP system is expected to discover dependencies between predicates, i.e. both derive a definition for a certain predicate and establish whether this new definition should be used in other rules being learnt. This is true in the case the language bias...
allows mutual definitions and where the dependencies are not hierarchically defined \textit{a priori}. Otherwise the problem can be split into multiple learning tasks, one for each predicate [Raedt & Lavrac, 1996].

As shown in [Malerba et al., 1998], multi-predicate learning is closely related to \textit{recursive} learning. The latter involves learning rules that depend on themselves (through rules defined in the background knowledge and other rules in the hypothesis or self-recursion). The recursive setting shares the same challenges as the multi-predicate setting since also in this case the learning system must support the discovery of the recursive dependency.

Operationally, recently proposed solutions for multi-predicate learning generalise existing ILP system and involve the use of abduction to produce additional examples to consider when generating interdependent rules in different iterations (e.g. [Kakas et al., 1998]) or by generalising the whole set of examples in parallel (e.g. [Malerba et al., 1998]).

3.6.2 Non-observational learning

In \textit{Non-Observational Learning} (NOL) [Moyle, 2002] the target concepts, namely the predicates allowed in the head of the hypotheses, and the examples do not necessarily refer to the same predicate. This amounts to having mode declarations that refer to different predicates from those that appear in the examples. For example in an Event Calculus [Kowalski & Sergot, 1986] based problem we may observe fluents at particular time points, codified by \textit{holdsAt} predicates but we may want to learn the definitions of \textit{initiate} and \textit{terminate}, i.e. how events cause a state change.

Some systems, like \textsc{foil}, assume a setting that does not encompass NOL. In other cases, like for example \textsc{progol}, extensions have been developed to handle NOL as discussed in Section 3.7. Other systems like \textsc{xhail} have been designed specifically to take into account this type of learning setting.

3.6.3 Predicate invention

\textit{Predicate invention} [Khan et al., 1998, Muggleton, 1994] is the process of creating new predicates that are not currently defined in the given theories or examples. Creating a new predicate can improve the compression – i.e. a measure of the ratio of the accuracy and the complexity of the final solution – by defining a new concept that can be used in the clauses that are learnt. Besides, some problems can only be solved if predicate invention is allowed [Stahl, 1994].

When mode declarations are used, predicate invention can be captured by allowing the ILP system to learn definitions for predicates that do not appear in the modelled domain. Practically, generic predicate names can be added in new head mode declarations. This is used in [Khan et al., 1998], where predicate invention is tackled
through “repeated learning”. In this case the system is given an appropriate bias in the search to compensate for the fact that the use of a predicate that is not yet defined does not improve locally the partial hypothesis being constructed.

The main problem in predicate invention is that, given that a potentially infinite number of new predicates can in principle be added, the search space becomes much larger than in cases where it is ruled out. Additional bias and constraints are required to render this technique practically applicable.

3.7 ILP systems

In this section we provide an overview of some ILP systems that are closely related to this thesis or that are used for comparison. The ILP approach we pursue is based on top-down search, thus we will concentrate our presentation on the techniques and the systems that support this type of approach. In this section we present more in depth four ILP systems, including some short introductory remarks on their limitations. We discuss more in detail other ILP systems in the other Chapters of this thesis. In particular systems that are based on abduction are discussed in Chapter 4 and 5. Systems that use ASP are discussed in Chapter 6.

Amongst those based on a bottom-up approach, we mention GOLEM [Muggleton & Feng, 1990], a bottom-up system based on relative least general generalisation (RLGG). RLGG generalises pairs of examples considering the facts given as background knowledge, creating the least upper bound of the two, in the $\theta$-subsumption lattice. The search in GOLEM greedily selects the clause with best coverage over those created by relative least general generalisation.

Bottom-up systems proved successful in some applications [Muggleton et al., 1992], but they tend to produce overly specific solutions and usually support constrained forms of the input theories (e.g. requiring determinacy [Muggleton & Feng, 1990], a restriction that, in general terms, requires that goals referring to predicates appearing in the language bias are allowed to succeed at most once for a given instantiation of the variables) [d’Amato et al., 2009]. These limitations were amongst the motivations that lead to the development of PROGOL and other more recent techniques based on top-down search.

3.7.1 Progol

Progol [Muggleton, 1995] is considered one of the most successful and influential ILP systems, as proved by its applications (e.g. [Muggleton, 2005], [Cootes et al., 2001], [Tamaddoni-Nezhad et al., 2006]) and systems that extend its original algorithm (e.g. [Muggleton & Bryant, 2000, Moyle, 2002]). It is based on a preliminary computation of a rule that is used as lower bound for a top-down specialisation within a sequential covering
iteration. The search is guided by a heuristic based on the coverage of the examples and the complexity of the rule and guarantees optimality with respect to this heuristic function.

**Mode-Directed Inverse Entailment**

The task of inducing a hypothesis can be seen as the dual task of performing deduction. Instead of deriving conclusions from a set of rules, the objective in this case is to compute the rules that can be used to derive the conclusions. Many of the early ILP systems are based on operators that performed some type of inverted deduction. These approaches inspired *Mode-Directed Inverse Entailment* (MDIE).

The idea behind MDIE is that given the generic ILP task, where \( h \) and \( e \) are both Horn clauses and \( B \) is a Horn theory, we want to derive \( h \) such that \( B \cup \{h\} \models e \). This is logically equivalent to \( B \cup \{\neg e\} \models \neg h \). This means that the negation of the hypothesis can be deductively obtained from \( B \) and \( \neg e \). Let \( \bot \) be the negation of the so called *most specific clause* (or *bottom clause*) \( \bot \). \( \bot \) is the potentially infinite conjunction of all the ground literals that are true in all models of \( B \cup \{\neg e\} \). The clause \( h \) must contain a subset of the ground atoms in \( \bot \), i.e. \( B \cup \{\neg e\} \models \bot \models \neg h \). So \( \bot \) can be used to derive \( h \), since \( h \models \bot \). A lower bound for the search can be derived by negating \( \bot \), obtaining the clause made of all the negated consequences of the theory, i.e. \( \{\neg l : l \text{ is a literal and } B \land \neg e \models l\} \).

The most-specific clause \( \bot \) can be infinite and so far we have not considered the mode declarations. In practice, Progol searches for the most specific clause \( \bot \in \mathcal{R}^i \), where \( \mathcal{R}^i \) is the subset of the rule space of a set of mode declarations \( M \) where all the rules have variables with maximum depth \( i \). The variable depth \( d(X) \) of a variable \( X \) in a rule \( r \) is given by:

\[
d(X) = \begin{cases} 
0, & \text{if } X \text{ is in the head of } r \\
\max_{Y \in U_X} d(Y) + 1, & \text{otherwise}
\end{cases}
\]  

(3.3)

where \( U_X \) is the set of variables in atoms in the body of \( r \) containing \( X \).

**Search**

A subset of the hypotheses such that \( h \models \bot \) can be obtained using \( \theta \)-subsumption. So after the most specific clause is derived, Prolog generalises it performing a top-down search that is bound by \( \bot \) from below and by the empty clause from top. A specialisation operator is applied on the clause, that adds literals to it while maintaining \( \theta \)-subsumption over the bottom set, i.e. \( h \preceq_{\theta} \bot \) and compatibility with mode declarations, i.e. \( h \in \mathcal{R}_i \). We recall from [Muggleton, 1995] the definition of the specialisation operator used in PROGOL. A variable is *splittable* in a compatible rule if it is an input variable in the head or an output variable in the body.
Also, an order on the literals in \( \bot \) is assumed (since \( \bot \) is an ordered compatible rule), \( \bot(k) \) is the \( k \)-th element in \( \bot \). Since \( h \not\subseteq \bot \), there must be a substitution \( \theta \) such that for each literal in \( h \), there is a literal \( l \) in \( \bot \) and \( h\theta = l \). Given a clause \( r \in \mathcal{R} \) and a substitution \( \theta \) such that \( r\theta \subseteq \bot \), \( \rho_s \), the specialisation operator, uses an intermediate operator \( \delta \) defined as follows\(^4\): \( \langle p(v_1, \ldots, v_m), \theta_m \rangle \in \delta(\theta, k) \) iff

- \( p(u_1, \ldots, u_m) \) is the \( k \)-th literal in \( \bot \)
- \( \theta_0 = \theta \)
- if \( v_j / u_j \in \theta_{j-1} \) then \( \theta_j = \theta_{j-1} \) for \( 0 < j \leq m \)

The refinement operator takes into account a substitution and the index of the literal in \( \bot \) and is defined as follows \( \langle r', \theta', k' \rangle \in \rho(\langle r, \theta, k \rangle) \) iff either

\[
r' = c \cup \{l\}, k' = k, \langle l, \theta' \rangle \in \delta(\theta, k), r' \in \mathcal{R}
\]

or

\[
r' = c, k' = k + 1, \theta' = \theta
\]

for \( 1 \leq k \leq |\bot| \).

More than one specialisation is possible in general, thus the search can be seen as a tree where each node is a clause and all the children are specialisations. The search is handled by an \( A^* \) type search that is guided by a compression measure (similar to the function \( l_{\text{progol}} \) previously defined). The search ultimately returns the optimal solution according to this measure.

**PROGOL extensions**

PROGOL\(^5\) \cite{muggleton2000}, uses Theory Completion using Inverse Entailment (TCIE), a technique that extends the given background theory with the use of contrapositives. Each clause of the type \( h \leftarrow b_1, b_2 \) is equivalent to \( \neg b_1 \leftarrow \neg h, b_2 \) and \( \neg b_2 \leftarrow \neg h, b_1 \). This cannot be directly represented in a definite program, but it can be simulated using a new predicate \( \text{not} \_p \) in place of a predicate \( p \) used in a negative literal of the type \( \neg p(...) \). Thus given a theory \( P \), a new theory \( P^c \) is generated that includes all the contrapositives and then used to calculate the negated most specific clause \( \neg \bot \) that includes now also predicates of the type \( \text{not} \_p \). These predicates are transformed into \( p \) in \( \bot \) and are thus positive literals that can be used in the head of hypotheses generalised from \( \bot \).

\(^{4}\)We do not consider variable splitting, assuming this is already taken into account by adding appropriate equalities to the mode declarations.
ALECTO [Moyle, 2002] uses abduction to extend the base PROGOL algorithm in order to deal with NOL. The background theory used is a Horn theory. For each positive example \( e_i \), an abductive explanation \( \Delta_i \) is derived. The union of these explanations is called the start set \( \phi = \bigcup \Delta_i \). The start set takes the role of the examples and is generalised to produce the final hypothesis.

PROGOL, the PROGOL5 extension and ALECTO are implemented in ALEPH, a prototype system that is able to emulate functionality of several ILP systems [Srinivasan, 2003].

Remarks

The shortcomings of PROGOL are discussed in detail in [Ray, 2005]. In particular this procedure is known to be incomplete, i.e. it is possible that some literals are not computed using this method but should be part of \( \bot \). Also, the ordering on \( \bot \) can cause incompleteness in the search as shown in [Muggleton, 1995]. Although PROGOL is recognised as a very effective system for a wide class of ILP problems, it cannot be used without missing relevant solutions in cases where a normal program is involved. Furthermore, it does not support multi-predicate learning although it can be employed for predicate invention under repeated guided learning sessions [Khan et al., 1998]. NOL is supported by ALECTO. Nevertheless, as reported in [Moyle, 2002] and discussed in [Ray, 2005], ALECTO cannot be used to derive abductive explanations from negative examples and involving normal clauses.

3.7.2 XHAIL

XHAIL [Ray, 2009b] is the ILP system that relates the most to the solutions presented in this thesis. Although the underlying procedure is radically different, it supports nonmonotonicity and uses ALP as the underlying computational mechanism. XHAIL was designed as an extension of the HAIL procedure [Ray, 2005] that was devised in order to overcome the limitations related to incompleteness in PROGOL5. Instead of generating a most specific clause, XHAIL constructs a most specific theory, called a kernel set.

XHAIL constructs a set of hypotheses in three phases. Each of these phases is computed in terms of an ALP task instantiated with a theory, a goal and integrity constraints specific to the particular phase.

**Abductive Phase** In the first phase, the head mode declarations are used to construct the set of abducibles of an abductive task \( \langle E, B, \mathcal{R}^+, \emptyset \rangle \). The set of abducibles determines the facts that can be formed using head mode declarations \( \mathcal{R}^+ \). The output of this phase is a set of abducibles \( \Delta^a = \{h_1, ..., h_n\} \). Note that this is a nondeterministic task; in general one or more solutions are possible. The algorithm gives priority to subset-minimal solutions and then generates also other solutions.
**Deductive Phase**  The second phase constructs a kernel set:

\[
K = \begin{cases} 
    h_1 \leftarrow b_{1,1}, \ldots, b_{1,m_1} \\
    \quad \ldots \\
    h_n \leftarrow b_{n,1}, \ldots, b_{n,m_n} 
\end{cases}
\]  \hspace{1cm} (3.4)

Each atom \(h_i \in \Delta^a, 1 \leq i \leq n\) is computed in the abductive phase. The deductive phase completes the rules by computing each \(b_{i,j}\) for \(1 \leq i \leq n, 1 \leq j \leq m_i\), such that \(B \cup \Delta^a \models b_{i,j}\). This is a deductive task, or, as reported for uniformity and elegance in XHAIL an abductive task with empty set of abducibles \(g_i, B \cup \Delta^a, \emptyset, \emptyset\).

Goals \(g_i\) are generated by considering a set of terms \(t_i\) that is initialised using the input terms in the head \(h_i\) and extended with output terms. Each goal \(g_i\) is obtained from a body mode declaration where input placemarkers are replaced by terms in \(t_i\). The ground successful goals are then added to the body of the corresponding rule as literals \(b_{i,j}\).

As a last step in this phase the kernel set, which is a set of ground rules, is transformed into a pseudo-kernel \(K'\) where different ground terms are replaced by distinct variables.

**Inductive Phase**  In the third phase, the hypothesis is computed by searching for a compressive theory \(H\) that subsumes the pseudo-Kernel \(K'\), is consistent with the background knowledge, falls within the hypothesis space and satisfies the given examples. The subsumption is performed by deleting literals from the body of rules in \(K'\). Computationally this is achieved using a theory transformation that enables an abductive search for the target theory \(H\): a special atom predicate \(use(i, j)\) for each body condition \(b_{i,j}\) is introduced such that whenever an abductive answer is derived that contains \(use(i, j)\), the body condition is kept, otherwise it is deleted.

**Remarks**

XHAIL has been successfully applied to a number of nonmonotonic problems, and although it is not publicly available it is, to the best of our knowledge, the first nonmonotonic ILP system fully implemented. Despite this, it has several limitations that have motivated some of the novelties introduced in this thesis.

Firstly, although the system addresses the completeness limitations of PROGOL5, it is not supported by a completeness result. This is due to the use of inverse entailment that is constructed on classical first-order semantics and cannot be applied as it is to nonmonotonic semantics. Also, the transformation of the kernel set into a pseudo-kernel set may in principle exclude some rules that \(\theta\)-subsume the rules in the...
kernel set. Furthermore, the use of $\theta$ subsumption for generalisation is itself a source of incompleteness [Nienhuys-Cheng & de Wolf, 1997].

Secondly, the use of arbitrary abductive hypotheses on one side expands the space of solutions that can be derived by XHAIL, but on the other it causes an arbitrary number of kernel sets to be constructed that can potentially be generalised in a solution. In a worst case scenario the kernel set, intended to be used to constrain the generalisation search, becomes a source of heavy nondeterminism. This is further explained and exemplified in Section 5.8.

Thirdly, the system does not derive optimal hypotheses. This is not an inherent limitation but the result of a strategy that halts the search whenever a non-empty set of solutions is derived for a given kernel set. Potentially kernel sets resulting from non-minimal abductive explanations in the first phase may result in a more compressive hypothesis. In addition to this, the notion of optimality employed cannot be chosen by the user based on the specific problem and does not tolerate the presence of noise in the examples.

Lastly, the system has not been validated empirically on a benchmark of nonmonotonic ILP problems. Given the low maturity of the field this is not a trivial task. Besides addressing the aforementioned limitations, in this thesis we collected a number of problems and showed that the techniques here introduced are able to derive useful solutions in cases where other systems cannot be applied or produce results with a predictive accuracy that is comparable to that of other systems.

3.7.3 FOIL

FOIL (First Order Inductive Learning) [Quinlan & Cameron-Jones, 1995] is an upgrade to a first order representation of an earlier propositional system [Quinlan, 1993]. It employs a covering approach where rules are refined according to an information based search heuristic. The hypotheses language is limited to rules that do not contain functions and constants and whose heads have the same predicate as the examples (OPL).

The particular implementation of DERIVE-NEW-RULE (Algorithm 1) proceeds as follows. A rule $r_0$ is initially given by the rule with an empty body $p(X)$. Since FOIL performs OPL and single predicate learning, $p(X)$ is derived from any of the given examples. At a given iteration $i$, the algorithm seeks a condition $c_i$ to add to the body of the current rule $r_i = p(X) \leftarrow b$ based on the local training set $E_i$ ($E_0 = E$). The condition $c_i$ is selected amongst the (possibly negated) atoms $q(Y)$ derived from the mode declarations. The variables used in the conditions are either variables appearing in $B$ or new variables. The choice of $c_i$ is based on the weighted information gain heuristic defined below.

**Definition 3.14** Let $r$ be a rule and $E$ and $E^{+c}$ respectively be the set of (positive and negative) examples and
of positive examples covered by \( r \). The information of \( r \) is given by

\[
I(r) = \frac{|E^+ + c|}{|E|}
\]

**Definition 3.15** Let \( r \) and \( r' \) be two rules. The weighted information gain (WIG) is given by

\[
WIG(r', r) = |E^{++c}|(I(r') - I(r))
\]

where \( E^{++c} \) is the set of positive examples covered both by \( r \) and \( r' \).

At each iteration \( i \) the condition \( c_i \) that maximises the WIG is chosen.

**Remarks**

FOIL targets a restricted class of problems, namely OPL on a function-free first order language. This makes it particularly efficient, but on the downside it reduces its applicability. Furthermore it adopts a greedy search strategy that makes it suitable for problems where suboptimal hypotheses are acceptable. FOIL cannot be applied to problems with scarce noise-free examples where complex, possibly recursive and multi-predicate hypotheses are targeted, and to nonmonotonic ILP problem.

3.7.4 Toplog

Compared to the other ILP systems presented so far, TOPLOG [Muggleton et al., 2008] is much more recent and comparatively less validated but it is particularly influential for the work presented in this thesis. TOPLOG implements an approach called *Top-Directed Hypothesis Derivation (TDHD)* where a logic program called the *top theory* \( T \) is used as a first-order declarative bias to define the hypothesis space.

The top theory uses *terminal* predicates that can appear in the final hypothesis, and *non-terminal* predicates that are used to structure the hypothesis space similarly to non-terminal symbols in context free grammars. It is either provided as user input or derived from a given set of mode declarations. Toplog searches for a hypothesis \( H \) such that for each \( h \in H \)

\[
T \models h \tag{3.5}
\]

\[
B \cup \{h\} \models e \tag{3.6}
\]

TOPLOG first generates all the hypotheses \( H_e \) that are generalisations of some positive example \( e \), construct-
ing a set $H_c = \{ h_e : e \in E \}$. Then it chooses a subset $H \in H_c$ that maximises a compression-based score function using a greedy approach, taking into account also negative examples.

The mechanism that generates a candidate hypothesis $h_e$ is based on the SLD-refutation of the example. The clauses from $\top$ that are used in the derivation are collected and merged at the end of the proof to generate the final hypothesis as shown in the following example.

**Example 3.3** Consider the following ILP problem:

$$
\begin{align*}
B &= \{ b_1 : \text{can}(a, fly). \\
&\qquad b_2 : \text{can}(b, swim). \\
&\qquad b_3 : \text{ability}(fly). \\
&\qquad b_4 : \text{ability}(swim). \} \\
E &= \{ \text{not penguin}(a). \\
&\qquad \text{penguin}(b). \} \\
M &= \{ m_1 : \text{modeh}(\text{penguin}(+\text{any})) \\
&\qquad m_2 : \text{modeb}(\text{can}(+\text{any}, \#\text{ability})) \}
\end{align*}
$$

The following top theory is produced, where $\text{body}$ is a non-terminal symbol:

$$
\begin{align*}
\top &= \{ \top_1 : \text{penguin}(X) \leftarrow \text{body}(X) \\
&\qquad \top_2 : \text{body}(X) \\
&\qquad \top_3 : \text{body}(X) \leftarrow \text{can}(X, C), \text{ability}(C) \}
\end{align*}
$$

There are three SLD-refutations for the positive example $\text{penguin}(b)$, namely $[\top_1, \top_2]$ and $[\top_1, \top_3, b_2, b_4]$.

Toplog extracts from these, two candidate hypotheses $\{ H_1, H_2 \} = H_{\text{penguin}(b)}$:

$$
\begin{align*}
H_1 &= \{ \text{penguin}(X) \} \\
H_2 &= \{ \text{penguin}(X) \leftarrow \text{can}(X, swim) \}
\end{align*}
$$

Amongst these, $H_2$ maximises the compression-based score and thus is selected as best candidate.
Remarks

TOPLOG does not define how negation is handled. Extending the behaviour to nonmonotonic theories, using SLDNF does not produce the desired result. Cases where a rule of the top theory is used within a failure phase require an extension of the base algorithm. Furthermore nonmonotonic ILP problems often involve the learning of rules needed to entail some negative examples, that are not considered in the hypothesis generation phase of TOPLOG.

3.8 Conclusions

In this Chapter we have presented some notions of ILP, presenting a general setting and a common framework that is the basis of most ILP systems. We defined the conditions under which the search can be performed monotonically, by adding elements sequentially to the set of hypotheses being constructed. We discussed the importance of the assumption of monotonicity in the existing implementations.

We surveyed four ILP systems based on top-down search, that can be considered particularly influential and that relate the most to the content of this thesis. These systems will be used to illustrate our contribution in the next chapters through discussions on similarities and differences and examples.
4 Abductive Learning

The main goal of this chapter is to present a general framework for ILP that is based on abductive reasoning. More specifically we introduce a mechanism to map an ILP task, where the language bias is specified by means of mode declarations, into an equivalent ALP task. The solutions for the ALP task can be mapped back into solutions of the original ILP problem. This transformation can be seen as an alternative and equivalent characterisation of the ILP problem that can be used to reason about properties of the task as well as a general framework for the implementation of learning systems. We present the transformation together with some explanatory examples and prove that it preserves the semantics of the original ILP problem. Finally we relate our approach to existing work on the use of abductive reasoning in ILP.

4.1 Induction as abduction

The relationship between abductive and inductive reasoning has been subject of intense debate. Together with deduction, these forms of reasoning were considered by C. S. Peirce “the three grand classes of inference” [Peirce, 1931]. Abduction is the reasoning process that derives possible explanations from observations, or that derives assumptions that would lead to a given conclusion. Induction, on the other hand, from given known instances, derives generalisations of a certain concept. Induction and abduction only produce conjectures, differently from deduction that deals with correct forms of reasoning. The duality between conclusions and instances and between assumptions and generalisation suggests the two processes have much in common. Although a thorough discussion on the nature of the two reasoning processes is not in the scope of this thesis (for a deeper treatment we refer to [Flach & Kakas, 2000]), it is also clear from the description of the realisations of the two reasoning mechanisms in the world of logic programming (Chapters 2 and 3) that the two have common characteristics.

Whether inductive and abductive reasoning should even be addressed as the same process is also a matter of debate. Despite the similarities, as also observed in [Kakas & Flach, 2009], the computational models that have been in use in AI for the two reasoning mechanisms are distinct. Whenever the two have been integrated in the same reasoning process, abductive proof procedures have been employed either as components of ILP
systems (e.g. [Esposito et al., 2000]) or extended for inductive reasoning (e.g. [Kakas & Riguzzi, 2000]) as later discussed in Section 4.4. In contrast to these existing approaches, we propose a new ILP approach based on a translation into an equivalent ALP instance. The transformation relies on a set of mode declarations that implicitly define the space of candidate solutions for an ILP task. The solutions of the abductive task can then be translated back into a solution for the original ILP problem. Under this viewpoint the inductive process can be seen as a “meta-abductive” reasoning on the structure of the rules involved rather than on elements of the domain, given that this space is adequately defined (in our case by a set of mode declarations). The main advantage is the generality of the framework that, supported by results on soundness and completeness, can be used to derive inductive solutions for nonmonotonic problems and handle learning of recursive, multi-predicate and non-observational theories.

4.2 Rule representation

The type of ILP problem we address in the thesis is always associated with a set of mode declarations $M$ that characterises the space $R_M$ of the rules that can be part of hypotheses, as shown in Definition 3.7. The established definition of mode declarations, also used in this thesis, defines a total order over conditions in a rule. In fact, the definition requires that every variable that replaces an input placemarker in any of the literals in the body $b_i$ replaces an output placemarker in some atom $b_j, j < i$. This implies that $b_i$ must appear after $b_j$ in the rule whenever $b_i$ uses output variables from $b_j$. A logic rule corresponds to one or more ordered rules that are semantically equivalent. On the other hand, any ordered rule can be associated with only one equivalent (unordered) rule, given by the set of all the literals in the rule.

We recall from Chapter 2 that we denote the set of rules and ordered rules that are compatible with a set of mode declarations $M$ respectively as $R_M$ and $R^o_M$. Also, whenever writing ordered rules, the order of the conditions is given by the intuitive order – left to right – of the conditions in the rule. Notably, not necessarily all the total orders on the elements on the set will result in a rule that is compatible as shown in Example 3.1.

A set of mode declarations defines the space of rules that can be considered in candidate solutions of an ILP task, called rule space. Under this assumption it is possible to map rules that are allowed to appear in candidate hypotheses into an equivalent representation that explicitly refers to mode declarations. This ultimately simplifies the study of the hypothesis space and can be used within an abductive reasoning process.

We assume that each mode declaration $modeh(s)$ (resp. $modeb(s)$), has a unique identifier $m$, made explicit by the following notation $m : modeh(s)$ (resp. $m : modeb(s)$). When no ambiguity arises, with abuse of notation we use the identifier to denote the schema of the mode declaration. Given a literal $l$ and a mode
declaration \( m \) such that \( l \) is compatible with \( m \) we denote with \( \text{inp}(l, m) \), \( \text{out}(l, m) \), \( \text{con}(l, m) \) respectively the lists of terms in \( l \) that replace input placemarkers, output placemarkers and constant placemarkers in \( m \) in order of appearance, left to right. \( \text{type}(l, m) \) denotes the set of literals \( t_1(u_1), \ldots, t_n(u_n) \) such that \( t_i \) is the type of the placemaker replaced by the term \( u_i \) in \( l \), for \( i = 1, \ldots, n \).

**Example 4.1** Let \( l = \text{load}(A, B, 8) \) and \( m : \text{modeb}(\text{load}(+\text{car}, -\text{shape}, -\text{int})) \). Then \( \text{inp}(l, m) = [A] \), \( \text{out}(l, m) = [B, 8] \), \( \text{con}(l, m) = [] \) and \( \text{type}(l, m) = \{ \text{car}(A), \text{shape}(B), \text{int}(8) \} \).

It is our goal now to define a structure that captures all the relevant information that codifies all the compatible literals with respect to a given set of mode declarations.

**Definition 4.1** Let \( M \) be a set of mode declarations. A mode-based literal (MBL) is a tuple \( (m, c, i, o) \), where \( m \in M \), \( c \) is a list of constants with as many elements as the constant placemarkers in \( m \) and \( i \) and \( o \) are two lists of variables with as many elements respectively as the input and output placemarkers in \( m \). The lists \( i \) and \( o \) do not have any element in common and all the variables in \( o \) are different. We say a MBL \( f = (m, c, i, o) \) represents a literal \( l \) iff \( l \) is obtained from \( f \) by replacing all the input, output and constant placemarkers in \( m \) in order of appearance left to right respectively with the elements in \( i \), \( o \), \( c \).

**Example 4.2** Consider again the mode declaration \( m : \text{modeb}(\text{load}(+\text{car}, -\text{shape}, -\text{int})) \). The MBL \( (m, [], [A], [B, C]) \) represents the literal \( \text{load}(A, B, C) \).

A useful property of such a representation is that, given a compatible ordered rule, such a representation uniquely characterises the literals that appear therein. First we must restrict the acceptable sets of mode declarations, in order to avoid redundancies.

**Definition 4.2** A set of mode declarations \( M \) is normalised if for each compatible rule \( r \), if \( r \) is compatible with \( M \) both on \( [m_1, \ldots, m_n] \) and on \( [m'_1, \ldots, m'_n] \) then \( m_i = m'_i, 1 \leq i \leq n \).

Intuitively, for a set of mode declarations to be normalised it is sufficient that, for each pair of mode declarations, the schemas are different modulo the types, or, at least one of the two constant placemarkers refers to different types in the two mode declarations for which there is no constant that satisfies both. In the following we always assume that the sets of mode declarations are normalised.

**Theorem 4.1** (Representation correctness). Let \( M \) be a set of mode declarations, \( r \) a compatible ordered rule and \( l \) a literal in \( r \) compatible with a mode declaration in \( M \). Then there exists one and only one MBL that represents \( l \).
Proof If \( l \) is compatible with \( m \in M \), we can replace each input and output placemaker in \( m \) with variables in \( l \). These variables can be collected into the lists \( i, o \) with the defined order. Similarly all the constants replacing constant placemarkers are replaced by elements in \( c \). The MBL is completed with the identifier of \( m \). Thus there is always a way to construct a representation for a literal.

We need to prove that there is only one representation. Let us suppose \( h ← b_1, ..., b_n \) is compatible with \( M \) on \([m_0, m_1, ..., m_n]\) and that there is a literal \( l_j, 1 ≤ j ≤ n \), such that two MBL \( \langle m_j, c, i, o \rangle \) and \( \langle m_j, c', i', o' \rangle \) represent \( l_j \). At least one of the three lists in the MBL must be different. \( c ≠ c' \) would result in a different literal contradicting the initial assumption. Suppose one of the elements \( X \) in \( i' \) differs from the corresponding element \( X' \) in \( i \). Then if the rule is compatible, both \( X \) and \( X' \) must appear either as an input variable in the head or as an output variable in some preceding condition. These variables are all different by definition, thus using a different variable in this case would result in a different rule contradicting the initial assumption. Suppose one of the elements \( X \) in \( o \) differs from the corresponding element \( X' \) in \( o' \). In this case since all the output variables are different, the two rules would only different for a variable renaming and thus they would be equivalent contradicting the initial assumption. The case of two different MBL for the head can be similarly ruled out.

**Definition 4.3** Let \( M \) be a set of mode declarations. A mode-based rule (MBR) is a list of MBL. We say an MBR \([f_0, f_1, ..., f_n]\) represents a rule \( h ← b_1, ..., b_n \) iff \( f_0 \) represents \( h \) and each \( f_j \) represents \( b_j, j = 1, ..., n \).

Moreover \( f_j = \langle m, c, i, o \rangle \) satisfies the following requirements:

1. \( i_0 \cap o_1, ..., \cap o_m = \emptyset \)

2. and for each \( k = 1, ..., n \), if \( v ∈ i_k \) then \( ∃ g < k \) such that \( v ∈ o_g \) or \( v ∈ i_0 \)

The requirements are merely a rephrasing of the compatibility conditions for mode declarations previously defined (Definition 3.7).

**Example 4.3** Consider the mode declarations \( M \) from Example 3.1 and the rule \( p(X) ← q(X, Y), q(Y, a) \).

\( f_0 = \langle m_1, [], [X], [] \rangle, f_1 = \langle m_3, [], [X], [Y] \rangle \) and \( f_2 = \langle m_2, [a], [Y], [] \rangle \) are MBLs that represent respectively the literals \( p(X), r(X, Y), q(Y, a) \). Thus the MBR \([f_0, f_1, f_2]\) represents the rule \( p(X) ← q(X, Y), q(Y, a) \).

We want now to prove that for each element in \( R^o_M \), the set of ordered rules compatible with \( M \), there is one and only one MBR. Thus we can refer to MBR as an alternative representation for logic rules within a language bias specified by a set of mode declarations.

**Theorem 4.2** Let \( M \) be a set of mode declarations and \( r \) an ordered rule compatible with \( M \). There exists one and only one MBR that represents \( r \).
**Proof** Given any ordered rule that is compatible with a set of mode declaration, we can see that there is always a MBR representation for it, since it is sufficient to construct each MBL by filling the lists with the appropriate elements. The definitions of well-formed MBL and MBR are such that the requirements for a compatible rule are translated into requirements for the structure of the MBR. We want to prove that there exists only one MBR representation. We do this by induction on the length of the rule. The proposition is true for the head, for Theorem 4.1. We assume the proposition true for \( n \) conditions. So given a rule \( h \leftarrow b_1, \ldots, b_n \), we have a unique MBR \([r_0, \ldots, r_n]\). We add a condition \( b_{n+1} \) derived from a mode declaration \( m \). Given Theorem 4.1 there is only one possible MBL \( r_{n+1} \) that represents \( b_{n+1} \). Thus, given that the proposition holds for \( n \) conditions there is a unique MBR for \( h \leftarrow b_1, \ldots, b_n, b_{n+1} \), given by \([r_0, \ldots, r_n, r_{n+1}]\).

We can define an isomorphism from the space of MBRs that represent compatible rules (namely, those that respect the conditions in Definition 4.3) to the space of ordered rules. For a given set of mode declarations \( M \), we call this function \( r_M(\cdot) \).

It is possible to simplify the notation for MBRs. The output list is superfluous: given an MBR \([f_0, f_1, \ldots, f_n]\), for each \( f_i \), the output list is given by a list of variables that do not appear before in the rule. If \( k \) is the sum of the number of input variables in the head and the number of output variables in all the MBL \( f_j \), \( 0 < j < i \) then we can use a list of variables \((X_{k+1}, \ldots, X_{k+\text{out}(i)})\) for each \( f_i \). Similarly, input variables are constrained to refer to output variables or input variables in the head, and can thus abstract from the particular name and only refer to the order of appearance of these. This is clarified by the following example:

**Example 4.4** Consider the following mode declarations

\[
M = \begin{cases} 
  m_1 : mode(h(\text{uncle} (+\text{person}, +\text{person}))) \\
  m_2 : mode(f(\text{father} (-\text{person}, +\text{person}))) \\
  m_3 : mode(b(\text{gender} (+\text{person}, \#mf))) \\
  m_4 : mode(s(\text{siblings} (+\text{person}, +\text{person}))) 
\end{cases}
\]

The rule

\[
\text{uncle}(X_1, X_2) \leftarrow \text{father}(X_3, X_2), \text{siblings}(X_3, X_1), \text{gender}(X_1, m)
\]

is represented as \( r_M([m_1, [], []], [m_2, [], [2]], [m_4, [], [3, 1]], [m_3, [m], [1]]) \). Each tuple in the list only contains the mode declaration, the constants and the indexes of the input variables. For example, 2 in the second MBL refers to the second input variable in the head, \( X_2 \). Note that the particular name of the variables is not relevant.
4.2.1 Mapping mode declarations into an abductive top theory

Having defined a way to represent rules as structures that refer to the mode declarations and that can be codified as logic atoms (rather than rules) we can translate an ILP task into an equivalent ALP task. In order to do this we define a theory transformation and then prove that the semantics coincide for the two tasks.

The transformation relies on the explicit representation of mode declarations as “meta-rules” that structure the rule space. This representation resembles the one proposed in [Muggleton et al., 2008], where the concept of top theory is introduced. In our case we employ abductive reasoning to derive the final rules, instead of extracting them from an SLD derivation. We defer a deeper comparison to Section 4.4.

Definition 4.4 Given a set $M$ of mode declarations, the top theory $\top_M$ is constructed as follows:

- For each head declaration $m : \text{modeh}(s)$, the following rule is in $\top_M$

  $s^* \leftarrow$
  \[\text{body}(\text{inp}(s^*), [(m, \text{con}(s^*), [])])\]

- For each body declaration $m : \text{modeb}(s)$, the following clause is in $\top_M$

  $\text{body}(I, R) \leftarrow$
  \[\text{link}(\text{inp}(s^*), I, \text{Links}), s^*, \text{append}(R, (m, \text{con}(s^*), \text{Links}), \text{NR}), \text{append}(I, \text{out}(s^*), O), \text{body}(O, \text{NR})\]

- The following rules are in $\top_M$ together with a standard definition for the append predicate.

  $\text{body}(I, \text{NR}) \leftarrow$
  \[r(\text{NR})\]

  \[\text{link}([HL1|TL1], L2, [X|TV]) \leftarrow\]
  \[n\text{th1}(X, HL1, L2),\]
  \[\text{link}(TL1, L2, TV)\]

  \[\text{link}([], L2, []).\]
Rules (4.1) have a variabilised schema in the head. In the body they contain a condition that relates the truth of the head to body atoms. This condition links the relevant variables to the rest of the body and contains the MBL of the head. The rule can be declaratively read as “$s^*$ is true if it is in the head of a rule of the hypothesis and the body of this rule succeeds”. In the same way rules (4.2) contain a body condition that contains the MBL for the condition $s^*$ used in the same rule. The link predicate has as inputs a list of variables, namely the rule variables collected so far, and the variables that replace input placemarkers in the condition and produces as output a list of indexes, based on the unification of the input variables with variables in the rule variables. The rule (4.3) contains an abducible in the body that intuitively asserts the truth of the MBR contained in the argument, that is constructed by appending the MBL that appear in rules (4.2). The predicates append and nth1 and the list representation have their standard Prolog definition (e.g. the definition in [Costa et al., 2011]). The first is such that the third argument is a list obtained by concatenating the lists in the first two arguments. The latter has in the first argument the index of the second argument in the list given as third argument (e.g. nth1(3, 1, [8, 12, 1, 4])). Control over the types is performed by adding as conditions the set of literals \( \text{type}(s^*, s) \) to rules (4.1) and (4.2).

We first show the generation of the top theory through an examples and then discuss its use within the learning process in Section 4.3. More examples of the transformation can be found in Chapter 5.

**Example 4.5** Consider the following set of mode declarations $M$:

\[
M = \begin{cases} 
    m1 : \text{modeh}(p(+\text{any})) \\
    m2 : \text{modeb}(q(+\text{any}, \#\text{any})) \\
    m3 : \text{modeb}(q(+\text{any}, -\text{any})) 
\end{cases} \tag{4.5}
\]

The corresponding top theory $\top_M$ contains, together with rules (4.3) and (4.4) the following rules:

\[
p(V1) \leftarrow \\
\text{body}(V1, [(m1, [], [])]) \tag{4.6}
\]

\[
\text{body}(I, L) \leftarrow \\
\text{link}([V1], I, \text{Links}), \\
q(V1, C1), \\
\text{append}(L, (m2, [C1], \text{Links}, M), \\
\text{append}(I, [], O), \\
\text{body}(O, M) \tag{4.7}
\]

65
\[ p(V1) \text{ is the variabilisation of } p(\text{+any}), \quad q(V1, C1) \text{ is the variabilisation of } q(\text{+any}, \#\text{any}) \text{ and } q(V1, V2) \text{ is the variabilisation of } q(\text{+any}, -\text{any}). \] The first rule is derived from the head mode declaration \( m_1 \), the second and the third respectively from the body mode declarations \( m_2 \) and \( m_3 \). This is a purely syntactical transformation that involves the instantiation to the particular mode declaration of the general transformation in Definition 4.4.

The intuition behind this transformation may be not immediately clear, but will be evident after a formal justification is provided in Section 4.3. The head mode declarations are transformed into rules that are satisfied if an “artificial” body predicate is true. This predicate is used to instantiate the reasoning on the structure of rules that ultimately produce inductive hypotheses. Rule (4.3) provides a base case while the other rules defining body state that the conditions contained in the respective mode declarations can be used to extend a rule, for an appropriate instantiation of the variables.

### 4.3 Learning setting

Having defined how to construct an abductive top theory from a set of mode declarations, we discuss how this theory relates to the original ILP task \( \langle E, B, M \rangle \). The function \( ht_M(\Delta) = \{ c : \exists r(l) \in \Delta, c = rt_M(l) \} \) is used to extract the solution associated with an abductive hypothesis \( \Delta \). The following theorem is the main equivalence result for the abductive transformation.

**Theorem 4.3** Let \( \langle E, B, M \rangle \) be an ILP task, \( T_M \) the top theory derived from \( M \), \( H \in \mathcal{R}_M \) a normal program, \( \Delta \) a set of rule atoms such that \( H = ht_M(\Delta) \). Then \( B \cup T_M \cup \Delta \equiv B_{B \cup H} \cup B \cup H \).

**Proof** We use unfolding transformations to prove the equivalence. The initial theory of the unfolding transformation sequence is \( P_0 = B \cup T \cup \Delta \). Consider rules (4.1) of the top theory. There is one for each head mode declaration \( m \in M_h \). Let us pick for simplicity one of these rules compatible with a certain \( m_0 : modeh(s) \in M_h \). Applying unfolding to this rule on the atom \( body(inp(s^*)), (m_0, con(s^*), []) \), we need
to consider all the rules that define $body/2$ in $P_0$. By doing this, we obtain a new theory $P_1$ in which the rule we considered is replaced by the following rules, one for each body mode declaration $modeb(s_1) \in M_b$:

$$s_0^* \leftarrow$$
$$link(inp(s_1^*), inp(s_0^*), Links_1),$$
$$s_1^*,$$
$$append([m_0, con(s_0^*), []], [m_1, con(s_1^*), Links_1], NR),$$
$$append(inp(s_0^*), out(s_1^*), O),$$
$$body(O, NR)$$

(4.9)

We must also consider rule (4.3), since it defines $body$. Thus also the following rule is in $P_1$.

$$s_0^* \leftarrow$$
$$r([m_0, con(s_0^*), []])$$

(4.10)

Resulting from a single application of unfolding we have now $|M_b|+1$ rules in place of the initial one. Similarly we can apply unfolding on atoms of the type $body(O, NR)$ in the $|M_b|$ rules (4.9). Again, there are $|M_b|+1$ rules in $P_1$ that define the predicate $body/2$ and that unify with the atom. This process can be repeated $n$ times. The result after $n$ steps is a theory $P_n$ that contains the following $\sum_{i=1,...,n-1} |M_b|^i$ rules (i.e. rule (4.10), plus the $|M_b|$ rules obtained from rules (4.9) and (4.3) and so on), for $i = 1, ..., n-1$ and for all $m \in M_b$:

$$s_0^* \leftarrow$$
$$link(inp(s_1^*), inp(s_0^*), Links_1),$$
$$s_1^*,$$
$$append([m_0, con(s_0^*), []], [m_1, con(s_1^*), Links_1], NR),$$
$$append(inp(s_0^*), out(s_1^*), O),$$
$$...$$

(4.11)

$$link(I_i, O_{i-1}, Links_i),$$
$$s_i^*,$$
$$append(NR_{i-1}, [m_n, con(s_i^*), Links_i], NR),$$
$$append(O_{i-1}, out(s_n^*), O_i),$$
$$r(NR)$$
The theory $P_n$ also contains the following $|Mb|^n$ rules:

$$s^*_0 \leftarrow$$

\begin{align*}
\text{link}(\text{inp}(s^*_1), \text{inp}(s^*_0), \text{Links}_1), \\
s^*_1, \\
\text{append}([m_0, \text{con}(s^*_0)], [[], [m_1, \text{con}(s^*_1), \text{Links}_1], \text{NR}_1), \\
\text{append}(\text{inp}(s^*_0), \text{out}(s^*_1), \text{O}_1), \\
\vdots \\
\text{link}(I_n, \text{O}_{n-1}, \text{Links}_n), \\
s^*_n, \\
\text{append}(\text{NR}_{n-1}, [m_n, \text{con}(s^*_n), \text{Links}_n], \text{NR}_n), \\
\text{append}(\text{O}_{n-1}, \text{out}(s^*_n), \text{O}_n), \\
\text{body}(O_n, \text{NR})
\end{align*}

\hspace{1cm} (4.12)

Note that each $m_j, j = 1, \ldots, i$ in formulae 4.11 and $m_j, j = 1, \ldots, n$ in formulae 4.12 is such that $m_j \in Mb$. Also, it is possible that $m_q = m_p$ for $p, q > 0$, since same rule in the top theory can be used more than once in the transformation sequence. It must be pointed out that we never applied unfolding recursively on the same rule, thus respecting the condition in Theorem 2.3 for semantics equivalence.

Let us consider $\Delta$. It contains atoms defining $r/1$ and let us assume the length of the list (the transformed rule) that is the argument of $r$ is at most $n - 1$. Only rules (4.11) in $P_n$ contain $r/1$ atoms in the body. We can apply unfolding again and delete all the rules that do not contain a condition that can be unified with one of the atoms in $\Delta$. This preserves the semantics since the body of these rules is always false. Considering the definitions of $\text{append/3}$ and $\text{link/3}$, the remaining rules (4.11), again by applying unfolding to all the conditions except $s^*_1, \ldots, s^*_i$, are equivalent to $H$. In fact, it is easy to observe that the literals that are not derived from mode declarations have the effect, in the transformed rules, of appending the lists and linking the variables coherently with the given definition of MBR. We call $\top'$ the theory obtained from $\top$ by deleting rules (4.1). Such rules have been transformed by the repeated applications of the unfolding transformation. At this point we have a theory $P'$ that contains $H$, $B$, $\top'$, $\Delta$ and rules (4.12) by applying semantics-preserving transformations. $\top'$ and $\Delta$ define a set of predicates disjoint from $H$ and $B$ (we assume that the special predicates $r/1$, $\text{link/3}$, $\text{append/3}$ and $\text{body/2}$ never appear in $B$ or in the mode declarations and thus in any possible $H$). The only rules in $P'$ that still use special predicates to define elements of the inductive problem are rules (4.12). These rules contain a $\text{body/2}$ atom in the body. But these rules will never have a true body since $\Delta$ contains abducibles $r/1$ whose argument is a list of at most $n - 1$ elements. The second argument of the $\text{body/2}$ atom in rules (4.12) is a list of length $n$ and the definition of $\text{body/2}$ in $\top$ is such that this length can...
only increase by repeated applications of unfolding (there is no atom that decreases the length of the list in the transformed rule). Thus we can safely discard rules (4.12) as the body conditions are not true and can never be true in any of the rules obtained by repeated applications of unfolding. The new theory $P'' = H \cup B \cup T' \cup \Delta$ is semantically equivalent to $H \cup B \cup T \cup \Delta$, given that it is obtained by an unfolding transformation sequence, and the definitions contained in $T' \cup \Delta$ are disjoint from those in $H \cup B$ and are not used in any body of the rules in $H \cup B$. Thus from each model of $H \cup B \cup T' \cup \Delta$ we can isolate a subset of the model that only contains elements from the set $U_{B \cup H}$ (the Herbrand universe of $B \cup H$). It is clear that these subsets are only determined by the rules in $H \cup B$. Thus, for each model of the original theory, $P_0$ is equal to exactly one model of $H \cup B$ that is a subset of a model of $H \cup B \cup T' \cup \Delta$, thus proving the theorem.

**Theorem 4.4** Given the ILP task $\langle B, E, R, M \rangle$, $H$ is a hypothesis if and only if $\Delta$ is an abductive solution for the ALP task $\langle B \cup T, \{ r/1 \}, \emptyset \rangle$ and $H = ht_M(\Delta)$.

**Proof** $H = ht_M(\Delta)$ is a hypothesis iff $B \cup H \models E$. Given Theorem 4.3 $B \cup H \models E$ iff $B \cup T \cup \Delta \models E$ since $E \in U_{B \cup H}$. Thus $\Delta$ is a solution for $\langle B \cup T, \{ r/1 \}, \emptyset \rangle$.

To establish the soundness and completeness properties of an ILP system based on the transformation we can use Theorem 4.4 together with a result on soundness and completeness of the abductive system used. If the abductive system is able to find any possible abductive solution $\Delta$ and these solutions are correct then ultimately we can derive all and only the possible inductive solutions for a certain problem. We will discuss in the next chapters the constraints that need to be considered on completeness for concrete implementations.

### 4.4 Related work

The integration of abductive reasoning into ILP has been studied in several works. The main difference with all these studies is that what we have presented here is a general mapping technique rather than an ad hoc implementation that employs abductive reasoning as part of it.

ACL (Abductive Concept Learning) [Kakas & Riguzzi, 2000] is the most comprehensive system and extends the usual setting with the learning of integrity constraints. It also replaces the usual deduction based coverage with abductive-based coverage, in order to support problems with incomplete information and the use of integrity constraints better. Instead of requiring a solution that logically entails the given examples, ACL derives solutions such that the examples can be explained abductively, given a set of additional abducibles. For positive examples it is required that there exists an explanation, while for the negative examples it must be impossible to find a subset of the abducibles that is an abductive explanation. The system is structured in two phases
ACL1, that deals with the learning of rules, and ACL2 that learns integrity constraints. In this work it is also shown how abductive reasoning enables multi-predicate learning. As recognised by the authors the problem is inherently nonmonotonic thus a standard covering loop approach cannot be used on the unconstrained input language. This is addressed by imposing a limitation on the background knowledge and the language of the hypothesis: the background knowledge cannot depend on target predicates. This constraint results in a limited setting with respect to the one considered here which instead allows for NOL and recursion through predicates in the background knowledge as shown more in detail in Chapter 5. ACL operates under the conditions in Theorem 3.2 that simplify the learning task and enable the use of a sequential covering approach. ACL cannot be applied without a loss of completeness outside these conditions.

MPL-A (Multiple Predicate Learning by Abduction), extends ACL in order to learn definite programs that define multiple interdependent predicates. The algorithm generates abductive hypotheses about predicates with an incomplete definition in order to ensure that negative examples are not covered.

A mechanism that is similar to the one used in MPL-A emerges from the implementation of the abductive transformation presented in Chapter 5. Also in that case, the abductive procedure produces partial goals that represent “artificial” examples generated from a partial definition. This is not an artificial mechanism added to support multiple-predicate learning but an indirect result of the transformation presented here.

As previously described, XHAIL [Ray, 2009b] (eXtended Hybrid Abductive Inductive Learning), uses abduction in its three phases. In the first phase abduction is used in order to derive atoms that are generalised into heads of the kernel set. This is the phase of the algorithm where abduction is used in its full setting, while in the second phase abduction is, in practice, used for a deductive task and in the third phase abduction performs a generalisation search. A deeper comparison with XHAIL is deferred to Chapter 5.

LAP (Learning Abductive Programs) [Lamma et al., 1997] targets the learning of abductive logic programs, a different problem from the one tackled here, that uses an abductive logic program to learn a normal logic program. The system employs a covering loop approach that generates rules sequentially. Interestingly, similarly to ACL, the example coverage is tested abductively, thus an abductive solution is generated together with the rule. The positive literals of target predicates in the abductive solutions are added to the positive examples and the atoms corresponding to negative literals are added to negative examples. This closely resembles the implicit generation of “artificial examples” discussed in Chapter 5.

AILP (Abductive Inductive Logic Programming) [Adé & Denecker, 1995] extends the abductive proof procedure SLDNFA in order to enable inductive reasoning. The parallel with the work presented here is particularly interesting since SLDNFA is used in Chapter 5 to instantiate the transformation and supports normal clauses. The approach is based on the extension of the specific proof procedure rather than on a general mapping. Moreover
AILP is a generic framework and crucially it does not define how the refinement of a given hypothesis operates, but is concentrated on what literals are used as examples, these being derived during the proof procedure, and when a refinement is triggered. One of the goals of this thesis is to address this gap.

4.5 Conclusion

We have presented a general methodology to map an ILP problem into an equivalent ALP problem such that any solution of the former corresponds to a solution of the latter and any solution of the latter corresponds to a solution of the former, thus providing a sound and complete transformation.

The transformation relies on the given mode declarations. They are used to produce rules in a so called top theory that encodes the rule space. The process of finding a solution becomes a “meta-abductive” reasoning process, in the sense that the reasoning is not only on elements of the domain, but on the structure of the rules that are a solution for the given ILP problem.

The transformation can be used in principle by any ALP system, and depending on the choice it is possible to report the soundness and completeness properties of the learning in terms of the soundness and completeness of the underlying inference system used, as shown in the next chapters. Despite the many cases in the literature of integration between abductive and inductive logic programming, this is the first approach that consists of a full problem reduction.
5 Top-directed Abductive Learning

In Chapter 4 we have introduced a transformation that enables the use of ALP systems to find inductive hypotheses. Here we investigate the design of an ILP system, called TAL, that builds on that transformation. We describe the abductive proof procedure used and how learning under nonmonotonic inference takes place. We also discuss how TAL can be instantiated to the case of monotonic ILP problems by applying constraints on the inputs and redefining the search strategy. Like in [Muggleton et al., 2008] the computation is driven by the given examples and the proof is used to construct the final hypotheses, but in TAL this is done by employing a different transformation of the given inductive task, i.e. the one described in Chapter 4, and by using ALP to compute the hypothesis from the examples. We show how the process resulting from the transformation and abductive reasoning enables learning under negation as failure and recursive and multi-predicate learning. The system also guarantees completeness, under the restrictions of the underlying abductive system. Also, unlike in [Muggleton et al., 2008] and in most ILP systems, positive and negative examples are treated symmetrically.

The search space depends on all the examples and considers refinements of full hypotheses and not single rules.

In the particular case of definite theories, TAL is able to compute hypotheses that are not found by established inverse entailment-based systems like PROGOL [Muggleton, 1995] or ALECTO [Moyle, 2002] and provides a more effective solution for the task of learning interdependent concepts compared to the state-of-the-art ILP systems, e.g. [Kimber et al., 2009, Ray, 2009b]. In particular we show through an example how TAL is less affected by non-determinism in the search than these systems.

We first discuss the abductive proof procedure used in TAL then we introduce some necessary refinements towards a practically viable ILP system. Then we analyse efficiency issues and how the search space defined by the abductive transformation can be effectively tailored for a specific inductive learning task, e.g. single clause monotonic ILP tasks. Finally we provide an empirical comparison with some other ILP systems.

5.1 Top-directed abductive learning

The basic mode of execution of TAL (Top-directed Abductive Learning) consists of executing an abductive proof procedure with the theory produced by the transformation introduced in Chapter 4. Algorithm 2 shows
the main computational steps. First the transformation executed by the procedure PRE-PROCESSING generates the top theory. The abductive proof procedure, which is non-deterministic and thus can generate more than one solution, derives a solution $\Delta$ that is then translated back into an inductive solution $H$ by the procedure POST-PROCESSING that implements the transformation $rt_M(\cdot)$.

### Algorithm 2 TAL

**Inputs:** $E$ examples; $B$ background theory; $M$ mode declarations; $I$ integrity constraints  
**Outputs:** $H$ hypothesis  

\[
\begin{align*}
\top_M &= \text{PRE-PROCESSING}(E, B, M) \\
\Delta &= \text{ABDUCE}(\top_M \cup B, I, E) \\
H &= \text{POST-PROCESSING}(\Delta, M)
\end{align*}
\]

Note that this basic version only generates candidate hypotheses and does not consider any preference between them. We address this problem later in the chapter. The inputs for the algorithm also include a set of integrity constraints. These can be used to add additional bias to the search and are supported by the abductive proof procedure. For example integrity constraints can be used to avoid two mode declarations being used within the same rule\(^1\).

#### 5.1.1 The even-odd problem

In this section we discuss an example of learning the definition of even and odd numbers. A different formulation of this problem was first used in [Yamamoto et al., 2010] to show that the system PROGOL is not complete. We present a new version of the problem that shows the additional difficulty of learning non-observed predicates and involves multi-predicate learning.

\[
\begin{align*}
B &= \{ \text{even}(0) \} \\
E &= \{ \text{odd}(s(s(0))), \text{not odd}(s(s(0))) \} \\
M &= \{ \text{eh} : \text{modeh}(\text{even}(\text{+nat})), \text{oh} : \text{modeh}(\text{odd}(\text{+nat})), \text{bn} : \text{modeb}(\text{not odd}(\text{+nat})), \text{be} : \text{modeb}(\text{even}(\text{+nat})), \text{bs} : \text{modeb}(\text{+nat} = s(\text{-nat})) \}
\end{align*}
\]

Following the transformation described in Chapter 4 we obtain the following abductive top theory:

---

\(^1\)Integrity constraints can be used as goals to further refine the intended semantics of the learning process since, as shown in Definition 2.3, they are required to be consequences of the background theory and the abductive solution. In practice, the abductive procedure employed may result in poor performance when the integrity constraints are not ground and do not involve abducibles. In this case the ASPAL system, discussed in Chapter 6 would provide a better performance.
An execution of Algorithm 2 returns as a solution for the ALP task $\langle E, B \cup T_M, \{r\}, \emptyset \rangle$:

$$\Delta_1 = \left\{ r([oh, [], []], (bs, [], [1]), (bn, [], [2])) \right\}$$

which corresponds to the rule $\text{odd}(X) \leftarrow X = s(Y), \text{not odd}(Y)$ as computed by the POST-PROCESSING procedure. The rule can be obtained using the MBR encoded in the list $[(oh, [], []), (bs, [], [1]), (bn, [], [2])]$. The first element refers to the head, and uses the mode declaration $oh$. Since no constants appear the head atom encoded is $\text{odd}(X)$. Similarly the second element of the list uses the mode declaration $bs$, so the first body literal in this rule is an equality between variables, as encoded in mode declaration $bs$. In this case the third argument of the MBL specifies that the input variable in this equality body literal is linked to the first non-input variable appearing in the rule, in this case $X$. The corresponding equality literal is thus $X = s(Y)$.

The variable $Y$, as previously states in Section 4.2 must not replace any other output placemarker in the rule. The last element of the list makes use of the mode declaration $bn$. Here the input variable is instantiated as the second non-input variable appearing in the rule, which is $Y$. Another subset-minimal abductive solution is:

$$\Delta_2 = \left\{ r([oh, [], []], (bs, [], [1]), (be, [], [2])) \\
r([eh, [], []], (bn, [], [1])) \right\}$$

This solution corresponds to the rules $\text{odd}(X) \leftarrow X = s(Y), \text{even}(Y)$ and $\text{even}(X) \leftarrow \text{not odd}(X)$. Since two atoms are part of the abductive solution, the hypothesis contains two rules. The abductive problem, as previously defined, includes an infinite number of solutions, but the actual top theory uses a set of additional control predicates that limits the maximum depth of the rules and check that conditions are not repeated in the rules$^2$.

$^2$Consider one of the rules in $T_M$. Given a maximum depth $d$ for the rules in the rule space, it is sufficient to add the conditions
Assuming that a sound and complete ALP system is used to derive all the abductive explanations, according to Theorem 4.4, all the possible solutions are derived (there are 5 solutions for this problem, if limited to at most two rules and two conditions). We further analyse this example with respect to other ILP systems later in Section 5.5.

5.2 SLDNFA and extensions for ILP

TAL is based on the SLDNFA [Denecker & Schreye, 1998] abductive system. Notably, SLDNFA is able to handle a large class of abductive problems, including cases that cause abductive floundering in other proof procedures [Mancarella et al., 2002]. Furthermore, SLDNFA supports non-ground abducibles and thus it provides more freedom in the actual implementation of TAL, allowing more flexibility on the evaluation order of the goals and the order of selection of the literals. Also, since SLDNFA is an abductive extension of the SLDF procedure, it represents the most natural extension of TOLOG [Muggleton et al., 2008], where hypotheses are constructed by inspecting the SLD refutations. More specifically TAL builds on a development of SLDNFA called A-SYSTEM. The main difference consists in the incorporation of a constraint solver that handles constraints over real numbers and finite domains.

5.2.1 A-SYSTEM

We provide here a description of the main parts of the proof procedure. Note that this section does not contain a contribution of this thesis but provides background on the abductive system A-SYSTEM used in TAL. We do not go into the details of the proof procedure that are given in [Denecker & Schreye, 1998] and [Van Nuffelen, 2004]. Instead we focus on the most relevant features from our ILP perspective. The input for A-SYSTEM is an ALP task $\langle q_0, P, A, I \rangle$. The procedure can be seen as derivation of formulae of the type $\forall q_0 \leftarrow \Phi$, where $q_0$ is the initial (abductive) query. $\Phi$ includes a conjunction of goal formulae, that can be either positive or negative. A positive goal, represents a conjunction of literals $l_1, \ldots, l_n$ where variables are open. Negative goals represent formulae of the type $\forall X. \leftarrow l_1, \ldots, l_n$ where $X$ contains a subset of the variables appearing in $l_1, \ldots, l_n$. During the derivation negative literals are deleted from positive goals and added to negative goals and vice versa, similarly to SLDNF. The use of explicit quantifiers in the negative goals is required to deal with the abduction of non-ground atoms [Denecker & Schreye, 1998]. In fact goals can contain

\[ \text{length}(\text{NRule}, L), L \leq d + 1, \text{where length has the standard Prolog definition in which the second argument is the length of the list in the first argument.} \]

\[ ^3 \text{In particular we employed and extended a further refinement of the A-SYSTEM proof procedure [Ma, ], that was used to implement the distributed abductive system used in [Ma et al., 2010]. This further refinement of the original SLDNFA proof procedure provides a more modular Prolog implementation that facilitates ad hoc modifications of the ILP algorithm, by making each rewriting step more explicit and easily extensible. We will still refer to this implementation of the proof procedure as A-SYSTEM for simplicity.} \]
two type of variables: free variables, that are quantified universally by the quantifier at the beginning of the formula $\forall q_0 \leftarrow \Phi$ (also called positive variables); and variables that are quantified in one of the conjuncts of $\Phi$ (also called negative variables).

In order to simplify the exposition, using a convention introduced in [Van Nuffelen, 2004], we can represent each state of the derivation $\forall q_0 \leftarrow \Phi$ as a tuple $S = \langle G, \Delta, \Delta^*, C \rangle$ where

- $G$ is a set of (positive and negative) goal formulae
- $\Delta$ is a set of atoms $a(\overline{t})$
- $\Delta^*$ is a set of integrity constraints containing an abducible atom $\leftarrow a(\overline{t}), B$
- $C$ is a repository of finite domain constraints and inequalities

In the tuple, $\Delta$ becomes a way to capture abducibles left as residuals from positive goals. The meaning $\Phi$ of the state $\forall q_0 \leftarrow \Phi$ is given by the conjunction of the formulae in all the elements of the state tuple. The application of inference rules may be non-deterministic, thus we define a derivation tree in which every node is a state. A derivation starts with the given query (i.e. with $\Phi = q_0 = G$) and it terminates when $G$, the first element of the abductive state, is empty, i.e. a solution or success state is reached. More specifically, when integrity constraints $I$ are provided, the root is the initial state $S_0 = \langle q_0 \cup I_g, \emptyset, I_a, \emptyset \rangle$, where $I$ is partitioned into integrity constraints that contain abducibles $I_a$ and integrity constraints that do not $I_g$. All the children of a node are the states that can be derived from the application of inference rules to that node. A leaf can be either failed, due to inconsistencies, or successful, when a solution state with empty goal is reached. A derivation is called finite if all the branches are finite. A solution for the derivation is given by the set $\Delta$ in a successful state.

We report the main inference rules in Table 5.1, detailing only the elements of the state that change. Note that for each element in the goal set, it is essential to define which variables are universally quantified. For example when a negative literal is selected in a positive goal, the proof makes sure that the new negative goal derived from the literal fails for all possible instantiations of the variable. Let $S_i = \langle G_i, \Delta_i, \Delta_i^*, C_i \rangle$ be an abductive state, $F$ a selected goal from $G_i$ and $G_i^− = G_i \setminus \{F\}$. The state $S_{i+1} = \langle G_{i+1}, \Delta_{i+1}, \Delta_{i+1}^*, C_{i+1} \rangle$ is a child of $S_i$ if obtained by applying one of the inference rules in Table 5.1. The rule UNF-1 applies unfolding on one of the literals of the goal, similarly to SLD. The rule UNF-2 operates on a negative goal. For all rules whose head unifies with the selected literal $p(\overline{t})$ a new goal is generated and added to the goal set where all the variables are universally quantified. All the goals generated must eventually fail. NEG-1 and NEG-2 deal with negated literals in a goal and generate negative goals from negated literals in positive goals and vice versa positive goals from negated literals in negative goals. The rule ABD-1 is the only one that alters the current abductive solution.
Table 5.1: Inference operators in `A-SYSTEM`. “OR” in the third column separates nondeterministic choices. In order to facilitate the reading, $F$ is represented as a singleton set. The predicate $a$ is used for abducible atoms, while $p$ is used for non-abducible atoms.
by adding elements to $\Delta$. Note that in this case if the abducible unifies with an atom in the constraints $\Delta^*$, new negative goals are added to make sure these constraints are respected.

Example 5.1 Consider the abductive task $\langle B, \{r\}, \emptyset \rangle$ and the query not $\text{penguin}(a)$ where:

$$B = \begin{cases} 
  b_1 : \text{can}(a, fly), \\
  b_2 : \text{can}(b, swim), \\
  b_3 : \text{ability}(fly), \\
  b_4 : \text{ability}(swim), \\
  b_5 : \text{penguin}(X) \leftarrow \text{not superpenguin}(X), \\
  \top_1 : \text{superpenguin}(X) \leftarrow \text{body}(X, [\text{head}]), \\
  \top_2 : \text{body}(X, L) \leftarrow r(L) \\
  \top_3 : \text{body}(X, L) \leftarrow \text{can}(X, C), \text{ability}(C), \text{append}(L, [(\text{cond}, C)], L1), \text{body}(X, L1) \\
  a : \text{append}(A, B, C) \leftarrow ... \end{cases}$$

(5.1)

The example encodes a simplified abductive translation of the ILP task $\langle \{\text{not penguin}(a)\}, \emptyset, \emptyset, \emptyset \rangle$. In the following we omit the quantifier for the negative goals, considering them universally quantified within the goal.

The initial state is:

$$S_0 = \langle \{\text{not penguin}(a)\}, \emptyset, \emptyset, \emptyset \rangle$$

The only applicable inference rule is NEG-1, that transforms the state by changing the positive goal into a negative goal:

$$S_1 = \langle \langle \text{penguin}(a) \rangle, \emptyset, \emptyset, \emptyset \rangle$$

The UNF-2 rule is applied since penguin is not an abducible, obtaining:

$$S_2 = \langle \{\forall X \leftarrow a = X, \text{not superpenguin}(X)\}, \emptyset, \emptyset, \emptyset \rangle$$

The EQ-2 rule is applied. Since the rule is nondeterministic, two children are produced. We focus on the following child (ignoring the other, where the inequality $X \neq a$ is added to $C$):

$$S_3 = \langle \langle \text{not superpenguin}(a) \rangle, \emptyset, \emptyset, \emptyset \rangle$$

The NEG-2 rule is applied. In this case two nodes are generated. In one state the proof contains a positive goal superpenguin(a) (we skip the other branch of the tree as it is bound to fail eventually):

$$S_4 = \langle \{\text{superpenguin}(a)\}, \emptyset, \emptyset, \emptyset \rangle$$

UNF-1 is applied, using rule $\top_1$ (we also skip the application of rule EQ-1 for brevity):
we apply UNF-1 again, but this time two rules can be used, creating two children nodes. We explore the branch resulting from using $T_2$ and explore the other child node $S_8$ later:

$$S_6 = \langle \{ r([head]) \}, \emptyset, \emptyset, \emptyset \rangle$$

The only literal in the goal now is an abducible, so ABD-1 applies to it, producing an empty goal set, thus marked as a solution state:

$$S_7 = \langle \emptyset, \{ r([head]) \}, \emptyset, \emptyset \rangle$$

As a result of the application of UNF-1 on $S_5$ using rule $T_3$ we obtain:

$$S_8 = \langle \{ can(a, C), ability(C), append(L, [(cond, C)], L1), body(a, L1) \}, \emptyset, \emptyset, \emptyset \rangle$$

We now skip some straightforward applications of the UNF-1 rule on the literals in the only goal. Eventually the following state is obtained:

$$S_9 = \langle \{ body(a, [head]), \emptyset, \emptyset, \emptyset \} \rangle$$

Like in $S_5$ we can apply UNF-1 on two rules. We explore the branch created from using $T_2$:

$$S_{10} = \langle \{ r([head, (cond, fly)]) \}, \emptyset, \emptyset, \emptyset \rangle$$

We can apply ABD-1 that results again in a solution state:

$$S_{11} = \langle \emptyset, \{ r([head, (cond, fly)]) \}, \emptyset, \emptyset \rangle$$

The procedure continues exploring the other child of state $S_9$. We can observe that by repeated application of unfolding using rule $T_3$ the proof ends up in an infinite branch. This type of recursion is handled in TAL by imposing a maximum length on the list argument in the body atom and also by a check on the repetition of conditions in the rule. From the two solution states we can extract two solutions. $S_7$ contains the abductive solution $r([head])$. In this simplified example we assume head corresponds to a mode declaration, similarly to the transformation introduced in Chapter 4. Thus $r([head])$ corresponds to the solution

$$rt_M([head]) = superpenguin(X)$$

Similarly $S_{11}$ contains the abductive solution $r([head, (cond, fly)])$, which corresponds to the solution

$$rt_M([head, (cond, fly)]) = superpenguin(X) ← can(X, fly)$$

**Soundness and completeness**

The soundness and completeness results of the SLDNFA proof procedure are provided more extensively in [Denecker & Schreye, 1998] and [Van Nuffelen, 2004]. Since we relate the results to the 3-valued completion
semantics, in the remainder of this Chapter propositions of the type $P \models f$ are to be interpreted under 3-valued completion semantics (see Sections 2.4.1 and 2.6 and [Denecker & Schreye, 1998]).

**Theorem 5.1** Soundness [Van Nuffelen, 2004]. Let $(P, A, I)$ be an abductive program and $(\emptyset, \Delta, \Delta^*, C)$ be a solution state for the query $q$ with a substitution $\theta$. Then

$$P \cup \Delta \models q\theta$$

and $P \cup \Delta$ is consistent.

**Theorem 5.2** Completeness [Van Nuffelen, 2004]. Let $(P, A, I)$ be an abductive program and $W$ be a derivation tree for a given query $q$. If $W$ is finite then:

- if all the branches of $W$ are failed, then $P \models \neg\exists q$
- if $P \cup \exists q$ is satisfiable, then $W$ contains a successful branch.

In Theorem 5.2, the requirement that the derivation tree is finite can be considered a restrictive condition. In particular we are interested in a stronger result that states that for each solution of the abductive problem there is a corresponding branch in the derivation tree that encodes the solution. The class of programs that ensure termination for the abductive procedure is studied in [Verbaeten, 1999]. Termination is ensured for acyclic and abductive nonrecursive programs\(^4\). We refer to [Verbaeten, 1999] for a definition of abductive nonrecursive program. This property is query dependent and it prevents situations where a negative goal leads to the abduction of elements that in turn reactivate the negative goal on a different query. Assuming abductive nonrecursiveness and acyclicity, the completeness result remains weak as it still does not guarantee that all the solutions for the corresponding ILP problem are found. A stronger result is provided in [Denecker & Schreye, 1998] for a variation of SLDNFA called SLDNFA+. An SLDNFA+ refutation is equivalent to an SLDNFA refutation where in addition we require that all constraints in $\Delta^*$ are atomic irreducible equality goals. Whenever this extension is applied we can use the following stronger result on completeness:

**Theorem 5.3** (Simplified from [Denecker & Schreye, 1998]). For any abductive solution $\Delta$ for a given ALP task, each finite derivation tree contains a success leaf with abductive answer $\Delta'$ such that $\Delta' \subseteq \Delta$.

The theorem states that an abductive derivation tree always contains all the subset-minimal abductive hypotheses.

\(^4\)The queries considered here are always ground, thus bounded as required in [Verbaeten, 1999].
It is possible to prove that, for the particular structure of the top theory, \( TAL \) only derives ground abducibles and is such that \( SLDNFA^+ \) and the base \( SLDNFA \) coincide. We do not provide a full formal account of this property in order not to clutter the exposition with overly specific notions. Intuitively since abducibles contain lists that are formed using the \( append \) predicate in the body of the rules in the top theory, the positive goals that lead to the application of the rule \( ABD\)-1 always include the instantiation of the constants (if we assume that the type check always grounds the variable in it) and the instantiation of the third element in the MBL, which similarly always grounds the index in the variable linking. If the assumption that the type check for constant elements always grounds variables does not hold, it is possible to use a mode of execution in the underlying abductive system that implements the additional derivation steps in \( SLDNFA^+ \), thus conforming to the completeness result in Theorem 5.3.

5.3 Soundness and Completeness of \( TAL \)

Based on Theorems 4.4, 5.1, 5.2, and 5.3 we can state the following soundness and completeness result for \( TAL \):

**Theorem 5.4** Let \( \langle B, E, R_M \rangle \) be an ILP task, \( T_M \) be the top theory generated from \( M \) and \( T_M \cup B \) an acyclic program. If \( T_M \cup B \) is abductive nonrecursive for the query \( E \) then \( H = TAL(E, B, M, \emptyset) \) iff \( H \) is a subset-minimal solution for the ILP task \( \langle E, B, M \rangle \).

**Proof** The theorem follows from Theorems 5.3 and 4.3. The subset-minimality of the ILP hypothesis follows from the fact that each rule is represented by one abducible.

As a final remark, if two solutions \( H' \) and \( H \) exists for the inductive task such that \( H \subset H' \), \( H' \) is not guaranteed to be derived as it is not a subset-minimal solution. This does not penalise the system in practical applications (at least in all the cases where a more compressive solution is preferable) since the additional rules in \( H' \) can be safely removed without affecting the coverage on the examples.

5.4 Inductive extensions and \( TAL_H \)

Although the mechanism described so far is supported by a completeness result, in many cases this is not sufficient as efficiency should be taken into account.

Candidate solutions for the ILP task can be extracted from nodes in the proof and not just from leaves. For this purpose we have extended the mechanisms in two respects. First, the concept of partial rule is introduced,
modelled as a special atom for the procedure. Second, the abductive system is extended with additional support for search strategies based on the properties of the partial hypotheses.

### 5.4.1 Partial rules

We add a condition to the rules in the top theory that models the fact that in a successful branch generated from the current state a certain rule or a specialisation of the rule will be included. Rules (4.2) are extended as follows (the new condition is underlined):

\[\text{body}(I, R) \leftarrow \]
\[\text{link}(\text{inp}(s^*), I, \text{Links}),\]
\[s^*, \]
\[\text{append}(R, (m, \text{con}(s^*), \text{Links}), \text{NR}),\]
\[\text{pr}(	ext{NR}), \]
\[\text{append}(I, \text{out}(s^*), O),\]
\[\text{body}(O, \text{NR})\]  

(5.2)

The intuitive meaning of the condition is that if \(\text{pr}(\text{NR})\) is abduced for a certain MBR \(nr\) then the partial rule represented by \(nr\) will be part of a rule contained in an inductive solution. In order to define how the use of this new abducible affects the derivation, we consider the following result.

**Proposition 5.5** If \(r([a_1, \ldots, a_n])\) is in the abductive solution \(\Delta\) for \((B \cup T, \{r/1, \text{pr}/1\}, \emptyset)\) then also \(\text{pr}([a_1]), \text{pr}([a_1, a_2]), \ldots, \text{pr}([a_1, \ldots, a_n])\) are in \(\Delta\).

**Proof** Let us consider the only operator that adds elements to the abuctive solution in the proof procedure, \(\text{ABD-1}\). \(r([a_1, \ldots, a_n])\) is added to the current solution only if it is the selected literal in a positive goal. Considering the possible ways \(r([a_1, \ldots, a_n])\) can appear in the goal list, at some previous point \(\text{UNF-1}\) or \(\text{UNF-2}\) must have been used on \(\text{body}([a_1, \ldots, a_n])\). The only way the list can be constructed is by using the append atom in rules (5.2). But whenever a new element \(a_i\) of the list is added we know that also the corresponding literal \(\text{pr}([a_1, \ldots, a_i])\) appears in one of the goals. This suffices to prove the proposition since in every successful branch the goal set must be empty, thus at some point \(\text{ABD-1}\) must have been applied on \(\text{pr}([a_1, \ldots, a_i])\) and thus must have been added to \(\Delta\).

The proposition allows us to have more control over the construction of the hypothesis, allowing us to pursue various extensions as discussed later in Section 5.4.2. It states that the proof procedure monotonically adds conditions to rules being constructed, and these can be traced in the proof procedure by means of the current
abductive solution. In fact, given a $\Delta'$ in a state of the proof procedure, we can extract a set of rules $H'$ such that any success state of the procedure corresponds to an inductive solution $H$ where for each clause $r \in H'$, it holds that $r \in H$. Moreover, $pr/1$ is a control abducible with no direct effect on the solutions. Therefore, after the addition of the new conditions to rules (4.2), and considering the additional atoms in $\Delta$ it is easy to verify that Theorem 4.4 still holds and that $pr/1$ abducibles do not affect soundness and completeness of the system.

Furthermore, the use of partial rules allows for better control of the search space. When a partial rule abducible $pr(mbr)$ is encountered in a negative goal and it is not part of the current abductive solution, it is added to $\Delta^*$. This has the effect of a constraint over the construction of the hypothesis. The negative goal succeeds and thus the proof continues. If $pr(mbr)$ is selected in a positive goal then the proof reactivates the constraint, thus lazily checking the failure of the negative goal. We illustrate this through an example.

**Example 5.2** Consider $B$ in Example 3.3, used previously to illustrate the top theory produced by TOPLOG (it is interesting to compare the two). We recall the involved inputs:

\[
B = \begin{cases} 
  b_1 : \text{can}(a, \text{fly}), \\
  b_2 : \text{can}(b, \text{swim}).
\end{cases} \quad (5.3)
\]

\[
E = \begin{cases} 
  \text{not penguin}(a), \\
  \text{penguin}(b).
\end{cases} \quad (5.4)
\]

\[
M = \begin{cases} 
  m_1 : \text{modeh(penguin(\text{any}))}, \\
  m_2 : \text{modeb(can(\text{any, #ability}))}
\end{cases} \quad (5.5)
\]

*The top theory produced in TAL is:*
We report here some crucial states in the abductive proof. The initial state is:

\[ S_0 = \langle \{ \text{not penguin}(a), \text{penguin}(b) \}, \emptyset, \emptyset, \emptyset \rangle \]

The selected first negative example is transformed into a negative goal:

\[ S_1 = \langle \{ \leftarrow \text{penguin}(a) \} \cup \{ \text{penguin}(b) \}, \emptyset, \emptyset, \emptyset \rangle \]

The literal in the negative goal unifies with the head of \( T_1 \). Applying UNF-2 we obtain the goal

\[ \{ \leftarrow \text{body}([a], [(m1, [], [])]) \} \] that in turns unify with the head of \( T_2 \) and \( T_3 \) resulting in the following state:

\[ S_2 = \langle \{ \leftarrow \text{link}([X], \text{Inputs}, \text{Links}), \ldots \} \cup \{ \leftarrow r([(m1, [], [])]) \} \cup \{ \text{penguin}(b) \}, \emptyset, \emptyset, \emptyset \rangle. \]

Note that two negative goals are generated. The procedure continues by unfolding the elements of the body of \( T_2 \). Eventually the proof procedure reaches the following state:

\[ S_3 = \langle \{ \leftarrow \text{pr}([(m1, [], []), (m2, [fly], [1])]), \text{body}(a, [(m1, [], []), (m2, [fly], [1])]) \} \cup \{ \leftarrow r([(m1, [], [])]) \} \cup \{ \text{penguin}(b) \}, \emptyset, \emptyset, \emptyset \rangle. \]

Without the use of the partial rule abducible, the proof would continue by applying UNF-2 to \( \text{body}(a, [(m1, [], []), (m2, [fly], [1])]), \) producing two negative goals and eventually selecting the positive example. The use of the partial rule favours better handling of the derivation. Since \( \text{pr} \) is an abducible predicate, the rule ABD-2 is applied and the current negative goal is added to \( \Delta^* \) while the remaining elements in \( G \), in this case \( \{ \leftarrow r([(m1, [], [])]) \} \) and the other example, are left for the proof to continue the derivation. \( \Delta^* \) acts as a constraint repository: whenever \( \text{pr}([(m1, [], []), (m2, [fly], [1])]) \) is abduced in the same branch of the proof, then the constraint is activated and the proof verifies that the rest of the negative goal \( \text{body}(a, [(m1, [], []), (m2, [fly], [1])]) \) fails. This is done “lazily”: only if, as a result of goals generated from
other examples, the \( pr([(m_1, [], []), (m_2, [fly], [1])]) \) is selected in a positive goal. Given Proposition 5.5, this happens whenever a rule that includes \( penguin(X) \leftarrow can(X, fly) \) appears in the final solution. In the particular case of the example, the negative goal is never activated thus part of the derivation is correctly disregarded.

The following state is obtained from the state \( S_3 \):

\[
S_4 = \{ \{ \leftarrow r([(m_1, [], [])]) \} \cup \{ penguin(b) \}, \emptyset, \\
\{ \leftarrow pr([(m_1, [], []), (m_2, [fly], [1])]), body(a, [(m_1, [], []), (m_2, [fly], [1])]), \emptyset \}. 
\]

Applying ABD-2 to the remaining negative goal, the following state is obtained:

\[
S_5 = \{ \{ penguin(b) \}, \emptyset, \{ \leftarrow r([(m_1, [], [])]) \} \cup \\
\{ \leftarrow pr([(m_1, [], []), (m_2, [fly], [1])]), body(a, [(m_1, [], []), (m_2, [fly], [1])]), \emptyset \}. 
\]

The proof continues with the selection of \( penguin(b) \). The derivation of an abductive hypothesis for \( penguin(b) \) is constrained by \( \Delta^* \). In fact, during the derivation similar steps to those performed for \( penguin(a) \) are performed (differently from before the goals are positive here). In this case the possible solution that includes the abducible \( pr([(m_1, [], []), (m_2, [fly], [1])]) \) is excluded as a consequence of the fact that such abducible is part of a constraint in \( \Delta^* \). Eventually the proof terminates and returns as the only solution the following:

\[
\Delta = \begin{cases} 
pr([(m_1, [], []), (m_2, [swim], [1])]) \\
r([(m_1, [], []), (m_2, [swim], [1])])
\end{cases} \tag{5.7}
\]

From which the following inductive hypothesis \( H = rt^{-1}(\Delta) \) can be extracted:

\[
H = \begin{cases} 
penguin(X) \leftarrow can(X, swim). \\
\end{cases} \tag{5.8}
\]

Note that, compared to the solutions derived by TOPLOG in Example 3.3, only solutions that cover the positive example and do not cover the negative example are returned. Therefore the TOPLOG’s phase where one of the hypotheses is searched for amongst those that cover the positive example (and possibly the negative) is not required.

The use of the partial rule mechanism also enables the use of integrity constraints both to restrict the search space and to prune the search tree before a final solution is derived. For instance, in the even-odd problem discussed in Section 5.1.1, it is possible to instantiate integrity constraints that avoid solutions of the type \( odd(X) \leftarrow even(X) \). These integrity constraints state that an abductive solution cannot contain the abducibles that lead to such a solution. In particular in order to avoid that the \( even \) and \( odd \) literals use the same variable,
we instantiate the following set of integrity constraints:\footnote{The \textit{member} atom is true if the first argument is an element of the second argument. We also used the standard Prolog bar notation where, given \( l = [h|t] \), \( h \) (the head) is the first element of the list \( l \) and \( t \) (the tail) is given by all the elements in \( l \) except \( h \).}:

\[
I = \begin{cases} 
\leftarrow \text{pr}(([oh, [], []])[L]), \text{member}((be, [], [1]), L) \\
\leftarrow \text{pr}(([oh, [], []])[L]), \text{member}((bn, [], [1]), L) 
\end{cases} \tag{5.9}
\]

The integrity constraints state that it is impossible to have an abductive solution that includes a rule that uses the head mode declaration \textit{oh} and has a condition that uses the mode declaration \textit{bn} or \textit{bn} with the input variable linked to the first variable appearing in the rule. Note that the abducible predicate involved refer to partial rules, so the proof procedure is able to detect violations early in the proof procedure, rather than discarding the rule when it is fully formed.

### 5.4.2 Extensions of the abductive procedure

Whenever a partial rule is abduced a variation is occurring in the inductive hypothesis being derived. This can be exploited to detect the crucial points in the derivation where the partial inductive solution is refined and these states can be used to inform the search by means of heuristics. There are two main reasons why this is useful. Firstly, it is preferable to inspect all the partial abductive solutions derived within the search, since in many cases the search can be halted whenever a node corresponds to a satisfactory hypothesis, avoiding the derivation of a success node. If a partial abductive solution corresponds to an inductive hypothesis the procedure can terminate. Moreover the search can be guided by heuristics that, despite being not as effective as in the monotonic case (since the nonmonotonicity affects in general also the heuristics, e.g. the effect of adding a condition to a rule is not predictable), focus the search on “promising” states. Secondly, the use of heuristics in the search improves efficiency for classes of problem that support monotonic learning. A sequential covering-based extension of \textsc{tal} presented in Section 5.6 makes use of such heuristics.

In order to integrate heuristics, the abductive proof procedure has been extended to take a heuristic function of the partial solution into account and direct the search accordingly. The original depth-first strategy of the abductive system is replaced by a best-first strategy that collects all the states in a list ordered by the heuristic function being used. A state \( S \) in this new abductive procedure is given by the tuple \( \langle \gamma, G, \Delta, \Delta^*, C \rangle \), where \( \gamma \) is a real number that denotes the loss value of the state. Considering the inference operators, we can distinguish two types of states: \textit{abductive states}, where inference operators are applied that do not affect the current solution, and \textit{inductive states}, where the abductive (and thus also the inductive hypothesis) is changing. Inductive
states are those resulting from the use of the ABD-1 operator, whilst all the others are abductive states. The loss function must be evaluated when the only states available are inductive, thus matching the behaviour of other ILP systems (e.g. FOIL) that operate through cycles of refinement and evaluation of the hypothesis. In order to do this the loss associated with abductive states is kept lower than the loss associated with inductive states so that they always have priority. Specifically, the loss $\gamma$ after the application of any inference rule is $-\inf$ with the exception of states obtained from the application of the rule ABD-1, that correspond to inductive states, where the loss is evaluated.

A mode of execution of TAL, called TALH (TAL with Heuristics), has been developed. It is based on an abductive procedure that implements an A* style search. During the search, every time the inference rule ABD-1 is applied, the hypothesis equivalent to the new abductive solution $H = rt(\Delta)$ is evaluated and saved as the best hypothesis if the associated loss is the best so far. Also a pruning function can be instantiated to prune branches of the proof that generate unsatisfactory solutions. As shown in Algorithm 3, the pre-processing generates a top theory (that contains pr/1 abducibles in this case). Also, the new algorithm accepts as input two types of examples. A first type, called goal examples, is used as the goal of the abductive procedure (similarly to the example selected in PROGOL to generate the bottom clause, but in our case the examples can be in general more than one), while the rest of the examples are used to test the partial solutions.

The modified abductive procedure is called ABDUCE2 and shown in Algorithm 4. The procedure keeps track of the best abductive hypothesis, taking care of the translation of the abductive solutions into inductive hypotheses. The arguments of ABDUCE2 are a set of abductive states (initially $(+\inf, E_g, \emptyset, \emptyset, \emptyset)$ is the only state), the current best solution, the current lowest loss and a set of examples that are used in the loss function. The procedure selects the state with the lowest loss amongst those provided as input (line 1 of Algorithm 4) and computes all the child states using the operators in Table 5.1. Then for each of these states the corresponding loss is calculated (line 5). Note that the loss is calculated with respect to all the examples $E$ and not only the goal examples. A PRUNE procedure is used to prune “uninteresting” states. For example, in monotonic problems that require perfect coverage, it can be used to prune branches of the tree that contain a partial hypothesis that entails a negative example. Then the new loss is compared to the best so far and the solution is saved if it is better (it has a lowest loss) than the one currently saved. A TERMINATE procedure checks whether the search should terminate or not (e.g. when perfect coverage is required the search terminates when a satisfactory hypothesis is found). If there are states available the procedure calls itself recursively using the new set of states available.

Example 5.3 Consider Example 5.2 and the set $E_g = \{\text{penguin}(b)\}$. We define the procedure LOSS($\Delta$, $E$) so that it returns the number of examples entailed by $rt^{-1}(\Delta)$ together with the background knowledge.
Algorithm 3 TALH

**Inputs:** $E_g$ goal examples; $E$ examples; $B$ background theory; $M$ mode declarations

**Outputs:** $H$ hypothesis

$T = \text{PRE-PROCESSING}(E_g, B, M)$

$H = \text{ABDUCE2}((+\infty, E_g, \emptyset, \emptyset), \emptyset, +\infty, E)$

Algorithm 4 ABDUCE2

**Inputs:** $\Psi$ set of states of the proof procedure, $best_h$ current best hypothesis, $l_{loss}$ current lowest loss, $E$ set of test examples

**Outputs:** $best_h$ current best hypothesis

1: $S = \text{SELECT}(\Psi)$
2: $\Psi' = \text{set of all the children of states } S$
3: **for all** $S* \in \Psi'$ **do**
4: **if** $S* = \langle \gamma, G, \Delta, \Delta^*, C \rangle$ is obtained from ABD-1 **then**
5: $new_l = \text{LOSS}(\Delta, E)$
6: $S* = \langle new_l, G, \Delta, \Delta^*, C \rangle$
7: **if** $\text{PRUNE}(\Delta, E, l_{loss})$ **then**
8: $\Psi' = \Psi' \setminus \{S*\}$
9: **end if**
10: **if** $new_l < l_{loss}$ **then**
11: $best_h = rt^{-1}(\Delta)$
12: $l_{loss} = new_l$
13: **end if**
14: **end if**
15: **end for**
16: $\Psi_{new} = \Psi' \cup \Psi \setminus \{S\}$
17: **if** $\Psi_{new} = \emptyset$ **or** $\text{TERMINATE}(best_h, l_{loss}, \Psi_{new})$ **then**
18: **return** $best_h$
19: **else**
20: **return** $\text{ABDUCE2}(\Psi_{new}, best_h, l_{loss}, E)$
21: **end if**
\textsc{terminate}(h, l, \Psi) \text{ returns } \text{true} \text{ if all the examples are correctly entailed (the positive examples are covered and the negative examples are not) and false otherwise. Since the problem is monotonic and we are only learning a single rule, } \text{prune}(\Delta, E, l) \text{ can safely be instantiated as a function that returns true if all the positive examples in } E \text{ are covered and false otherwise. In fact in a monotonic setting we know that no additional condition can cause a positive example that is not currently covered to be entailed. In other words the terminate procedure would never return true for states generated from branches that are pruned.}

In this particularly easy example it is easy to observe that the first application of \textsc{abd-1} adds to \Delta (in different branches of the derivation) one of the following three abducibles: \( r([m1, \emptyset, \emptyset]) \), \( \text{pr}([(m1, \emptyset, \emptyset), (m2, [\text{swim}], [1])] \), \( \text{pr}([(m1, \emptyset, \emptyset), (m2, [\text{fly}], [1])] \). \( \Delta = \{r([m1, \emptyset, \emptyset])\} \) corresponds to the rule penguin(\( X \)). This state has loss 0 (one positive example minus one negative example covered) and it is not pruned as it does not cause any negative example to be covered. \( \Delta = \{\text{pr}([(m1, \emptyset, \emptyset), (m2, [\text{fly}], [1])]\} \) corresponds to the rule penguin(\( X \)) \( \leftarrow \text{can}(X, \text{fly}) \). This state has loss −1 since one negative example and no positive examples are covered. This branch is pruned because the current hypothesis does not entail any positive example. \( \Delta = \{\text{pr}([(m1, \emptyset, \emptyset), (m2, [\text{swim}], [1])]\} \) corresponds to the rule penguin(\( X \)) \( \leftarrow \text{can}(X, \text{swim}) \). This rule causes the search to terminate since it covers the positive and not the negative example.

It is interesting to observe how in this example we obtain the same result as in Example 5.2 where the full derivation was being performed with no pruning. Here the number of states generated is much smaller since in the previous case all the branches also processed a negative example. In the case presented here instead, the negative example is only used to check the coverage and decide whether to prune or not the branch, not as additional goal for the proof procedure. Also note that in this example the loss function has no effect on the outcome of the derivation because the derivation is terminated before the loss can be used to prioritise a branch (the loss function is in general useful for problems that involve a large number of example and more complex rules where it can be used in a similar way as in other ILP systems).

The solution returned is the one with the lowest loss \( H \). Note that the algorithm is general and can be customised by redefining the \textsc{loss}, \textsc{prune} and \textsc{terminate} procedures. A particular instantiation of the pruning and scoring function, together with additional constraints on the inputs, which guarantees that \( H \) contains only a single rule, can be used to simulate the search in PROGOL, as shown in Section 5.6. Note that if the \textsc{prune} and the \textsc{terminate} procedures in Algorithm 4 always return false (thus do not prune any state and terminate only when there are no states available) then \textsc{talh} is equivalent to \textsc{tal} as it explores the same states possibly in a different order established by the \textsc{select} and \textsc{loss} procedures. Thus in this particular case the completeness of the search of the algorithm is guaranteed. In all the other cases the completeness is not preserved, but it is worth stressing that unlike other ILP systems the choice of the relevant search space is not
constrained by inherent completeness limitations but it is fully under control and customisable by the choice of adequate scoring and pruning functions.

The scoring and pruning mechanisms implemented in TALH enable the use of search strategies commonly employed in ILP. In the reminder of this Chapter we show some possible uses, inspired by these search strategies. A full investigation on the optimal settings is out of the scope of this thesis. In general the choice of particular search strategy or the choice of sacrificing completeness for performance is very much function of the problem at hand.

5.5 Learning settings

For a better understanding of the learning process it is interesting to see how the search can be adapted to capture specific learning tasks. Clearly, because of the completeness result, the hypotheses derived by TAL include those derived by other systems. Here we give an intuition of the computational steps that are involved in restricted cases, comparing them to those of other systems.

Single clause learning

Let us assume (i) only one positive example $e$ is used to generate the hypothesis; (ii) the hypothesis consists of a single rule; (iii) the example uses the predicate $p/n$ that is not defined in the theory, that does not appear in any body mode declaration and for which a mode declaration $m : modeh(p(\bar{t}))$ is in $M$; and (iv) the background theory $B$ is a definite program. The previous assumptions are common for ILP systems that use a sequential covering approach (e.g. PROGOL).

The initial state of the abductive procedure is $S_0 = (+inf, \{p(\bar{s})\}, \emptyset, \emptyset, \emptyset)$. The mode declaration $modeh(p(\bar{t}))$ causes the following rule to be included in the top theory: $p(\bar{t}) \leftarrow body(inp(m^*), [(m, con(m^*), [])])$. Since $p(\bar{s})$ does not appear in any condition in the clauses in $\top_M$ and $B$, only a rule with head corresponding to the MBL $(m, con(m^*), [])$ will be part of the final solution. Thus, in this setting TAL learns a single rule. The abductive hypothesis $\Delta$, and consequently the corresponding inductive hypothesis $H$, is refined when the operator $\text{ABD}^{-1}$ is applied, abducing a partial rule atom $pr([a_1, \ldots, a_i])$.

Based on Proposition 5.5, when this happens the effect on the final rule is that a new condition is added. This is ultimately equivalent to the use of a refinement operator of the type $\rho(r) = r \cup \{l\}$.

It is interesting to characterise also what condition $l$ is added to the rule. Let us consider the case $pr([a_1, \ldots, a_i])$ is added to $\Delta$, where $a_i = (m_b, con(s^*), links)$, for a given $m_b : modeb(s) \in M$. Equivalently the literal $l$ represented by the MBL $a_i$ is added to the rule represented by the MBR $[a_1, \ldots, a_{i-1}]$.
is in the solution of a success node. Then all the other literals in the body of the rule of the top theory being used must be proved by the proof procedure and $s^\ast \theta$ will be one of these goals. This means that $s^\ast \theta$ is proved by the proof procedure, using only the UNF-1 and EQ-2 operators on rules in the background theory. In fact, by construction, $s^\ast \theta = l \theta$ is proved using a restriction of SLDNFA that is equivalent to SLD on rules of $B$, since the only clauses that can be used to prove $l \theta$ are in $B$.

Summing up, the literal being added to a partially constructed rule that appears in a success node during the proof procedure is such that $B |\models l \theta$. It is easy to verify that the $pr$ abducible is only added to $\Delta$ if $B |\models l \theta$. In PROGOL the most specific clause $h \leftarrow \delta_1, ..., \delta_n$ is such that $B |\models \text{SLD}_{b_i} \delta_i, i = 1, ..., n$. The search procedure derives the final rule in which for each condition $b_j$ in the body there is a substitution $\theta$ such that $b_j \theta = \delta_n$. Thus both in TAL and PROGOL the literals appearing in the body of the rule are a generalisation of consequences of the theory.

**Example 5.4** Consider the following problem and the positive example $p(a)$:

$$M = \begin{cases} 
m1 : modeh(p(+any)) \\
m2 : modeb(q(+any, \#any)) \\
m3 : modeb(q(+any, -any)) 
\end{cases}$$

$$B = \begin{cases} 
q(a, b). \\
q(d, a).
\end{cases}$$

The first four states of a branch of the abductive procedure (equality inference rules and unfolding on append and link literals are omitted for brevity) are:

$S_1 = \langle \{p(a)\}, \emptyset, \emptyset, \emptyset \rangle$.

$S_2 = \langle \{pr([m1, [], []]), \text{body}([a], [(m1, [], [])]), \emptyset, \emptyset, \emptyset \rangle$.

$S_3 = \langle \{\text{body}([a], [(m1, [], [])]), \{pr([m1, [], []])\}, \emptyset, \emptyset \rangle$.

$S_4 = \langle \{q(a, C), pr([m1, [], []], (m2, [C], [1])), \text{body}([a], [(m1, [], [])], (m2, [C], [1])), \{pr([m1, [], []])\}, \emptyset, \emptyset \rangle$.

At this point the proof verifies that $B |\models q(a, C) \theta$ for some $\theta$. Since this succeeds, with $\theta = \{C/b\}$ then the partial rule atom $pr([m1, [], []], (m2, [b], [1]))$ is abduced. The MBR encoded in the argument represents the partial rule $p(X) \leftarrow q(X, b)$. Note that the top theory here has the effect of appropriately constraining the search since $q(X, a)$ is never a candidate condition in the body.
Non-observational single clause learning

Let us assume a less restrictive setting: (i) only one positive example \( e \) is used to generate the hypothesis; (ii) predicates used in body mode declarations do not depend on predicates that appear in head mode declarations.

This setting allows us to focus on the problem of non-observational learning. A common solution is based on abductive reasoning. ALECTO for example computes abductive solutions \( \Delta \) for the ALP task \( (e, B, A, \emptyset) \), where \( A \) is given by the space of facts \( R_{M^+} \) obtained from the head mode declarations in a set \( M \). Each element \( a \in \Delta \) is then used to construct a rule using inverse entailment.

TAL performs abductive reasoning as a particular case, whenever the mode declarations only contain head mode declarations. In this case, rules (4.2) are not used in the top theory as they are generated from body mode declarations. Given a mode declaration \( m : modeh(p(T)) \), the only way for a goal \( p(T) \) to succeed is to apply the operator \( UNF^{-1} \) using the following two rules in the top theory:

\[
\begin{align*}
p(T) & \leftarrow body(inp(p(T)), [(m, con(p(T)), [])]) \\
body(I, NR) & \leftarrow r(NR)
\end{align*}
\]

In this case \( r([(m, con(s^*), [])]) \) can be abduced and the final hypothesis will thus contain the MBR for an atom obtained from \( m \). According to Theorem 4.4 the final hypothesis will contain an atom compatible with \( m \) with no body that abductively explains the given example. If body mode declarations are added to this scenario, rules are constructed that contain \( (m, con(s^*), []) \) in the MBR for the head and other MBLs, constructed as in the case of Section 5.5. So, like for ALECTO, the final hypothesis will contain in the head atoms that abductively explain the given example.

Non-observational multi-predicate and recursive learning

We consider now the most general case, with no restriction on the inputs of the algorithm. As a consequence of completeness, TAL is able to derive hypotheses that contain recursion and define multiple predicates. Abstracting from the steps of the abductive procedure, when adding elements \( pr(a) \) to the body of a rule, a new goal (an instance of the literal represented by \( a \)) is added to the set of the goals. The new goal can be proved using clauses in the background knowledge but also by creating a new clause that leads indirectly to success in proving the goal. The new clause can reuse the current abductive solution and equivalently the partially constructed
rules, thus enabling recursive (if the clauses refer to the same predicate) and multi-predicate learning.

This is best explained by referring to a variant of the even-odd example. This time for a more befitting comparison with other ILP systems we consider a version of the problem that does not involve negation. The top theory containing the partial rule conditions is as follows:

\[ B = \{ \text{even}(0) \} \]
\[ E = \{ \text{odd}(s(s(0))) \}, \text{not \ odd}(s(s(0))) \} \]
\[ M = \{ \begin{align*}
  \text{eh} : & \ modh(\text{even}(+\text{nat})). \\
  \text{oh} : & \ modh(\text{odd}(+\text{nat})). \\
  \text{eb} : & \ modh(\text{even}(+\text{nat})). \\
  \text{ob} : & \ modh(\text{odd}(+\text{nat})). \\
  \text{sb} : & \ modh(+\text{nat} = s(−\text{nat})).
\end{align*} \]

\[ \mathcal{T}_M = \{ \begin{align*}
  \mathcal{T}_1 &: \text{even}(X) \leftarrow \\
  & \text{pr}([[\text{eh}, [], []]]), \\
  & \text{body}(X, [[\text{eh}, [], []]]) \\
  \mathcal{T}_2 &: \text{odd}(X) \leftarrow \\
  & \text{pr}([[\text{oh}, [], []]]), \\
  & \text{body}(X, [[\text{oh}, [], []]]) \\
  \mathcal{T}_3 &: \text{body}(\text{Inputs, Rule}) \leftarrow \text{r}(\text{Rule}) \\
  \mathcal{T}_4 &: \text{body}(\text{Inputs, Rule}) \leftarrow \\
  & \text{append}(\text{Rule}, [[\text{oh}, [], []]], \text{NRule}), \\
  & \text{link}([X], \text{Inputs, Links}), \\
  & \text{odd}(X), \\
  & \text{body}(\text{Inputs, NRule}) \\
  \mathcal{T}_5 &: \text{body}(\text{Inputs, Rule}) \leftarrow \\
  & \text{append}(\text{Rule}, [[\text{eh}, [], []]], \text{NRule}), \\
  & \text{link}([X], \text{Inputs, Links}), \\
  & \text{even}(X), \\
  & \text{pr}(\text{NRule}), \\
  & \text{body}(\text{Inputs, NRule}) \\
  \mathcal{T}_6 &: \text{body}(\text{Inputs, Rule}) \leftarrow \\
  & \text{append}(\text{Rule}, [[\text{sb}, [], []]], \text{NRule}), \\
  & \text{link}([X], \text{Inputs, Links}), \\
  & \text{s}(X) = Y, \\
  & \text{append}(\text{Inputs}, [Y], \text{Outputs}), \\
  & \text{pr}(\text{NRule}), \\
  & \text{body}(\text{Outputs, NRule})
\end{align*} \]

Figure 5.1 illustrates the abductive tree and Figure 5.2 details some crucial abductive states. The initial goal contains the given examples and the goal \text{odd}(s(s(s(0)))) is selected in \( S_0 \). Since no definition exists for it in \( B \), the proof procedure uses a rule in \( \mathcal{T}_M \), \( \text{pr}([[\text{oh}, [], []]]) \) is abduced in \( S_1 \) that corresponds to the rule \text{odd}(X) and the proof has at state \( S_2 \) the first choice point.

In \( S_2 \), the literal \( \text{body}([s(s(s(0))], [[\text{oh}, [], []]]) \) (see Figure 5.2) is selected and it can be resolved with the head of the four rules in the top theory \( \mathcal{T}_3, \mathcal{T}_4, \mathcal{T}_5, \mathcal{T}_6 \). Each of these rules corresponds to a possible refinement of the current rule \text{odd}(X). The two branches on the left fail because of additional integrity constraints used to rule out solutions of the type \text{odd}(X) \leftarrow \text{odd}(X) and \text{odd}(X) \leftarrow \text{even}(X)^6. The fourth branch from the

\footnote{Without these additional integrity constraints the abductive tree would contain two infinite branches in places of the failure nodes.}
left finalises the rule $\text{odd}(X)$. In $S_3$ the selected goal is the negative example $\text{not odd}(s(s(0)))$. This branch ultimately fails because the current abductive solution causes the goal $\text{odd}(s(s(0)))$ to succeed. This is expected since the inductive solution $\text{odd}(X)$ covers the negative example.

The remaining branch, obtained from $T_6$, updates the abductive solution in $S_4$ that here corresponds to the rule $\text{odd}(X) \leftarrow X = s(Y)$. In $S_5$, like in $S_2$, there is another choice point, again due to the four rules in the top theory $T_3, T_4, T_5, T_6$. In each of the four branches the rule in the current hypothesis is extended with one of the three mode declarations or finalised. We concentrate on only one of these branches, the one that applies unfolding on the current condition with rule $T_5$ (the others lead eventually to finite failure).

One of the conditions links the input variables to available variables. Here the condition results in a choice point, i.e. the input variable in the condition can be unified either with $s(s(s(0)))$ and in general result in a link with the input variable in the head, or with $s(s(0))$ and result in a link with the output variable of the other condition in the body. Crucially, the two children of $S_6$, $S_7$ and $S_8$, result in a set of goals containing respectively $\text{even}(s(s(0)))$ and $\text{even}(s(s(s(0))))$. These two goals can be seen as artificial examples. They play the same role of the examples provided but they are generated by the procedure. Considering $S_7$, the proof selects the goal $\text{even}(s(s(0)))$. This must be proved using clauses in the top theory, namely $T_1$. We omit most of the derivation for brevity, that follows the same steps that led to state $S_7$. Differently from before, the original goal in $S_7$ is $\text{even}(s(s(0)))$ rather than $\text{odd}(s(s(0)))$ so in this portion of the tree one of the branches contains the state $S_8$, which is obtained after constructing another partial rule $\text{even}(X) \leftarrow X = s(Y)$. The goal of this state is particularly meaningful (see Figure 5.2). It states that, for this branch to succeed:

- The goal $\text{odd}(s(0))$, an “artificial” example, conditional to the rule constructed so far, must succeed.
• The use of the clause \( \text{background knowledge} \), which makes it succeed. The remaining goals either unify with atoms in one of the clauses in the top theory (eventually producing a failure node) and with the only clause in the state

In other words, the candidate solution \( \{ \text{even}(X) \leftarrow X = s(Y) \} \cup \{ \text{odd}(X) \leftarrow X = s(Y), \text{even}(X) \} \) is an inductive solution if \( \text{odd}(s(0)) \) and \( \text{not odd}(s(0)) \) succeed. Again we skip a considerable part of the derivation, as shown in Figure 5.1. The proof uses the rules in the top theory to prove \( \text{odd}(s(0)) \) following the same steps as for the original example \( \text{odd}(s(s(0))) \) but this time the artificial example produced is \( \text{even}(0) \) in state \( S_9 \). Also, differently from before the abducibles selected as goals are not added to \( \Delta \) because they can be unified with atoms that are already in it.

The goals in \( S_9 \) contain \( \text{body}/2 \) goals that succeed by unfolding with rule \( \top_3 \), the abducibles contained in \( S_8 \), and the negative example \( \text{not odd}(s(0)) \). But this time the selected goal is \( \text{even}(0) \), which can be resolved with one of the clauses in the top theory (eventually producing a failure node) and with the only clause in the background knowledge, which makes it succeed. The remaining goals either unify with atoms in \( \Delta \) or result in the use of the clause \( \top_3 \) thus leading to a success state \( S_{10} \) with the following abductive solution:

\[
\Delta = \begin{cases} 
\text{pr}([([oh, [], []])]) \\
\text{pr}([([oh, [], []]), (sb, [], [1])]) \\
\text{pr}([([eh, [], []])]) \\
\text{pr}([([oh, [], []]), (sb, [], [1])]) \\
\text{pr}([([oh, [], []]), (sb, [], [1]), (eb, [], [2])]) \\
\text{pr}([([eh, [], []]), (sb, [], [1]), (ob, [], [2])]) \\
\text{pr}([([eh, [], []]), (sb, [], [1]), (ob, [], [2])]) \\
\end{cases}
\]  \tag{5.10}
That corresponds to the inductive solution:

\[
H = \begin{cases}
  \text{even}(X) \leftarrow \\
  X = s(Y), \\
  \text{odd}(Y) \\
  \text{odd}(X) \leftarrow \\
  X = s(Y), \\
  \text{even}(Y)
\end{cases}
\]

(5.11)

This examples shows some features of the system that although expected, given the completeness property, emerge as peculiarities of the system and not as a result of ad hoc solutions:

- Non-observational learning is performed since \text{even} does not appear in the examples
- Multiple interdependent predicates are learnt within the same task.
- A recursive definition is learnt. Note that the system is also capable of learning the base case of the recursion. In fact, if \text{even}(0) was not part of the theory it could have been proved using the rule in the top theory derived from an additional mode declaration \text{modeh} (\text{even}(\#\text{nat})).
- The Herbrand base involved in the learning task is infinite.

Although it is not shown in this example, TAL is able to perform predicate invention as a direct consequence of its completeness, whenever the mode declarations include a finite number of predicates that are not defined in the background theory and that are not used in the example. As in [Khan et al., 1998], the actual outcome of the learning is strongly influenced by an appropriate bias on the rule space.

### 5.6 Cover loop approach in TAL

As shown in Section 2, for a certain class of nonmonotonic ILP problems it is still possible to control the search based on the predictable effects of specialising a rule or adding a new rule to a partial hypothesis. Thus, in order to handle this class more efficiently and also ILP over definite theories, we have implemented a mode of execution of TAL that is based on the cover loop set approach.

The particular loss and pruning functions adopted resemble those used in PROGOL. The loss function is defined as the number of negative examples covered by the hypothesis, plus the complexity of the hypothesis.
Algorithm 5 COVER-LOOP-TAL

Inputs: $E$ examples; $B$ background theory; $M$ mode declarations; $I$ integrity constraints

Outputs: $H$ hypotheses

$H = \emptyset$

while a termination condition holds do

$e \in E$

$H_{\text{new}} = \text{TAL-H}(e, E, B, M, I)$

$H = H \cup H_{\text{new}}$

Let $E$ be the set of examples that are not covered by $H$

end while

return $H$

minus the positive examples (see Section 3.2):

$$\text{LOSS} (\Delta, E) = l_{\text{progol}} (rt^{-1}(\Delta), E)$$

The pruning function calculates a lower bound for the current rule that is given by

$$lb(\Delta, E) = -\text{positive-covered}(rt(\Delta), E) + \text{complexity}(rt(\Delta), E) + 1$$

that assumes that the current hypothesis will be refined optimally to cover only the positive examples and none of the negative after one literal is added, and prunes the branch if the lower bound is higher than the lowest lost currently associated to the best solution, because according to the effect of the monotonicity assumption on coverage, it is impossible that the current hypothesis is refined to become a hypothesis that is better than the best solution so far:

$$\text{PRUNE}(\Delta, E, \text{best} . l) = \begin{cases} 
\text{true} & \text{if } lb(\Delta, E) > \text{best} . l \\
\text{false} & \text{otherwise}
\end{cases}$$

This mode of execution employs the ordering defined in Chapter 6 to reduce redundant states.

We can observe that a different search strategy and heuristic would result in an execution mode that is similar to the system FOIL. Using a greedy search strategy and information gain as scoring function, the system would iteratively add a condition to the current rule and pick the one with the lowest loss. In contrast to FOIL however, the only conditions considered are such that they cover the selected example $e$. In general the functions used can be set by the final user based on domain requirements.
5.7 Implementation and evaluation

TAL has been developed in Prolog. In particular, the SICStus Prolog system was used to implement both the transformations and the heuristics extension, in addition to the original abductive procedure that is available for SICStus [Ma, ]. SICStus is considered slower than other Prolog implementations but it was chosen because, in our opinion, it is more reliable than the alternatives and thus more appropriate for the development of a prototype. The current implementation provides a set of options that can be used to customise the search. These include: the maximum depth of the abductive proof, the search strategy, the selection strategy in the proof (e.g. whether the types are tested before or after the actual condition of the body), and whether the search is monotonic or not.

Despite the theoretical importance of a framework that fully supports nonmonotonicity in ILP, the practical use of the system is restrained by the underlying computation mechanisms. When using TAL in its base implementation with no heuristics, a high portion of the search space is replicated, since the system does not rule out complex dependencies amongst rules and examples. Furthermore, the use of negation in most cases requires that failure is ensured for universally quantified variables, which in turn produces a large amount of negative goals. In Chapter 6, we mitigate these issues and provide an alternative solution by replacing the underlying computation mechanism and introducing strategies that lead to a considerable increase in performance.

Given that for applied nonmonotonic problems, the solution provided in Chapter 6 produces better results, here we use the monotonic restriction of TAL for an empirical evaluation in comparison to ALEPH, that implements the PROGOL inverse entailment approach and that can be considered the reference ILP system, and to TOPLOG that, like TAL, is based on the use of a top theory.

We chose some datasets that were previously used in ILP experiments and that capture real world problems that require a relational representations: mutagenesis [Srinivasan et al., 1996], carcinogenesis [Srinivasan et al., 1997], alzheimers-amine [King et al., 1995] and DSSTox [Richard & Williams, 2002]. All these datasets share a common structure where an active molecule must be characterised based on some properties. The particular type of activity is specific to the dataset. This type of dataset assumes a non-perfect representation of the domain. In fact, the properties may not be the only relevant information to the characterisation of activity. Thus in this case we expect the system to have less than perfect accuracy.

The accuracy measures for ALEPH and TOPLOG are reported here from [Santos, 2010]. The configuration of the two other systems is reported in [Santos, 2010]. Here the same set of mode declarations and homogeneous conditions have been used in the configuration of TAL. The tested datasets are some of the most challenging for ILP and confirm that TAL, under the discussed constraints, is able to explore the search space that is also ex-
explored by exhaustive search in inverse entailment systems. The accuracy for the tested problems is comparable to that of ALEPH and TOPLOG.

The computation time in TAL is on average higher by one to two orders of magnitude compared to the computation time for the other two systems. This is not surprising since, as reported in [Santos, 2010] the computation time for a given problem in SICStus is on average ten times higher than the computation time in YAP, the system used to implement ALEPH and TOPLOG and the technical characteristics of the machine used for the experiment in TAL are lower than those of the machine used in the original experiments. Moreover we do not expect faster computations given that TAL is in the prototype stage and not designed for performance.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>TAL</th>
<th>ALEPH</th>
<th>TOPLOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>carcinogenesis</td>
<td>65.6% ± 9.9%</td>
<td>60.2% ± 8.7%</td>
<td>57.1% ± 14.4%</td>
</tr>
<tr>
<td>alzheimers-amine</td>
<td>67.8% ± 3.8%</td>
<td>74.2% ± 3.9%</td>
<td>67.7% ± 5.6%</td>
</tr>
<tr>
<td>dsstox</td>
<td>68.6% ± 8.5%</td>
<td>70.2% ± 6.4%</td>
<td>68.1% ± 2.9%</td>
</tr>
<tr>
<td>mutagenesis</td>
<td>77.1% ± 8.3%</td>
<td>78.9% ± 11%</td>
<td>80.0% ± 9.5%</td>
</tr>
</tbody>
</table>

Table 5.2: Accuracy over a 10-fold cross-validation (and standard deviation) on the datasets for TAL, ALEPH and TOPLOG

5.8 Discussion and related work

The example discussed in 5.5 is particularly useful to discuss the differences with other ILP systems. We use it in this section as a starting point for comparisons.

We recall from Section 3.7.1 that PROGOL constructs a most specific clause $\bot$ from a given positive example $e$, such that $B \cup \{\neg e\} \models \neg \bot$. In the case of the even-odd example, as already discussed in [Yamamoto, 1997], it is not possible for PROGOL to derive the hypothesis in 5.11 ($H = \{\text{even}(X) \leftarrow X = s(Y), \text{odd}(Y)\}$ $\cup \{\text{odd}(X) \leftarrow X = s(Y), \text{even}(Y)\}$). This is a consequence of the fact that a single clause $\bot$ is derived from a single example, and thus a generalisation of it will never contain two clauses. Another type of incompleteness is highlighted in [Yamamoto, 1997], where it is shown that even for the simplified case where $B = \{\text{even}(0)\} \cup \{\text{even}(s(X)) \leftarrow \text{odd}(X)\}$ and $E = \{\text{odd}(s(s(s(0))))\}$ the correct hypothesis $\text{odd}(s(X)) \leftarrow \text{even}(s(X))$ cannot be derived. This shows how TAL, on the restricted case of definite programs, is able to derive a larger set of hypotheses than PROGOL.

XHAIL solves PROGOL’s incompleteness issues. Indeed, XHAIL is able to derive the solution in 5.11 but
it is instructive to point out some limitations of the approach in this example. In the first abductive phase, an abductive hypothesis $\Delta$ is computed such that $B \cup \Delta \models e$. The preference criterion over such sets $\Delta$ is for subset-minimal hypotheses. In this case the preferred abductive hypothesis is $\Delta = \{\text{odd}(s(s(0)))\}$. After the deductive and inductive phase of XHAIL, the solution (5.11) is not found, since it contains two rules. XHAIL backtracks to the first abductive phase and computes a new solution $\Delta$, this time not a minimal one. The choice in this case is not aided by any preference criterion, thus, if the set of candidates is infinite, the search for a useful solution is blind and may consider an infinite number of candidates that cannot be used to construct the desired inductive hypothesis (e.g. $\Delta_1 = \{\text{odd}(s(s(0)))\}, \text{odd}(s(s(s(0))))\}$, $\Delta_2 = \{\text{odd}(s(s(0)))\}, \text{odd}(s(s(s(0))))\}$, etc.). In TAL this search is driven by mode declarations. In particular a partial hypothesis is created together with some conditional goals. The search is bound by the hypothesis space, which in this case is much more limited than the space of abductive explanations of the example. It is legitimate to formalise this problem in a finite domain, also in this case, although the problem becomes decidable, the number of candidate solutions for the abductive phase in XHAIL greatly exceeds the number of candidate hypotheses.

In Section 3.7.4 we introduced TOPLOG, a recent ILP system that inspired TAL. TAL operates like the hypothesis finding phase of TOPLOG whenever the given problem is monotonic and a single positive seed example is used. For nonmonotonic problems TAL extends the TOPLOG derivation by adopting an abductive proof procedure and by constructing hypotheses rather than single rules. Referring to Section 5.5, TOPLOG can only use the positive example $\{\text{odd}(s(s(0)))\}$ to instantiate the derivation, as it is undefined how the system operates for negative examples. TOPLOG would not derive either of the rules in 5.11 since they only entail the positive example if taken together.

5.8.1 Other nonmonotonic ILP systems

We review here some other solutions to nonmonotonic ILP in comparison to TAL. [Dimopoulos & Kakas, 1995], one of the earliest approaches to the learning of logic programs under nonmonotonicity, assumes that the concepts to be learnt are organised hierarchically and can include exceptions. In order to do this the authors employ a form of classical negation together with NAF, relying on a priority order amongst the rules in the program. A standard ILP procedure is employed to learn “default” definitions. These rules are allowed to cover negative examples that are later used to expand the default rule through exceptions. The constraint on the hierarchical structure over the concepts to be learnt forbids the learning of the intended hypothesis in the example in Section 5.1.1. The approach supports the learning of concept with exceptions well, but the representation language is not equivalent to a Prolog or an ASP program since it handles negation based on a priority order over the rules.
learnt. Overall the scope is more limited than the scope of the approach presented here.

Similarly to SLDNFA, used here as the abductive back-end of TAL, [Martin & Vrain, 1995] relies on the 3-valued completion semantics. A procedure is presented that learns concepts in the single-predicate learning case. The approach follows a traditional sequential covering approach and does not provide a completeness result.

[Yamamoto, 2003] proposes a solution to a formulation of the even-odd problem. A residue $rs(S)$ of a theory $S$ is obtained by filtering out tautological clauses from the clausal theory obtained by negating the clauses in $S$. A hypothesis $H$ is such that $H \models rs(B \cup rs(E))$, i.e. it entails the residue of the background theory together with the residue of the examples. Although this procedure leads to a solution for the even-odd problem, a discussion of the whole computation process is missing. [Yamamoto, 2003] lacks a concrete realisation in a procedure and an implementation that can be validated with respect to larger ILP problems.

IMPARO [Kimber et al., 2009] proposed an extension to IE called induction on failure. Intuitively, when literals in the body of the rules of a certain hypothesis are not consequences of the background theory together with the hypothesis, a new induction task is performed that uses such literals as examples. Interestingly, this is similar to what happens in the abductive derivation in TAL, when artificial examples are created from partial hypotheses. Currently, IMPARO is limited to Horn theories.

CF-INDUCTION [Inoue, 2004] is the only ILP system that supports negation (in this case classical negation) that provides a formal result on completeness. Crucially the system deals with classical negation and thus the procedure involved is monotonic. The system computes hypotheses that consist of full clausal theories, using the inverse entailment framework to derive a generality lower bound. The inference is performed by SOL (Skipping Ordered Linear) resolution [Nabeshima et al., 2003]. As discussed in [Ray, 2005], the main limitations of this approach are the interaction with the user, required for the selection of the so called characteristic clause and the expensive computational steps involved in the conversions to conjunctive normal forms and inverse resolution generalisation.

5.9 Conclusions

The transformation introduced in Chapter 4 provides the basis for the implementation of ILP systems. Similarly to the Inverse Entailment framework that has inspired and driven the development of a number of implementations, we believe ILP can benefit from such a theoretical framework. In this chapter we have introduced an ILP system, called TAL, that makes use of that transformation, from which it inherits the soundness, and, more critically, the completeness properties. TAL can be applied to complex classes of ILP problems like multi-
predicate learning and non-observational predicate learning, solving these in a complete and principled way. We showed how the system can be extended in order to support search heuristics and make use of the facilities of the underlying ALP procedure employed.

By restricting the language of the task, it is possible to constrain the learning system in order to gain efficiency from the common monotonicity assumptions of other ILP systems. We showed how under these constraints TAL learns hypotheses of comparable accuracy with TOPOLOG and the reference ILP system PROGOL in four notable ILP problems. As observed, despite the theoretical properties that ensure that all the desired solutions are within the search space, TAL can be inefficient for problems that make heavy use of integrity constraints and where the derivation is slowed down by a high number of negative goals. In order to address these issues we introduce in the next Chapter a new system, derived from our transformation framework, that adopts a different computational paradigm based on the generation of answer sets.
Figure 5.2: Selected states of the derivation tree.
6 Abductive Learning in Answer Set Programming

Answer Set Programming (ASP) has recently gained recognition as an efficient approach to nonmonotonic logic programming. Building upon the notion of answer set semantics, its success is in part due to the availability of efficient and powerful inference engines and solvers. Growing from the need for better computational support for negation in logic programming, ASP is nowadays an established computational framework that supports an expressive language, includes constraints and aggregates, provides optimisation features and builds on efficient solving algorithms. Unlike Prolog, it requires a finite domain and theories are grounded before they are used for inference. Other features of ASP are discussed in Section 2.4.2. Despite the recent growth of ASP, little has been done to investigate the potential contribution that it can make to the field of Inductive Logic Programming.

Besides the challenges posed by nonmonotonicity, existing ILP systems are not declarative enough. In fact, users are sometimes required to experiment with the ordering of the rules, this being relevant not only to efficiency but even to termination or correctness. Moreover, when the learning task is particularly knowledge-intensive, such systems usually tend to perform redundant computations, thus overloading the total computation time with particularly heavy resolution inferences. This is expecially the case of those ILP applications in which a large portion or all of the admissible solutions are required.

In this chapter we present an instantiation of our learning framework, called ASPAL, that relies on a mapping of mode declarations into partially instantiated rules, and uses an ASP solver to find optimal as well as all possible inductive solutions. We use previous work that describes challenging problems for nonmonotonic ILP to test the applicability of the system.

6.1 An encoding for ASP

In this chapter we present a methodology that adapts our ILP framework, described in Chapter 4, to ASP. The resulting system can be used as an alternative to TAL, presented in Chapter 5. The transformation presented in Section 4.2.1 could be, in principle, coded as an ASP program, since it is proven complete and correct for stable model semantics. However, in this case we would suffer from practical problems and incur inefficiencies. For example, the definition of the append/3 predicate is against the programming style of ASP and a
more appropriate representation of the problem would be to map inductive solutions into models rather than constructing lists. Furthermore the grounding of the definition of such predicates would be particularly costly. This can be avoided by adapting the transformation and ultimately by producing an intermediate theory that represents the language bias in a convenient form for the ASP solver. The method we propose originates from the top theory, but is obtained by applying unfolding on the \textit{body} literals in order to obtain an explicit set of rules to be instantiated.

We gradually present a method for the encoding, starting from the simplifying assumption that a set of rules is given explicitly to represent the rule space. Let us assume \( n \) ordered rules to be defined explicitly as \( R = \{ h_i \leftarrow b_i : i = 1, ..., n \} \) instead of being implicitly defined by mode declarations, where \( b_i \) denotes a list of body literals. Let \( id(r) \) be a function that associates a \textit{unique ID} to each rule in \( R \). We can construct the theory \( \tilde{R} = \{ h_i \leftarrow b_i, \text{rule}(id(h_i \leftarrow b_i)) \} \), and denote the set of the \( n \) body atoms \( \text{rule}(id(r)) \) in \( \tilde{R} \) by \( A \).

Given a subset \( \Delta \) of \( A \), \( \text{hyp}(\Delta) = \{ id^{-1}(a) : \text{rule}(a) \in \Delta \} \) is the set of rules whose body atoms \( \text{rule}(id) \) are in \( \Delta \).

\begin{example}
Let \( R = \{ p(X) \leftarrow r(X, Y) \} \cup \{ p(X) \leftarrow q(X, a) \} \) and let \( id \) be a function defined as \( id(p(X) \leftarrow r(X, Y)) = id1 \), and \( id(p(X) \leftarrow q(X, a)) = id2 \). Then \( \tilde{R} = \{ p(X) \leftarrow r(X, Y), \text{rule}(id1) \} \cup \{ p(X) \leftarrow q(X, a), \text{rule}(id2) \} \) and \( A = \{ \text{rule}(id1), \text{rule}(id2) \} \), \( \text{hyp}(\{ \text{rule}(id1) \}) = \{ p(X) \leftarrow r(X, Y) \} \). Consider \( \Delta = \{ \text{rule}(id1) \} \). Then for any theory \( B \), \( \tilde{R} \cup \Delta \cup B \) has the same answer sets as \( \{ p(X) \leftarrow r(X, Y) \} \cup B \) (excluding the atom \( \text{rule}(id1) \) that is true in every answer set of the former theory).
\end{example}

We refine the previous representation by adding to the body of ordered rules, atoms of the form \( \text{rule}(id(r), \overline{C}) \) where \( \overline{C} \) is a list of variables. We recall that we use \( \text{res}(r, f) \) to denote the \textit{resolvent} \cite{Lloyd1984} of a rule \( r \) and an atomic rule \( f \). Now, for any set \( \Delta \) of atomic rules of the form \( \text{rule}(a, \overline{C}) \), then \( \text{hyp}(\Delta) = \{ \text{res}(id^{-1}(a), \text{rule}(a, \overline{C})) : \text{rule}(a, \overline{C}) \in \Delta \} \). In other words if \( \text{rule}(a, \overline{C}) \) is in \( \Delta \), then \( \text{hyp}(\Delta) \) includes the original rule that corresponds to the ID \( a \) and with all variables appearing in the list \( \overline{C} \) (that was added to its representation) instantiated with the terms in \( \overline{C} \).

We use this encoding to transform the problem of finding solutions of an ILP task into an abductive problem we can represent in ASP. The first argument of the \textit{rule} predicate identifies the structure of the rule being learnt, i.e. the literals in it and how the input variables are linked to output variables, while the second argument of the \textit{rule} predicate is an artifice that delegates the instantiation of constants appearing in the rule to the particular ASP solver used. Note that in this Chapter the identifier used as first argument of the rule predicate plays the same role as the MBR in Chapter 4, as clarified later in this section. The appropriateness of the transformation is more formally captured by the following theorem.

105
Theorem 6.1  Let $B$ be a normal logic program, $\tilde{R}$ a set of rules $r$ of the form $h \leftarrow \tilde{b}, \text{rule}(a, [C_1, ..., C_n])$ where $\text{id}(h \leftarrow \tilde{b}) = a$, and $A$ a set of ground atoms $\text{rule}(a, [c_1, ..., c_n])$ for which there is a rule $h \leftarrow \tilde{b}, \text{rule}(a, [C_1, ..., C_n]) \in \tilde{R}$. For each $\Delta \subseteq A$, $I$ is an answer set of $B \cup \tilde{R} \cup \Delta$ iff $I \setminus \Delta$ is an answer set of $B \cup \text{hyp}(\Delta)$.

Proof: Consider a rule $h \leftarrow \tilde{b}, \text{rule}(a, [C_1, ..., C_n]) \in \tilde{R}$. $\Delta$ contains $m$ atoms $\text{rule}(a, [c_{1,1}, ..., c_{1,n}])$, $..., \text{rule}(a, [c_{m,1}, ..., c_{m,n}])$. We can apply unfolding (that, recalling Theorem 2.3 is an answer set-preserving transformation) and obtain a new theory $\tilde{R}'$ that contains the $m$ rules $\text{res}((h \leftarrow \tilde{b}, \text{rule}(a, [C_1, ..., C_n])), \text{rule}(a, [c_{i,1}, ..., c_{i,n}]))$. If $m = 0$ the rule is not included in $\tilde{R}'$ (one condition is false). We can apply this transformation to each rule in $\tilde{R}$. As a final result we obtain exactly $\text{hyp}(\Delta)$ as defined before. Thus the theory $B \cup \tilde{R} \cup \Delta$ has the same answer sets as $B \cup \text{hyp}(\Delta) \cup \Delta$. Since the elements in $\Delta$ do not appear in the rest of the theory they only contribute to the truth value of $\text{rule}$ atoms in the answer sets of $B \cup \text{hyp}(\Delta) \cup \Delta$; thus for each answer set $I$ of $B \cup \text{hyp}(\Delta) \cup \Delta$, $I \setminus \Delta$ is a answer set of $B \cup \text{hyp}(\Delta)$. Note also that the if and only if statement holds in both direction since the unfolding transformation applied ensures that the original theory and the transformed theory have exactly the same answer sets.

The above representation mechanism can be used to transform the set $\mathcal{R}_M^0$ of ordered rules compatible with given mode declarations $M$ into two special theories $\tilde{\mathcal{R}}_M^0$ and $A_M$ as defined below.

Definition 6.1 (ASP encoding of mode declarations) Let $M$ be a set of mode declarations and $\mathcal{R}_M^0$ the set of all compatible ordered rules. The ASP encoding of $M$ is given by constructing the following two theories:

- $\tilde{\mathcal{R}}_M^0$: for each rule $h \leftarrow \tilde{b} \in \mathcal{R}_M^0$ a rule $h' \leftarrow \tilde{b}', \text{rule}(\text{id}(h' \leftarrow \tilde{b}'), \overline{c})$ is included in $\tilde{\mathcal{R}}_M^0$, where $h' \leftarrow \tilde{b}'$ is obtained from $h \leftarrow \tilde{b}$ by substituting every constant with a different variable and $\overline{c}$ is the list of all the new variables (in order of appearance left to right).

- $A_M$: for each rule $h \leftarrow \tilde{b} \in \mathcal{R}_M^0$, the atom $\text{rule}(\text{id}(h' \leftarrow \tilde{b}'), \overline{c})$ is added to $A_M$, where $h' \leftarrow \tilde{b}'$ is obtained from $h \leftarrow \tilde{b}$ by substituting every constant with a different variable and $\overline{c}$ is the list of all the constants appearing in $h \leftarrow \tilde{b}$ (in order of appearance left to right).

The following theorem is a consequence of Theorem 6.1 and states the semantic equivalence of a given original ILP task and the ILP task obtained using the representation illustrated in Definition 6.1.

Theorem 6.2  Consider the ILP task $\langle E, B, \mathcal{R}_M^0 \rangle$ with loss function $l(E, B, H)$. The set of rules $\text{hyp}(\Delta)$ is an optimal solution for it iff $\Delta$ is an optimal solution for the ILP task $\langle E, B \cup \tilde{\mathcal{R}}_M^0, A_M \rangle$ with loss function $l(E, B, \text{hyp}(\Delta))$. 

106
Proof By definition there is a rule \( r' \) in \( \bar{R}_M^\circ \) obtained from \( r \) by substituting constants with variables and a \( \delta = rule(id(r'), \overline{\sigma}) \in A_M \) where \( \overline{\sigma} \) are the constants appearing in \( r \) such that \( res(r', \delta) = r \). The list of constants that appears in left-to-right order in \( \overline{\sigma} \) is unique so, given a certain ID there cannot be two different atoms in \( A_M \) that can be used to obtain \( r \). Similarly, it is possible to verify that the rule \( r' \) is the only one that can be used to obtain \( r \) (two rules that only differ by a variable renaming are considered the same). Thus, for each rule \( r \in R_M^\circ \) there is one and only one \( \delta \in A_M \) such that \( hyp(\{ \delta \}) = r \). More generally for a given set of rules \( H \subseteq R_M^\circ \) there is one and only one \( \Delta \subseteq A_M \) such that \( hyp(\Delta) = H \).

The theory \( \bar{R}_M^\circ \) respects the assumptions in Theorem 6.1. Thus for each \( \Delta \in A, I \) is an answer set of \( B \cup \bar{R}_M^\circ \cup \Delta \) iff \( I \setminus \Delta \) is an answer set for \( B \cup hyp(\Delta) \). Let us consider an optimal solution \( \Delta' \), \( l(E, B, \Delta') = l(E, B, hyp(\Delta')) \) since they have the same answer sets and \( \sum_{r \in hyp(\Delta')} |r| = \sum_{r \in H} |r| \) (see Section 3.2). Suppose there is another \( H'' \) with \( l(E, B, hyp(\Delta')) \succ l(E, B, H'') \). Then in this case there would be a \( \Delta'' \) such that \( l(E, B, hyp(\Delta'')) \succ l(E, B, hyp(\Delta')) \) and \( \Delta' \) would not be the optimal solution. Dually if we consider an optimal solution \( H' \) there is only one \( \Delta' \) such that \( H' = hyp(\Delta') \) and \( l(E, B, \Delta') = l(H') \). If there was a \( \Delta'' \) with \( l(E, B, \Delta') \succ l(E, B, \Delta'') \) then \( hyp(\Delta'') \) would be the optimal solution.

We can use the above result to map an original ILP problem into a problem that can be solved in ASP, using standard techniques supported by most solvers. The new task can be seen as an abductive reasoning task, since the hypotheses are represented by sets of logic facts, similarly to Chapter 4. Also in this case we are using a flat representation of the hypothesis space where each rule is codified by an abducible. Differently from TAL the abductive search is delegated to the ASP solver and the integrated search based on SAT solving techniques.

Considering that \( M \) is used and that constants are replaced by variables in rules in \( \bar{R}_M^\circ \), the \( id \) function can be defined as follows: \( id(r) = (m_h, m_1, l_{1,1}, ..., l_{1,p_1}, ..., m_n, l_{n,1}, ..., l_{n,p_n}) \) where \( rt(r) = [(m_h, c_h), (l_1, c_1, [l_{1,1}, ..., l_{1,p_1}]), ..., (l_n, c_n, [l_{n,1}, ..., l_{n,p_n}])] \).

### 6.2 Clause space generation

In the previous section we have shown how to translate an ILP task into an ASP abduction task (or equivalently, in our case, into a simplified ILP task that can be solved using ASP solvers) that consists of deriving only atomic rules. The underlying assumption is that the space of inductive solutions is given as a set of ordered rules. Although mode declarations indirectly define a set of compatible (non-ordered) rules, in the declarative approach presented here we use mode declarations to generate candidate rules but we need to avoid redundancies in the final solution space. For example, consider the ordered rule \( p(X) \leftarrow q(X), r(X) \). There is another ordered rule that is semantically equivalent, obtained simply by inverting the order of the last two elements in...
the list. Systems based on inverse entailment already rely on the implicit order of the conditions in the rule(s) of the most specific hypothesis. But in our ASP representation of an ILP problem we need to address this issue. In order to do so we must first characterise equivalence classes over compatible ordered rules.

For a given compatible ordered rules, we might be tempted to define a subset of semantically equivalent compatible ordered rules by simply commuting body conditions. Unfortunately only some of the resulting rules would still be compatible because of condition (iii) in Definition 3.7, which imposes an ordering over conditions that share variables. In order to overcome this, we define a subset of \( R^o \) (we omit the implicit set of mode declarations \( M \) in the following), in which rules have conditions that satisfy a given total order. The **prioritised composition** [Chomicki, 2003] \( \triangleright \) of two order relations \( \succ_1 \) and \( \succ_2 \) is defined as

\[
a(\succ_1 \triangleright \succ_2) b \text{ iff } a \succ_1 b \lor (a \succ_1 b \land a \succ_2 b)
\]

where \( \sim_x \) is the indifference relation for \( \succ_x \) \( (a \sim_x b \text{ iff } a \not\succ_x b \text{ and } b \not\succ_x a) \).

**Definition 6.2** Let \( f_1 \) and \( f_2 \) be two body conditions in an ordered rule. Let \( \succ_p \) be the partial order where \( f_1 \succ_p f_2 \) iff there is an input variable in \( f_2 \) that is bound to an output variable in \( f_1 \). Let \( \succ_t \) be an arbitrary total order. We define a total order \( (\succ_p \triangleright \succ_t) \equiv \succ_r \) and denote the set of ordered rules such that each pair of conditions in the body respect \( \succ_r \) as \( R^r \subseteq R^o \).

The rationale behind the previous definition is that mode declarations instantiate an implicit partial order \( \succ_p \) on the body conditions in each rule within \( R^o \). Within this partial order we have the freedom to define a linear extension and derive a total order.

**Example 6.2** Let \( M \) be the set of mode declarations from Example 3.1 and \( r \) the ordered rule \( p(X^+) \leftarrow q(X^+, Y^-), q(Y^+, a), q(Y^+, Z^-), q(Z^+, b) \). In this case, \( q(X^+, Y^-) \prec_p q(Y^+, a), q(X^+, Y^-) \prec_p q(Y^+, Z^-), q(Y^+, Z^-) \prec_p q(Z^+, b) \).

It is possible to define three linear extensions over this partial order including the total order given by the ordering with which literals appears in \( r \). On the contrary, it is impossible that \( q(Y^+, a) \prec_r q(X^+, Y^-) \) since the former literal has an input variable that is bound to an output variable of the latter.

Given the order on the conditions we want to reason about the equivalence of ordered rules. For clarity, we recall that ordered rules are represented by lists of literals.

**Definition 6.3** Let \( r_1 \) and \( r_2 \) be two rules in \( R^o \). We say \( r_1 \) and \( r_2 \) are equivalent \( (r_1 \approx r_2) \) iff \( r_1 \) and \( r_2 \) have the same first element (the head) and there exists a permutation such that the list of the elements 2, ..., \( n \) (the body) in \( r_1 \) is equal to the list of elements 2, ..., \( n \) in \( r_2 \) up to variable renaming.
In other words the two rules are equivalent if they have the same head and have the same elements in the body in a different order.

**Proposition 6.3** For all \( r \in R^o \), there is a \( r' \in R^r \) such that \( r \approx r' \).

**Proof** We recall that rules in the set \( R^o \) are compatible with a given set of mode declarations. Thus, conditions in \( r \) respect the partial order \( \prec_p \) induced by the mode declarations. Conditions can always be ordered according to \( \prec_r \) (because it is always possible to define a linear extension) thus obtaining a rule \( r' \) such that \( r \approx r' \).

**Proposition 6.4** Let \( r, r' \in R^r \) such that \( r \neq r' \). Then \( r \neq r' \).

**Proof** Let us consider two distinct \( r, r' \in R^r \), and suppose \( r \approx r' \). The two rules must have the same literals. But in this case the same order is instantiated on them producing exactly the same body. This contradicts the initial assumption.

Given Propositions 6.3 and 6.4, we can say each element \( r \) in \( R^r \) characterises an equivalence class \( [r] = \{ r' \in R^o : r' \approx r \} \). The mapping \( R^o \mapsto R^r \) is thus a canonical projection. In other words, all rules within \( R^o \) can be represented by an equivalent element in \( R^r \). We can use the new set to remove the redundancy caused by ordered rules without missing any of the compatible rules and considering only one total ordering for each compatible rule. In order to derive a concrete implementation, we choose as order \( \succ_t \) the lexicographic order over an internal representation of the conditions. Also, since rules in \( R \) have an equivalent ordered rule in \( R^o \) (by Definition), given Propositions 6.3 and 6.4, each rule in \( R \) has only one equivalent rule in \( R^r \). This result makes it possible to use the rules in \( R^r \) in the ASP encoding of a given ILP problem without loss of completeness.

### 6.2.1 Skeleton rules

Having characterised the subset of ordered rules that match the space of (non-ordered) rules compatible with a set of mode declarations, and a way to map an ILP problem into an equivalent problem that can be solved by an ASP solver, we can put the two things together by characterising the set of ordered rules that we need to generate. Given an ILP task \( \langle E, B, R^o_M \rangle \), by Propositions 6.3 and 6.4, we can equivalently consider the task \( \langle E, B, R^r_M \rangle \). According to Theorem 6.2, we can use ASP to solve the equivalent task \( \langle E, B \cup \tilde{R}^r_M, A_M \rangle \). We call the set of rules in \( \tilde{R}^r_M \) *skeleton rules* and in order to simplify the notation, we denote the set as \( S_M \).
Example 6.3  Consider the following mode declarations

\[
M = \begin{cases} 
    m1 : \text{mode}(p(+\text{any})) \\
    m2 : \text{mode}(q(+\text{any}, \#\text{any})) \\
    m3 : \text{mode}(r(+\text{any}, -\text{any})) 
\end{cases}
\]

The skeleton rules with maximum one condition for \( M \) are:

\[
S_M = \begin{cases} 
    p(X) \leftarrow \text{rule}((m1), []) \\
    p(X) \leftarrow q(X, C), \text{rule}((m1, m2, 1), [C]) \\
    p(X) \leftarrow r(X, R), \text{rule}((m1, m3, 1), []) 
\end{cases}
\] (6.1)

Skeleton rules are used in the system as a preliminary stage towards final rules. They simplify the construction of a theory that represents the space of admissible rules and delegate the burden of the grounding of the constants (that is based on the particular theory \( B \)) to efficient off-the-shelf grounders. Before we delve into the algorithm used to generate these rules, we want to study what factors influence the number of skeleton rules. ASP solvers always go through an initial grounding phase, where the input program is transformed into an equivalent ground program. Despite the fact that techniques to speed up the computation are employed, this is often the bottleneck of the whole process (see also Table 6.1). Thus it is important to characterise what affects the time spent grounding the theory. In order to derive an effective strategy we want to study how the given inputs affect the number of skeleton rules defined by a set of mode declarations. We use \(|\text{out}(m)|, |\text{inp}(m)|, |\text{con}(m)|\) to denote respectively the number of output, input and constant placemarkers, in a mode declaration \( m \). The number of possible heads \( n_h \) for a skeleton rule is the number of head mode declarations \(|M_h| = n_h\), since each placemarker is substituted by a different variable with no degree of freedom. The number of possible instantiations of a body mode declaration within a rule \( n_b \) depends on the number of available variables that can be bound to input variables (namely the set of variables that appear as output variables in preceding conditions or as input variables in the head), denoted as \( v \) in the following:

\[
n_b(v) = \sum_{m \in M_b} v^{\text{inp}(m)} \leq |M_b|v^{\max\{|\text{inp}(m)| : m \in M_b\}}
\]

Let \( d \) be a limit on the number of conditions or depth of a rule. To simplify the notation we denote \( \max\{|\text{inp}(m)| : m \in M_b\} = \max_i \) and \( \max\{|\text{out}(m)| : m \in M_b\} = \max_o \). An upper bound for the number of variables given as input to a condition is the maximum number of outputs in a condition multiplied by the maximum number of conditions: \( \max_o \times d \). An upper bound for the number of skeleton rules \(|S_M|\) can
be derived considering for each place in the body all possible conditions:

$$|S_M| \leq |M_h| \times (|M_b| \times (max_o \times d)^{max_i})^d$$

This upper bound confirms the intuitive idea that the number of input placemarkers and the maximum number of conditions are the factors that most affect the hypothesis complexity. A key guideline that can be extracted from this upper bound is that problems with less depth are easier to solve. The ASPAL system therefore iterates over the maximum depth of rules in order to avoid the generation of superfluous skeleton rules and avoid costly groundings.

**Example 6.4** Consider the following set of mode declarations:

$$M = \begin{cases} 
\text{modeh(even}(+\text{nat})). \\
\text{modeh(odd}(+\text{nat})). \\
\text{modeb(even}(+\text{nat})). \\
\text{modeb(odd}(+\text{nat})). \\
\text{modeb(\text{nat} = s(}-\text{nat}))). 
\end{cases}$$

In this case $|M_h| = 2$, $|M_b| = 3$, $max_o = 1$ and $max_i = 1$. Assuming a maximum depth of $d = 2$ we expect $N_r \leq 2 \times 36$ skeleton rules. The precise number can be derived given that the number of possible heads is 2 (thus 2 is also the number of atomic rules) and the number of bodies that can be constructed is 12 (3 with a single condition and 10 with two conditions, avoiding repeated conditions, plus the empty body). So the actual number of skeleton rules in this case is $14 \times 2 + 2 = 30$.

### 6.3 Algorithm

The overall algorithm resembles the one presented in Chapter 5 for TAL. The main difference is that, given the considerations about the size of the rule space, whenever possible the algorithm exploits an iterative deepening approach over the depth of the rules. The learning process starts by generating the skeleton theory $S_M$ (and the associated set of atoms $A_M$), up to a certain depth $d$. After this the burden of the search for the final hypotheses is shifted onto the ASP solver. Algorithm 6 describes the overall procedure.

By default the system finds suboptimal solutions: $d$ is increased by one at each iteration and the cycle terminates whenever at least one solution is found. In this case the system finds all optimal solutions within a certain depth. If all the optimal solutions are required, after each iteration a target value for the loss function is determined in order to establish whether better solutions can be found. For example, suppose we are using
Algorithm 6 FIND-HYPOTHESIS

Inputs: E examples; B background theory; M mode declarations; l loss function

Outputs: H hypotheses

1: $d = 0$
2: $H = \emptyset$
3: while a termination condition does not hold do
4:   $S_M, A_M = \text{DERIVE-SKELETON-THEORY}(M, d)$
5:   $\{\Delta_1, \ldots, \Delta_n\} = \text{ASPAL}(E, S_M \cup B, A_M, l)$
6:   $H_{\text{new}} = \text{TRANSLATE-SOLUTIONS}(\{\Delta_1, \ldots, \Delta_n\}, S_M)$
7:   $H = \langle \text{select optimal or add all solutions from } H \cup H_{\text{new}} \rangle$
8:   increase $d$
9: end while
10: return $H$

the complexity loss function. If a solution with complexity 2 has been found in an iteration where $d = 1$, we know that the iteration with $d = 2$ will not find a better solution since the new solutions explored will have at least complexity 3 (the head plus the two conditions). The system in general can be run to find a (sub)optimal solution, all the (sub)optimal solutions and all the solutions within the given boundaries on the size of the hypothesis ignoring the loss function.

6.3.1 Derivation of skeleton rules

The set of abductive skeleton theories is constructed using a dynamic programming algorithm [Skiena, 2008]. The set of skeleton rules of depth 0 is given by the variabilised head mode declarations. The set of skeleton rules of depth $n$ is given by the set of skeleton rules of depth $n-1$, denoted $S^{n-1}_M$, plus the skeleton rules $h \leftarrow B, b, c$ such that there exists a rule $h \leftarrow B, b \in S^{n-1}_M$ and $b \prec_r c$.

The DERIVE-SKELETON-THEORY procedure is detailed in Algorithm 7. All the rules of depth 0 are created as $s^*$ by substituting input and output placemarkers in a schema $s$ with variables (line 4) and each of these is extended tail-recursively and the set $A$ is initialised with one atom for each of these rules. Though not reported here for readability, the types in the mode declarations are used to instantiate these atoms in the optimisation statements used by the ASP solver. Rules are extended with conditions by the procedure EXTEND-WITH-CONDITIONS. The final rules are completed by the ADD-CONDITION procedure with the rule abducible that contains the unique identifier and the list of variables as previously described.

The EXTEND-WITH-CONDITIONS procedure recursively adds conditions to the input rule. The given partial rule is extended (if allowed by the ordering constraint) by the ADD-CONDITION procedure. All the new rules are further extended by a recursive call. When the depth limits are reached or the ordering constraints do not allow new conditions to be added, the returned set of new rules is empty and the procedure returns only the input rule without further recursive calls.
Algorithm 7 DERIVE-SKELETON-THEORY

Inputs: $M$ mode declarations, $d$ maximum number of conditions

Outputs: $S$ set of skeleton rules, $A$ set of atoms

1: $S = \emptyset$
2: $A = \emptyset$
3: for all $m(s) \in M_h$ do
4:     $r = s^*$
5:     $a = \text{rule}(\text{id}(s^*), \text{con}(s^*))$
6:     $S_{\text{new}}, A_{\text{new}} = \text{EXTEND-WITH-CONDITIONS}(r, a, M, d)$
7:     $S = S \cup S_{\text{new}}$
8:     $A = A \cup A_{\text{new}}$
9: end for
10: $S = \text{ADD-CONDITION}(S)$
11: return $S, A$

Algorithm 8 EXTEND-WITH-CONDITIONS

Inputs: $r$ partially constructed rule, $a$ atom associated to the rule, $M$ mode declarations, $d$ maximum number of conditions

Outputs: $S_{\text{new}}, A_{\text{new}}$ rules obtained adding conditions to the input and corresponding abducibles

1: $S_{\text{new}} = \{r\}$
2: $A_{\text{new}} = \{a\}$
3: for all $m \in M$ do
4:     if $|r| < d$ then
5:         $SA = \text{ADD-CONDITION}(r, m)$
6:             for all $(q, b) \in SA$ do
7:                 $S_q, A_q = \text{EXTEND-WITH-CONDITIONS}(q, b, M)$
8:                 $S_{\text{new}} = S_{\text{new}} \cup S_q$
9:                 $A_{\text{new}} = A_{\text{new}} \cup A_q$
10:             end for
11:         end if
12: end for
13: return $S_{\text{new}}, A_{\text{new}}$
The `ADD-CONDITION` procedure takes into account the ordering and evaluates all the possible instantiations of a mode declaration into a skeleton rule condition. First all the possible bindings of the input variables are generated (Algorithm 9, lines 6 to 15). More specifically, for all input variables \(i\), with index \(j\), all the possible bindings with the accumulated variables are added to the set of equalities \(conditions[j]\) (line 10). Then the Cartesian product for all the bindings for all the variables is calculated. If the condition respects the ordering (line 20) a new rule is created for each of these bindings. Note that the procedure `ORDER-CHECK` returns `false` if the ordering is not respected, `true` if it is respected regardless of the instantiation of the constants and a set of inequalities if the ordering depends on the instantiation of the constants. The algorithm is further explained in Example 6.5.

```
Algorithm 9 ADD-CONDITION

Inputs: rule partially constructed rule, \(m\) body mode declaration
Outputs: rules set of rules obtained adding conditions derived from \(m\) to the input rule

1: \(variables = <\) input variables in the head or output variables in the conditions of \(rule >\)
2: \(ivariables = <\) input variables in \(m^*\)
3: \(j = 0\)
4: \(out\_rules = \emptyset\)
5: \(out\_abd = \emptyset\)
6: \(\text{for all } i \in ivariables \text{ do}\)
7: \(\text{for all } v \in variables \text{ do}\)
8: \(conditions[j] = \emptyset\)
9: \(\text{if } <i \text{ and } v \text{ are of the same type } > \text{ then}\)
10: \(conditions[j] = conditions[j] \cup \{i = v\}\)
11: \(\text{end if}\)
12: \(j = j + 1\)
13: \(\text{end for}\)
14: \(\text{end for}\)
15: \(bindings = conditions[0] \times ... \times conditions[length(ivariables) - 1]\)
16: \(\text{for all } b \in bindings \text{ do}\)
17: \(nrule = <\text{copy of rule } >\)
18: \(lc = <\text{last condition in rule } >\)
19: \(order\_conditions = ORDER-CHECK(lc, m^*)\)
20: \(\text{if } order\_conditions \neq false \text{ then}\)
21: \(<\text{add the literals } m^* \text{ and } b \text{ to } nrule } >\)
22: \(a = <\text{abducible corresponding to } nrule, \text{ including } order\_conditions >\)
23: \(out\_rules = out\_rules \cup nrule\)
24: \(out\_abd = out\_abd \cup a\)
25: \(\text{end if}\)
26: \(\text{end for}\)
27: \(\text{return } out\_rules, out\_abd\)
```
Example 6.5 Consider the following set of mode declarations:

\[
M = \begin{cases}
  m1 : modeh(penguin(+bird)) \\
  m2 : modeb(can(+bird, #ability)) \\
  m3 : modeb(looks_like(+bird, −bird))
\end{cases}
\]

The abductive skeleton theory \( S^0_M \) for rules of depth 0 is constructed using the variabilised schema of the only head mode declaration:

\[
S^0_M = \begin{cases}
penguin(A) : - \\
  rule(m1, []). 
\end{cases}
\]

\[
A^0_M = \begin{cases}
  rule(m1, []). 
\end{cases}
\]

The set of rules of depth 1 is constructed by executing the EXTEND-WITH-CONDITIONS procedure on the rule in \( S^0_M \) and all the body mode declarations.

\[
S^1_M = \begin{cases}
penguin(A) : - \\
  looks_like(B, C), A = B, \\
  rule((m1, m3, 1), []). \\

penguin(A) : - \\
  can(B, C), A = B, \\
  rule((m1, m2, 1), [C]).
\end{cases}
\]

\[
A^1_M = \begin{cases}
  rule((m1, m3, 1), []). \\
  rule((m1, m2, 1), [C]).
\end{cases}
\]

Similarly for rules of depth 2 the EXTEND-WITH-CONDITIONS procedure is executed with the two rules in \( S^1_M \) and all the body mode declarations as arguments. The first two rules from top to bottom in the set \( S^2_M \) are obtained by adding the conditions that can be constructed from the mode declaration \( m3 \). The input variable can be bound to the input variable in the head and the output variable in the first condition, thus two rules are produced.
The abductive skeleton theory with maximum depth 2 is $S_M = S_M^0 \cup S_M^1 \cup S_M^2$.

Note that the order of instantiation of the skeleton rules must be further refined in the third rule. This is encoded as a constraint on the set of abducibles:

$$A_M = \{ \text{rule}(m1, [], []), \text{rule}(m1, m3, 1, []), \text{rule}(m1, m3, 1, m3, 1, []), \text{rule}(m1, m3, 1, m3, 2, []) \} \cup \{ \text{rule}(m1, m2, 1, [C]): \text{ability}(C) \} \cup \{ \text{rule}(m1, m3, 1, m2, 1, [E]): \text{ability}(E) \} \cup \{ \text{rule}(m1, m2, 1, m2, 1, [C, E]): \text{ability}(C), \text{ability}(E), C < E \}$$

The set of abducibles is instantiated during the grounding phase of the ASP solving process. Supposing that the background knowledge contains the following facts that define ability/1: $\{ \text{ability(swim)} \}$ $\cup$ $\{ \text{ability(fly)} \}$ the set is grounded as:

$$A_M = \{ \text{rule}(m1, [], []), \text{rule}(m1, m3, 1, []), \text{rule}(m1, m3, 1, m3, 1, []), \text{rule}(m1, m3, 1, m3, 2, []) \} \cup \{ \text{rule}(m1, m2, 1, [fly]), \text{rule}(m1, m2, 1, [swim]) \} \cup \{ \text{rule}(m1, m3, 1, m2, 1, [swim]), \text{rule}(m1, m3, 1, m2, 1, [fly]) \} \cup \{ \text{rule}(m1, m3, 1, m2, 2, [swim]), \text{rule}(m1, m3, 1, m2, 2, [fly]) \} \cup \ldots$$
Given the skeleton theory \( S_M \), the procedure \textsc{ASPal} is called to generate all the answer sets. The procedure invokes the ASP solver on the theory \( B \cup S_M \) with an additional optimisation statement [Gebser et al., 2011] produced on the set \( A_M \) and on the examples, based on the particular loss function that is used\(^1\). We can explicitly state the set \( A_M = \{a_1, ..., a_n\} \) making use of aggregates [Gebser et al., 2011] as follows in \texttt{clingo}, the solver used in \textsc{ASPal}:

\[
0 \{a_1, ..., a_n\} \max_a.
\]

where \( \max_a \) defines the maximum number of elements in \( A \) that can be true in an answer set, and consequently the maximum number of rules in the final inductive hypothesis. This amounts to performing abduction on the set of abducibles \( \{a_1, ..., a_n\} \), where the final solution is constrained to have at most \( \max_a \) elements.

The final step, executed by the procedure \textsc{TRANSLATE-SOLUTIONS}, extracts the abducibles from the generated answer sets and produces an inductive hypothesis by appropriately substituting the constants in the skeleton rules with those appearing in the abducibles.

**Example 6.6** Consider example 6.5 and the following background knowledge and examples:

\[
B = \begin{cases}
  \text{bird}(a). \text{bird}(b). \\
  \text{can}(a, \text{fly}). \text{can}(b, \text{swim}). \\
  \text{ability}(\text{fly}). \text{ability}(\text{swim}). \\
  \text{looks_like}(a, b). \\
  \text{looks_like}(X, Y) \leftarrow \text{looks_like}(Y, X).
\end{cases}
\]

\[
E = \begin{cases}
  \text{penguin}(b). \\
  \text{not penguin}(a).
\end{cases}
\]

Figure 6.1 shows the theory generated after the pre-processing that is given as input to the ASP solver. The ASP solver returns three answer sets whose projections on \( A_S \) are: \( \Delta_1 = \{\text{rule}((m_1, m_2, 1), [\text{swim}])\} \), \( \Delta_2 = \{\text{rule}((m_1, m_3, 1, m_2), [\text{swim}])\} \), \( \Delta_3 = \{\text{rule}((m_1, m_3, 1, m_2, 1, 2), [\text{fly}])\} \). The abductive solutions correspond to the following three inductive solutions:

\(^1\)Optimisation statements are used to derive optimal answer sets based on statements of the type \( \text{minimize}\{a_1 = w(a_1), ..., a_n = w(a_n)\} \). An answer set is optimal if the sum of the weights \( w(a_i) \) is minimal amongst all the answer sets of the given program. In \textsc{ASPal} weights are assigned to \texttt{rule} atoms, based on the literals in the rule being represented, and to examples whenever noise is tolerated on them.
\[ H_1 = \begin{cases} \text{penguin}(A) \leftarrow \text{can}(A, \text{swim}). \\ \end{cases} \]

\[ H_2 = \begin{cases} \text{penguin}(A) \leftarrow \text{looks_like}(A, B), \\ \text{can}(A, \text{swim}). \\ \end{cases} \]

\[ H_3 = \begin{cases} \text{penguin}(A) \leftarrow \text{looks_like}(A, B), \\ \text{can}(A, \text{swim}). \\ \end{cases} \]

\[ 6.4 \text{ Experiments} \]

We collected some problems that require a nonmonotonic ILP system from existing work. The problems \text{oblasc,ctas,mobile,tuner} and \text{ws} are based on the methodology described in [Alrajeh et al., 2009], and in particular the last two are discussed respectively in [Alrajeh et al., 2011] and [Uchitel et al., 2009]. In these problems, formal requirements are extracted from scenario-based descriptions, based on a logic translation of the given specifications. The problems \text{lact} [Ray, 2009a] and \text{meta} [Ray et al., 2010a] are applications to biology. \text{norm} [Corapi et al., 2011a] applies nonmonotonic ILP to revise an ASP theory encoding a normative framework in order to match a set of given use cases that model the intended normative behaviour.

\text{ASPAL} is executed to find one optimal solution and all the suboptimal solutions within the minimum depth for which a solution exists according to the function \( t_{\text{complexity}} \) as required in the respective papers. All the problems have a similar structure with a temporal component. Examples model instances of some expected behaviour of the system described and integrity constraints play a similar role to examples, defining hard constraints on acceptable models. It is worth remarking that these integrity constraints are of crucial importance. Other ILP systems do not allow specification of these integrity constraints, whereas those that do allow it (including \text{TAL}) are affected in the computational time. This is because, as the example shows, integrity constraints are used to drive the derivation of the hypotheses but they are not necessarily ground. ASP solvers take them into account and discard models that are not in accordance with them.

In other experiments performed we confirmed the intuition that the total computation time depends on the maximum depth of the rules and not on the maximum number of rules in a hypothesis, as conjectured in Section 6.2.1. Despite the complexity of the problems at hand, solutions are usually found within a small depth and consequently the computation time for the problems in Table 6.1 is relatively low. In fact the grounding time has a dominant share in the total computation time and it is affected by the maximum depth of the hypotheses.
bird(a).
bird(b).
can(a,fly).
can(b,swim).
ability(fly).
ability(swim).
looks_like(a,b).
looks_like(X,Y):-bird(X),bird(Y),looks_like(Y,X).
:- not goal.
goal :- penguin(b),not penguin(a).
penguin(A):-
  bird(A),
  rule(r(idpeqm,c,l),1).
penguin(A):-
  bird(A), A=D, bird(D), ability(E), can(D,E),
  rule(r(idpeqm,c,l),(idcaQg,c(E),l(1))),2).
penguin(A):-
  bird(A), A=D, bird(D), ability(E), can(D,E),
  A=F, bird(F), ability(G), can(F,G),
  rule(r(idpeqm,c,l),(idcaQg,c(E),l(1)),(idcaQg,c(G),l(1))),3).
penguin(A):-
  bird(A), A=B, bird(B), bird(C),
  looks_like(B,C),
  rule(r(idpeqm,c,l),(idloD2B,c,l(1))),2).
penguin(A):-
  bird(A), A=B, bird(B), bird(C), looks_like(B,C), A=F, bird(F), ability(G), can(F,G),
  rule(r(idpeqm,c,l),(idloD2B,c,l(1)),(idcaQg,c(G),l(1))),3).
penguin(A):-
  bird(A), A=B, bird(B), bird(C), looks_like(B,C), C=F, bird(F), ability(G), can(F,G),
  rule(r(idpeqm,c,l),(idloD2B,c,l(1)),(idcaQg,c(G),l(2))),3).
penguin(A):-
  bird(A), A=B, bird(B), bird(C), looks_like(B,C), A=D, bird(D), bird(E), looks_like(D,E),
  rule(r(idpeqm,c,l),(idloD2B,c,l(1)),(idloD2B,c,l(1))),3).
penguin(A):-
  bird(A), A=B, bird(B), bird(C), looks_like(B,C), C=D, bird(D), bird(E), looks_like(D,E),
  rule(r(idpeqm,c,l),(idloD2B,c,l(1)),(idloD2B,c,l(2))),3).

0 {rule(r(idpeqm,c,l),1),
rule(r(idpeqm,c,l),(idcaQg,c(E),l(1))),2):ability(E),
rule(r(idpeqm,c,l),(idcaQg,c(E),l(1)),(idcaQg,c(G),l(1))),3):ability(E):ability(G):c(E)<c(G),
rule(r(idpeqm,c,l),(idloD2B,c,l(1))),2),
rule(r(idpeqm,c,l),(idloD2B,c,l(1)),(idcaQg,c(G),l(1))),3):ability(G),
rule(r(idpeqm,c,l),(idloD2B,c,l(1)),(idcaQg,c(G),l(2))),3):ability(G),
rule(r(idpeqm,c,l),(idloD2B,c,l(1)),(idloD2B,c,l(1))),3),
rule(r(idpeqm,c,l),(idloD2B,c,l(1)),(idloD2B,c,l(2))),3)} 1.

Figure 6.1: ASP theory generated for Example 6.6. The theory is generated by a Python implementation of the transformation described in Section 6.3. The rules slightly differ from those used in the examples since a simplified notation was used in those.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>oblas</td>
<td>38</td>
<td>5</td>
<td>100</td>
<td>2</td>
<td>9</td>
<td>2.27</td>
<td>1</td>
<td>1.49</td>
<td>18</td>
<td>4.35</td>
<td>15</td>
</tr>
<tr>
<td>ctas</td>
<td>66</td>
<td>12</td>
<td>56</td>
<td>1</td>
<td>13</td>
<td>0.27</td>
<td>6</td>
<td>0.19</td>
<td>5</td>
<td>0.28</td>
<td>1</td>
</tr>
<tr>
<td>mobile</td>
<td>66</td>
<td>13</td>
<td>49</td>
<td>1</td>
<td>6</td>
<td>0.47</td>
<td>2</td>
<td>0.4</td>
<td>51</td>
<td>0.51</td>
<td>1</td>
</tr>
<tr>
<td>lact</td>
<td>2</td>
<td>1</td>
<td>19</td>
<td>3</td>
<td>3</td>
<td>0.01</td>
<td>2</td>
<td>0.01</td>
<td>10</td>
<td>0.01</td>
<td>1</td>
</tr>
<tr>
<td>tuner</td>
<td>19</td>
<td>5</td>
<td>67</td>
<td>4</td>
<td>4</td>
<td>0.37</td>
<td>2</td>
<td>0.22</td>
<td>8</td>
<td>1.63</td>
<td>1</td>
</tr>
<tr>
<td>ws</td>
<td>27</td>
<td>5</td>
<td>45</td>
<td>8</td>
<td>4</td>
<td>0.05</td>
<td>2</td>
<td>0.04</td>
<td>8</td>
<td>0.13</td>
<td>9</td>
</tr>
<tr>
<td>norm</td>
<td>57</td>
<td>0</td>
<td>44</td>
<td>7</td>
<td>16</td>
<td>0.12</td>
<td>5</td>
<td>0.09</td>
<td>56</td>
<td>0.09</td>
<td>12</td>
</tr>
<tr>
<td>meta</td>
<td>7</td>
<td>2</td>
<td>21</td>
<td>2</td>
<td>2</td>
<td>0.01</td>
<td>2</td>
<td>0.01</td>
<td>2</td>
<td>0.01</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 6.1: Computation time and other information on the experiments. From left to right: the number of rules (with more than one literal), integrity constraints, facts, examples and mode declarations; the time to derive an optimal solution and the number of rules in such solution; the grounding time; the number of skeleton rules derived; the computation time for the derivation of all the suboptimal rules with minimal depth and the number of such suboptimal solutions.

but not by the maximum number of rules, which affects only the actual solving time.

Given the novelty of the research area it was not possible to compare the results obtained to those of other nonmonotonic ILP system. As discussed in Section 1.1 and in Chapter 3, XHAIL the only nonmonotonic system that supports a very similar class of problems to those supported by ASPAL is not publicly available. A comparison with traditional ILP systems was also impossible since they support a different class of languages that could not be used to represent the formulation of the problems used in the experiments.

We consider TAL and ASPAL alternative solutions for different classes of problems. In fact, for the problems considered here TAL takes a time that is one to two orders of magnitude higher. Dually, if we consider the problems in Section 5.7, ASPAL exhibits a much higher computation time, in particular for the grounding phase, due to the large domains considered. Although we are far from providing a final word on the matter, we observed in our experiment that a greater use of negation and integrity constraints increases the size of the derivation tree for SLD-based systems like TAL. This was one of the driving motivations behind the design and development of ASPAL.

### 6.5 Related work

The integration of ASP and ILP has been addressed by Sakama in [Sakama, 2001a] and more extensively in [Sakama, 2005]. The author presents a characterisation of the problem and a number of theoretical results on the conditions under which derived rules imply literals. The results are used as foundations for a concrete algorithm. However, the algorithm considers only one example at a time. This causes the order in which the examples are considered to be crucial for the final outcome. The completeness and soundness results are given with respect to a single example.
Otero in [Otero, 2001] characterises the conditions for the existence of inductive hypotheses in the nonmonotonic ILP setting. The work addresses a different setting that consider multiple sets of examples, not considered here. However the author does not describe a concrete implementation and it does not address the computation of the hypotheses. The same author proposes in [Otero, 2003] a method to learn temporal action theories based on a conversion of a problem that would require nonmonotonic ILP into a formulation where a monotonic ILP approach can be used. As discussed more in detail in [Ray, 2009a], this approach can only be applied in a limited setting and is not a general purpose solution to nonmonotonic ILP problems.

**XHAIL** [Ray, 2009a] is a nonmonotonic ILP system based on the earlier procedure for definite programs described in [Ray et al., 2004]. XHAIL is able to solve the same class of problems as ASPAL and one of its implementations relies on ASP as a representation language and for the implementation of the search and the inference. It is interesting to notice the parallel between the iterative deepening approaches of the two systems. XHAIL iterates over the size of the abductive hypothesis computed in its first phase, which is used to construct the head of the rules. If an abductive solution that is extended into a kernel set (a most specific theory that is generalised into a final hypothesis) does not produce a valid hypothesis the system produces an alternative solution. This process does not involve only minimal solutions, but a number of abductive solutions that is only limited by the size of the Herbrand base of the program, regardless of the complexity of the hypotheses. Thus, as explained in [Corapi et al., 2010], this may cause a large number of abductive solutions to be derived in the first phase before the desired solution is found. Moreover for each of these solutions, the whole process of deriving a kernel set and generalising is performed. In ASPAL the three phases are all integrated in one and a solution is guaranteed to be found within the boundaries of the maximum depth. Together with the aforementioned issue, the main contributions of this work with respect to XHAIL are 1) an empirical evaluation of the approach based on existing problems; 2) a more flexible control over the optimality criterion; 3) the use of a methodology that is not based on inverse entailment.

Approaches to ILP based on *propositionalisation* [Krogel & Wrobel, 2002] construct a propositional theory as a preliminary step, often adopting transformations that do not preserve the original semantics but approximate it [Raedt, 2008]. Thus similarly to ASPAL, inference and search are performed on a ground theory and the results are translated back into a relational representation. Differently from ASPAL propositional algorithms are applied, rather than performing abduction on a transformed theory that includes structural meta-predicates. Propositionalisation also enables the use of traditional learning algorithms such as support vector machines or neural networks [Garcez et al., 2007]. As a notable example, LINUS [Lavrac & Dzeroski, 1994] transforms the training examples into attribute-value tuples. The input theory is constrained to non-recursive, definite clauses thus, unlike ASPAL, it addresses only monotonic problems.
6.6 Conclusion

We have presented an integrated approach to ILP based on ASP. The approach is based on a preliminary construction of so-called skeleton rules. The original problem is translated into an equivalent problem where the search and the grounding are delegated to an ASP solver. In contrast to methods based on inverse entailment, the intermediate theory generated serves as a generality upper bound for the final hypothesis and is instantiated by defining the constants that appear in the rule and by choosing the subset of skeleton rules that contribute to the solution. The number of skeleton rules grows exponentially with the number of allowed conditions, but for many significant applications, compressive rules (thus with shorter conditions) are preferred. We tackle this complexity issue by using an iterative deepening approach on the number of conditions. To the best of our knowledge this is the first nonmonotonic ILP system that has been empirically evaluated on a set of problems discussed in the literature.

The approach presented is very general and can be easily extended to deal with disjunctive clauses and the use of classical negation. We leave these extensions for future work together with the presentation of extensions for approximate solution of larger problems.
7 Theory Revision as Nonmonotonic ILP

7.1 Introduction

The area of Theory Revision (TR) is regarded as a generalisation of ILP, where not only new hypotheses can be learned but also existing clauses can be revised based on the assumption that they are not necessarily correct. TR is particularly useful when an initial, roughly correct theory is available and we want to improve it automatically or semi-automatically according to examples acquired incrementally over time. It is also a way to keep a theory, modelling a volatile domain, up-to-date with recent examples. TR can be seen as a two-sided task. When evidence contradicts the current theory a revision is needed. However, just updating the theory with factual evidence would simply amount to recording facts. Just as humans do not merely commit lists of facts to memory but find useful generalisations in order to reuse experience, so TR needs to generalise from observed facts.

TR is a particular case of the problem known as theory refinement, “the problem of improving the quality of a given theory” [Wrobel, 1996]. In general, a theory can be either restructured, i.e. its semantics does not change but it is only modified for efficiency, elegance or understanding, or revised, i.e. the semantics is modified to derive different conclusions.

Much of the existing body of work in this area concentrates on approaches that search for improvements of a given theory by applying operators to the rules. For example the expected gain in accuracy with respect to a given set of examples is calculated for possible alterations like the deletion or the addition of a condition in a rule. TR systems usually perform a greedy search that evaluates the effect of applying a change with respect to a set of examples. The change that causes the best improvement is implemented and the resulting theory is revised again. Though this approach is efficient, in some cases it tends to neglect revisions that involve more non-atomic changes or the introduction of complex concepts.

This chapter shows how nonmonotonic ILP can be applied to TR. The intuition behind the approach presented here is that a nonmonotonic ILP system has full power over the semantic changes of a given theory. In other words, by definition, nonmonotonic ILP systems are able to learn hypothesis $H$ such that the conse-
quences of $B \cup H$ are not necessarily a superset of the consequences of $B$. This is also a result of the application of TR systems that, rather than adding only new clauses, can retract and modify existing rules. Our approach consists of computing inductive hypotheses that are used as a prescription for syntactic variations in rules. A nonmonotonic ILP system derives a semantically correct hypothesis that is used to transform existing clauses syntactically. The final result is guaranteed to be consistent with the examples and the effect is equivalent to the one obtained with operator-based TR systems. This enables the application of the techniques described in this thesis to TR and provides an alternative approach for those classes of problems where locally optimal search techniques are not appropriate.

In this chapter we first introduce the setting for TR and review the main concepts of operator-based techniques. Then we provide our alternative solution to TR based on nonmonotonic ILP, discussing its computational steps and properties. For completeness, we review the related work, analysing the features of other TR systems. We also provide a novel application of TR to the problem of revising normative frameworks [Boella & Van der Torre, 2005].

### 7.2 Theory Revision

A TR system is biased towards the computation of theories that are similar to a given revisable theory. Whenever an initial (even incorrect) theory exists, either because it was provided by an expert or available from previous revisions, minimal revision is, in general, preferable to rediscovery. Our revision algorithm uses a measure of minimality similar to that proposed in [Wogulis & Pazzani, 1993] and defined in terms of number of revision operations required to transform one theory into another.

**Definition 7.1** Let $R'$ and $R$ be two logic programs. A revision transformation $r$ is such that $r(R) = R'$, and $R'$ is obtained from $R$ by deleting a rule, adding a fact, adding a condition to a rule in $R$ or deleting a condition from a rule in $R$. $R'$ is a revision of $R$ with distance $c(R, R') = n$ iff $R' = r^n(R)$ and there is no $m < n$ such that $R' = r^m(R)$.

For example, given the theory $R = \{fly(X) \leftarrow bird(X)\}$, the theory $R' = \{fly(X) \leftarrow bird(X), \text{not penguin}(X)\}$ is a revision of $R$ with distance 1. Note that, although we refer to Definition 7.1, it is also possible to weight revisions differently or introduce different transformations, e.g. increasing the distance by 2 when deleting a rule.

**Definition 7.2** (Theory revision task and hypothesis). A TR task is a tuple $\langle E, B, R, R' \rangle$ where $E$ is a set of literals, called examples, $B$ is a theory, called background theory, $R \subseteq \mathcal{R}$ is a theory, called revisable theory,
and \( R \) is a rule space. The theory \( R' \), called revised theory, is a TR hypothesis for the task \( \langle E, B, R, R' \rangle \) with distance \( c(R, R') \), iff (i) \( R' \subseteq R \); (ii) \( B \cup R' \models E \); (iii) if a theory \( S \) exists that satisfies conditions (i) and (ii) then \( c(R, S) \geq c(R, R') \), (i.e. minimal revision).

Optimal hypotheses can be defined similarly to Definition 3.3, by introducing a loss function in place of requiring entailment. In the case of TR, the loss function should include also the notion of minimality of the transformation.

Let us consider \( B = \{ \text{animal}(X). \text{bird}(X). \text{penguin}(c). \text{pigeon}(b) \} \), \( R \) as in the previous example, and \( E = \{ \text{fly}(b) \} \) and \( R' \) a superset of \( R' \) given above. \( R' \) is a hypothesis for the TR task \( \langle E, B, R, R' \rangle \) with distance 1. The main difference with the ILP task given in Definition 3.4 is the availability of an initial revisable theory and the consequent bias, as discussed in more detail in the following sections.

The method introduced here is based on mode declarations. We adapt Definition 7.2 in order to support the specification of the rule space \( R \) in terms of mode declarations. Whenever the TR system uses mode declarations we will refer to it as a tuple \( \langle E, B, R, M \rangle \) where condition (i) in Definition 7.2 becomes \( R' \subseteq \mathcal{R}_M \).

### 7.2.1 The operator based approach

Traditionally, TR systems are based on the application of revision transformations. It is useful to introduce the standard approach to appreciate how the approach we propose later in this chapter differs from it. Add rule, delete rule, add condition and delete condition are atomic revision operators and their application can transform a theory \( R \) into any theory \( R' \) within \( \mathcal{R} \) [Wogulis & Pazzani, 1993]. In general the operators used are not simply atomic operators but more complex operators. This is required in some cases because combined or repeated applications can factorise transactions that could never be explored by the particular exploration strategy employed (e.g. hill climbing can be trapped in local maxima) [Alphonse, 2004].

Algorithm 10 shows the main steps performed by most TR systems. After the initialisation of the best score, the algorithms enters an iteration whose termination is based on the loss of the derived revision, usually defined, in turn, on the coverage of the revised theory and the distance. For example if \( B \cup R \) entails the examples, there is no need for further revisions and the procedure TERMINATE returns true, thus terminating the revision.

The procedure FIND REVISION POINTS finds and collects the clauses (or other revision points, like conditions or predicates) whose revision can improve the theory with respect to the examples. Usually, some useful information can be derived during this step that is used to drive the exploration of the revisions, e.g. the revision points are ordered by point potential that is a measure of the expected improvement. GENERATE REVISIONS uses the revision points, to generate a list of possible revisions using the given rule space as constraints. These two steps are characterise the particular revision strategy and a number of implementations have been given in
**Algorithm 10** THEOREY-REVISION

**Inputs:** $R$ revisable theory, $B$ background theory, $R$ rule space, $E$ examples

**Outputs:** $R$ revised theory

1: $R_{in} = R$
2: $l_{loss} = LOSS(\emptyset, R_{in}, B, E)$
3: while not TERminate($best_score, R$) do
4:     $rp = \text{FIND\_REVISION\_POINTS}(R, B, E)$
5:     $tl = \text{GENERATE\_REV_IONS}(rp, R)$
6:     $best\_revision = \emptyset$
7:     for all revision $\in tl$ do
8:         $l = LOSS(revision, R, B, E)$
9:         if $l < l_{loss}$ then
10:            $l_{loss} = l$
11:            $best\_revision = revision$
12:        end if
13:     end for
14:     apply $best\_revision$ to $R$
15: end while
16: return $R$

Literature (see 7.4.2 as an example of such implementations based on the use of heuristics driven by transaction potential, intuitively an upper bound on the score obtained from a revision). A further iteration tests the generated revisions in order to seek the revision that results in the lowest loss. When no more revisions are available or can improve the theory the cycle terminates and the best transaction is applied.

Not shown in Algorithm 10 for simplicity, in some systems, in order to avoid checking all the possible revisions, a lower bound on the score for a given revision can be used involving the point potential or a function of the number of examples affected by the revision. If the lower bound for the score of a certain revision operator is higher than the current lowest loss, the inner cycle can be terminated. This restricts the search only to a subset of the $tl$ possible revisions generated. Considering the algorithm as exploration of the tree of possible revisions, the inner cycle explores horizontally the children of a node, while the outer cycle deepens the search.

Although we focus on non-interactive systems, it is worth mentioning that since the space of possible solutions is, in general, bigger than the ILP hypothesis space (given that the learning of new rules is a particular case of TR), some approaches to TR are based on interaction, where an oracle (or expert) is queried about the truth of formulae, thus introducing at “run-time” examples that the learning process finds particularly important to direct the search.
7.3 Theory revision through nonmonotonic ILP

We present in this section a technique based on a theory transformation that can employ a generic ILP system to perform TR. The technique is based on the use of mode declarations, so we assume that the ILP system supports mode declarations as specification language. The revision algorithm consists of three phases: the pre-processing phase that transforms the rules of the revisable theory $R$ into defeasible rules by “weakening” the conditions; the learning phase that computes the solutions as inductive hypotheses of the transformed task; and the post-processing phase that re-factors the hypotheses into revised rules based on the changes learnt in the second phase. Informally, rules learned in the learning phase can be seen as prescriptions for changes in the revisable theory $R$ in order to cover the examples. These changes can be addition or deletion of entire rules, and addition or deletion of literals in the body of existing rules.

Algorithm 11 shows how a nonmonotonic ILP system, which would normally be used to learn rules from scratch, can be also used to discover a set of revisions to an initial set of rules. The input of the algorithm is a TR task. The mode declarations define the atoms that are allowed to be the head of newly learnt rules and conditions that can be part of the body of both newly derived rules and revised ones. For ease of exposition, given a revisable theory $R$, we use the convention that $i$ refers to the index of a given rule and $j$ refers to the index of a literal within a rule in $R$, assuming an arbitrary order for the rules.

**Algorithm 11 THEORY-REVISION**

**Inputs:** $E$ examples, $B$ background theory, $R$ revisable theory, $M$ mode declarations,

**Outputs:** $R$ revised theory

$$(R, M) = \text{PRE-PROCESSING}(R, M)$$

$H = \text{ILP}(E, B \cup \tilde{R}, \tilde{M})$

$R' = \text{POST-PROCESSING}(R, H)$

return $R'$

**Pre-processing phase:** During this phase (Algorithm 12) the given revisable theory $R$ is rewritten as a normal logic program, $\tilde{R}$, suitable for learning revisions. This consists of the following two syntactic transformations. First, for every rule in $R$, every body literal $b^i_j$ is replaced by the atom $\text{try}(i, j, b^i_j)$, where $i$ is the index of the rule, $j$ is the index of the body literal in the rule and the third argument is a reified term for the literal $b^i_j$. Second, the literal $\text{extension}(i, h_i, \text{var}_i)$ is added to the body of the rule where $i$ is the index of the rule, $h_i$ is the reified term for the head of the rule, and $\text{var}_i$ is the list of variables that appear in the body but not in the head of the rule. For each $\text{try}(i, j, b^i_j)$ introduced into the program, the rules $\text{try}(i, j, b^i_j) \leftarrow \text{use}(i, j), b^i_j$ and $\text{try}(i, j, b^i_j) \leftarrow \text{not use}(i, j)$ are added to the program, together with the definition $\text{use}(I, J) \leftarrow \text{not del}(I, J)$ of the predicate $\text{use}^1$. Head mode declarations for $\text{extension}$ and $\text{del}$ are added to $M$. In Algorithm 12, $h^i_\up$ denotes the term

---

1This part of the transformation uses a mechanism adopted before in HAIL [Ray, 2005] in its last inductive phase, where literals in the
obtained from \( h_j \) by replacing variables with input placemarkers. Overall, these transformations set the learning task to compute exceptions cases for rules in the revisable theory \( R \) and instances of body literals that need to be deleted.

Note that depending on the available knowledge about the given revisable theory, different levels of control can be applied to the type of the variables. For example if the rules are annotated with the type of the variables that appear in it and with the placemarkers associated with them, then it is possible to include the type information in the mode declarations and include in the list \( \text{var}_i \) only the variables associated with an output placemarker. In the default case the types are kept generic and all the variables are included in \( \text{var}_i \) so that they can be used in a revision of the rule.

**Algorithm 12** PRE-PROCESSING

**Inputs:** \( R \) revisable theory, \( M \) mode declarations

**Outputs:** \( \tilde{R} \) pre-processed theory; \( \tilde{M} \) pre-processed mode declarations

\[
\tilde{R} = \{ \text{use}(I, J) \leftarrow \text{not del}(I, J) \}
\]

\[
\tilde{M} = M \cup \{ \text{modeh(del(#index, #index))} \}
\]

for all \( h_i \leftarrow b_1^i, \ldots, b_n^i \in R \) do

Let \( \text{var}_i \) be the set of variables that appear in the body and not in the head of \( h_i \)

\[
\tilde{R} = \tilde{R} \cup \{ \text{try}(i, 1, b_1^i), \ldots, \text{try}(i, n, b_n^i), \text{extension}(i, h_i, \text{var}_i) \}
\]

\[
\tilde{M} = \tilde{M} \cup \{ \text{modeh(extension(#index, #index, \text{var}_i^i))} \}
\]

for all \( b_j^i \) do

\[
\tilde{R} = \tilde{R} \cup \{ \text{try}(i, j, b_j^i) \leftarrow \text{use}(i, j, b_j^i) \} \cup \{ \text{try}(i, j, b_j^i) \leftarrow \text{not use}(i, j) \}
\]

end for

end for

return \((\tilde{R}, \tilde{M})\)

**Example 7.1** Assume a set of mode declarations

\[
M = \begin{cases} 
\text{modeb(c1(+any))} \\
\text{modeb(c2(+any))} \\
\text{modeb(c3(+any))}
\end{cases}
\]

and a revisable theory \( R \) of the form \( R = \{ p(X) \leftarrow c1(X), c2(X) \} \). Following algorithm 12, first \( \tilde{R} \) is extended as \( R \cup \{ \text{use}(I, J) \leftarrow \text{not del}(I, J) \} \) and similarly \( \tilde{M} = M \cup \{ \text{modeh(del(#index, #index))} \} \). Then for the only rule in \( R \), \( \text{var}_i \) is empty, \( h_1 = p(X), b_1^1 = c1(X) \) and \( b_1^2 = c2(X) \). At the end of the algorithm the extended language bias is returned as:

\[
\tilde{M} = M \cup \{ \text{modeh(extension(1, p(+any), []))}, \text{modeh(del(1, 1)), modeh(del(1, 2))} \}
\]

---

bodies of the kernel set are rewritten as \text{try} predicates in order to be selected by an abductive search.

128
and the transformed program $\tilde{R}$ as:

$$\tilde{R} = \begin{cases} 
  p(X) \leftarrow \text{try}(1, 1, c_1(X)), \\
  \text{try}(1, 2, c_2(X)), \\
  \text{extension}(1, p(X), []). \\
\end{cases}$$

$$\begin{cases} 
  \text{try}(1, 1, c_1(X)) \leftarrow \text{use}(1, 1), c_1(X). \\
  \text{try}(1, 1, c_1(X)) \leftarrow \text{not use}(1, 1). \\
  \text{try}(1, 2, c_2(X)) \leftarrow \text{use}(1, 2), c_2(X). \\
  \text{try}(1, 2, c_2(X)) \leftarrow \text{not use}(1, 2). \\
  \text{use}(X, Y) \leftarrow \text{not del}(X, Y). \\
\end{cases}$$

**Learning phase:** This phase relies on a nonmonotonic ILP system in order to compute a set of hypotheses for the task $\langle E, B \cup \tilde{R}, \tilde{M} \rangle$. The minimality of the revision depends on the minimality notion used in the ILP system that is implemented in the loss function associated with the task (discussed in Section 7.3.1). In fact, the changes applied in the theory are a function of the new del facts learnt (each del fact in $H$ corresponds to the deletion of a condition) and of the new extension rules (the number of literals in the body corresponds to literals added to existing rules).

**Example 7.2** Consider Example 7.1. Assume that $E = \{p(a), \text{not } p(b)\}$ and that the background knowledge $B$ is:

$$B = \begin{cases} 
  c_2(X) \leftarrow c_3(X) \\
  c_3(b). \\
  c_2(a). \\
\end{cases}$$

Note that $B \cup R \not\models E$. A generic ILP system would produce in this case the following hypothesis:

$$H = \begin{cases} 
  \text{del}(1, 1). \\
  \text{extension}(1, p(X), []) \leftarrow \text{not } c_3(X). \\
\end{cases}$$

This hypothesis is such that $B \cup \tilde{R} \cup H \models E$.

**Post-processing phase:** The post-processing phase generates a revised theory $R'$ semantically equivalent to
\( \bar{R} \cup H \) (and thus consistent with \( E \)). Informally, for each \( \text{del}(i, j) \) fact in \( H \) the corresponding condition \( j \) in rule \( i \) in \( R \) is deleted. For each extension rule in \( H \) of the form \( \text{extension}(i, h_i, \text{var}_i) \leftarrow c_1, \ldots, c_n \), the corresponding rule \( i \) in \( R \) is replaced by a new rule with the additional conditions \( c_h, 1 \leq h \leq n \). When for a certain rule there is no corresponding extension rule in \( H \) then the rule is deleted due to the fact that the extension condition in the body is not defined in \( H \) or elsewhere in the revisable theory. This is theoretically justified by the soundness and completeness results given in Section 7.3.1. Intuitively if extension has no definition, then the body in which it appears is falsified, making the rule always true. Algorithm 13 details the procedure.

**Algorithm 13** POST-PROCESSING

**Inputs:** \( \bar{R} \) revisable theory; \( H \) inductive hypothesis

**Outputs:** \( R' \) revised theory

\[
R' = \emptyset \\
\text{for all } h_i \leftarrow b^1_i, \ldots, b^n_i \in R \text{ do} \\
\pi = \{ b^1_i, \ldots, b^n_i \} \setminus \{ b^j_i : \text{del}(i, j) \in H \} \\
\tau = \{ c_j : \text{extension}(id, h_i, \text{var}_i) \leftarrow c_1, \ldots, c_n \in H, i \leq j \leq n \} \\
R' = R' \cup \{ h_i \leftarrow \pi, \tau \} \\
\text{return } R'
\]

**Example 7.3** Considering again Example 7.1. Given the \( H \) and \( \bar{R} \) described in the previous steps of the example, the following final revised program \( R' \) would be derived:

\[
R' = \left\{ p(X) \leftarrow e2(X), \text{not } e3(X) \right\}
\]

It is worth remarking that the method presented is one of many possible solutions. For example, instead of adding extension literals to the body, we could add negative exception literals, as shown in the following example.

**Example 7.4** Considering the same inputs as the previous examples, the following theory would be generated
with this alternative encoding as a result of the pre-processing phase:

$$
\tilde{R} = \begin{cases} 
  p(X) \leftarrow \text{try}(1, 1, c1(X)), \\
  \text{try}(1, 2, c2(X)), \\
  \text{not exception}(1, p(X)). 
\end{cases}
$$

A possible inductive hypothesis would be given as:

$$
H = \begin{cases} 
  \text{del}(1, 1), \\
  \text{exception}(1, p(X), []) \leftarrow c3(X). 
\end{cases}
$$

Resulting in the revised theory:

$$
R' = \begin{cases} 
  p(X) \leftarrow c2(X), \text{not } c3(X) 
\end{cases}
$$

The advantage of using the \textit{extension} predicate for the revision compared to the \textit{exception} predicate is that all the conditions in the body can be added to the same rule and that negative recursion is prevented (maintaining the original tightness property and avoiding multiple stable models).

On the other side, using exceptions enables the use of an ILP system where the loss function does not need to be modified, given that a standard compression measure would be close to the intended minimality criterion sought in revision. In fact, in this case, an empty hypothesis results in no revision (rather than the deletion of all the rules).

### 7.3.1 Soundness, completeness and minimality

According to Definition 7.2, TR must take into account minimality. We can map the minimality of the revision into the minimality of the inductive hypothesis derived by the ILP system. In particular we use the distance function defined in Definition 7.1, i.e. the revision $R'$ of the theory $R$ has distance $c(R', R)$ if it can be obtained by applying $c(R', R)$ transformations (deleting/adding a condition, deleting/adding a rule) and it cannot be
obtained with less transformations. We recursively define a loss function for a hypothesis $H'' = \{r\} \cup H'$ as follows:

$$l'(E, B, H'') = \begin{cases} 
1 + l'(E, B, H') & \text{if } r = \text{del}(i, j) \\
|b| + l'(E, B, H') & \text{if } r = \text{extension}(i, h, v) \leftarrow b \\
|r| + l'(E, B, H') & \text{otherwise}
\end{cases} \quad (7.1)$$

where the base case for an empty hypothesis is given by $l(E, B, \emptyset) = 0$. In order to take into account deleted rules we further refine the function as follows

$$l(E, B, H) = l'(E, B, H) + \sum_{\{1 \leq i \leq |R|: \text{extension}(i, h, v) \leftarrow b \in \tilde{R}\}} g(i)$$

where $g(i) = n + 1$ iff $h \leftarrow b_1, \ldots, b_n, \text{extension}(i, h, v) \leftarrow b \in \tilde{R}$. $i$ ranges over the indexes of all the rules in the revisable theory, thus the sum adds to the loss function the total number of literals deleted. Note that an atomic rule of the type $\text{extension}(i, h, v)$ in $H$ does not contribute to the score.

It is easy to verify that the revision generated from a hypothesis $H$ with minimum loss is a minimal revision, with the same notion of minimality as in Definition 7.1. In the general case, depending on the task at hand this function can be combined with further requirements, e.g. on the coverage of the examples.

We want to study now the soundness and completeness property of this approach. In order to ensure soundness, it is sufficient to prove that as a result of the post processing the inductive hypothesis together with the initial theory is equivalent to the revised theory. For completeness we need to show that all the possible revisions have a corresponding hypothesis $H$.

**Theorem 7.1 (Soundness of the transformation)** Let $R' = \text{POST-PROCESSING}(R, H)$. Then $B \cup \tilde{R} \cup H \equiv_{\text{ut}_{B \cup R'}} B \cup R'$.

**Proof** We apply an unfolding transformation sequence to $B \cup \tilde{R} \cup H$. Consider a rule $h_i \leftarrow b_1^i, \ldots, b_n^i \in R$. Suppose $\text{del}(i, j) \in H$. We can apply unfolding to the corresponding rule $h_i \leftarrow \text{try}(i, 1, b_1^i), \ldots, \text{try}(i, n, b_n^i), \text{extension}(i, h_i, \text{var}_i)$ in $\tilde{R}$, using the definition of $\text{try}$ obtaining two rules $h_i \leftarrow \text{try}(i, 1, b_1^i), \ldots, \text{use}(i, j), b_j^i, \ldots \text{try}(i, n, b_n^i)$ and $h_i \leftarrow \text{try}(i, 1, b_1^i), \ldots, \text{not use}(i, j), \ldots \text{try}(i, n, b_n^i)$. Again, applying unfolding on these two rules using the definition of $\text{use}$, we obtain the following two rules

$$h_i \leftarrow \text{try}(i, 1, b_1^i), \ldots, \text{not del}(i, j), b_j^i, \ldots \text{try}(i, n, b_n^i)$$
and

\[ h_i \leftarrow \text{try}(i, 1, b_i^1), \ldots, \text{del}(i, j), \ldots, \text{try}(i, n, b_i^n) \]

Since \( \text{del}(i, j) \in H \) we can discard the former rule and apply unfolding again on the latter, obtaining

\[ h_i \leftarrow \text{try}(i, 1, b_i^1), \ldots, \text{try}(i, n, b_i^{j-1}), \text{try}(i, n, b_i^{j+1}), \ldots, \text{try}(i, n, b_i^n) \]

If \( \text{del}(i, j) \notin H \) it is easy to see that similarly the transformation sequence results in the rule

\[ h_i \leftarrow \text{try}(i, 1, b_i^1), \ldots, \text{try}(i, n, b_i^{j-1}), b_i^j, \text{try}(i, n, b_i^{j+1}), \ldots, \text{try}(i, n, b_i^n) \]

Generalising to multiple \( \text{del} \) atomic rules in \( H \) we obtain a theory \( R_{\text{del}} \) where, as in Algorithm 13, all the literals for which a corresponding \( \text{del} \) atomic rule is in \( H \) are deleted. According to Theorem 2.3, \( B \cup \bar{R} \cup H \equiv B \cup R_{\text{del}} \cup H \). We can apply the same reasoning to the rules in \( R_{\text{del}} \) adding conditions through \( \text{extension} \) rules and deleting those rules that do not contain \( \text{extension} \). We omit this part for brevity. As a final result we obtain the full revision of the rules given by adding and deleting conditions, called \( R_{\text{unf}} \). Let us partition \( H \) into \( H_{\text{new}} \) and \( H_{\text{revision}} \) where the latter contains the definitions of \( \text{del} \) and \( \text{extension} \). We call \( R^- \) the definitions of \( \text{try} \) and \( \text{use} \) from \( R_{\text{unf}} \) (they have not been affected by the unfolding transformations), while we call the rest of the rules, obtained by unfolding, \( R^+ \). \( R^+ \cup H_{\text{new}} \) is the theory \( R' \) as constructed by the algorithm (it contains the revised rules, those not affected by the revision and the newly derived rules). Given that the transformations preserve the semantics we have \( B \cup \bar{R} \cup H \equiv B \cup R' \cup \bar{R}^- \cup H_{\text{revision}} \). \( R^- \cup H_{\text{revision}} \) defines a set of predicates (namely \( \text{try}, \text{use} \) and \( \text{del} \)) that do not appear in \( B \cup R' \). Thus models of \( B \cup R' \) are characterised only by atoms in \( \mathcal{U}_{B \cup R'} \). Therefore \( B \cup R' \equiv \mathcal{U}_{B \cup R'} B \cup R' \cup \bar{R}^- \cup H_{\text{revision}} \equiv B \cup \bar{R} \cup H \) thus proving the theorem.

**Theorem 7.2 (Completeness of the revision).** Let \( \langle E, B, R, M \rangle \) be a TR task. Then for each possible revision \( r^n(R) \) there exists a \( H \) such that \( \text{POST-PROCESSING}(R, H) = r^n(R) \)

**Proof** We need to prove the existence of \( H \) for any possible revision. Adding rules is trivial since they are just required to appear in \( H \). So we concentrate on the case where \( H \) contains only revision rules and \( r \) only refers to deleting a rule or adding or deleting a condition. Let us prove the theorem by induction on \( n \). Clearly, for \( n = 0 \) there is a \( H \) such that nothing changes in the theory, that includes an atomic rule \( \text{exception}(i, h, v) \) for each rule \( i \). Suppose a hypothesis \( H_n \) exists for \( r^n(R) \) and let us consider \( r(r^n(R)) \). \( r \) can refer to the deletion of a rule, and to the addition or deletion of a condition. In the first case (deletion of a rule) we can construct a theory \( H_{n+1} = H_n \setminus \{ \text{extension}(i, h, v) \} \) for any rule \( i \), thus we can delete any rule. Similarly we can add a
del atomic rule or an extension \((i, h, v)\) rule to \(H_j\) thus obtaining any atomic revision that can be obtained by applying \(r\) on \(r^w(R)\). Note that since a hypothesis \(H\) exists for any possible revision, if the underlying system is complete, \(H\) will be a candidate solution and thus ultimately computed and used within the revision.

**Theorem 7.3** (Soundness and completeness of the revision) Let \((E, B, R, R_M)\) be a TR task and let ILP be a sound and complete ILP procedure. Then \(R' = \text{THEORY-REVISION}(E, B, R, M)\) in Algorithm 11 iff \(R'\) is a TR hypothesis.

**Proof** The soundness (the only if part) is ensured by Theorem 7.1. Since \(H\) is the optimal solution for the ILP task then, based on the semantics equivalence and the use of the loss function previously described, \(R'\) is also a minimal revision. The completeness (the if part) is ensured by Theorem 7.2. In fact every possible revision is such that there is an inductive hypothesis that maps into it. Given the completeness of the ILP system, such hypothesis is always found.

### 7.4 Related work

As previously discussed in Section 7.2.1, TR systems are traditionally based on hill-climbing searches over the space of possible revisions, where the search is guided by revision operators. For completeness, we review in this Section the main non-interactive TR systems. All these systems employ a radically different approach to TR from the one we propose here. Ultimately the main difference is that they cannot delegate the search to a nonmonotonic ILP system but they are ad hoc solutions. A very common solution employs complex revision operators that complement the atomic operators that delete and add conditions and rules. These complex operators are used to avoid getting trapped in a local optimum when using greedy search. On the other hand, this rules out exhaustive searches, given the high number of possible alternative revisions.

As samples of interactive first-order TR systems, MIS [Shapiro, 1981], one of the precursors of modern ILP and TR systems, performs general-to-specific search and it is based on the interaction with the user to identify incorrect clauses and for membership queries. CLINT [De Raedt & Bruynooghe, 1991] is another interactive TR system whose operators can only add and delete clauses. KR-FOCL [Pazzani, 1989] queries the user to choose which revision with the same accuracy to apply.

Amongst the many propositional (RTLS [Ginsberg, 1990], DUCE [Muggleton, 1987], DUCTOR [Cain, 1991]) TR systems, EITHER [Mooney & Ourston, 1991] is the predecessor of FORTE and it is based on similar operators using in addition abductive techniques. FLORA [Widmer & Kubat, 1996] explicitly deals with concept drift, embedding solutions for drift tracking, noise handling and recurring contexts.
7.4.1 RUTH

RUTH [Adé et al., 1994] is one of the earliest systems proposed for non-interactive first-order TR. The inputs for RUTH are $R$, a set of function-free definite rules, $B$ a set of function-free definite rules and, instead of examples, an integrity theory $I$ is provided. The integrity theory is used to generate examples. A first step checks whether $R$ violates $I$ and generates an example $e$ that is an instance of violation. The example $e$ that has been generated is added to an example list $E$. Whenever $E$ is not empty then one of the examples is chosen to instantiate a revision. RUTH terminates when $I$ is consistent with $R \cup B$.

Referring to Algorithm 10, if $e$ is an uncovered positive example then $tl$, the list of possible revisions at a certain iteration, is generated by the procedure generate_revisions and contains a list of the possible applications of so called exception, add clause and abduction operators that respectively add conditions to existing rules, clauses to the theory and abduce missing explanations. For positive examples there is no revision point detection. Otherwise, if the selected example is negative, a delete clause operator is used on a revision point previously identified. In this case the procedure find_revision_points derives all the clauses that are used in the derivation of such example. In general, after one of the operators $r$ is chosen, $r(R)$ is computed on the current revised theory $R$ and then checked again for consistency. Inconsistency might generate more examples and a new iteration starts.

As stated in [Adé et al., 1994] any search strategy could be potentially used but a depth first iterative deepening strategy is chosen, thus a depth-first solution with a number of transactions up to an increasing length is computed at each iteration.

7.4.2 FORTE

FORTE revises function-free definite theories. The inputs are $E$ (set of facts), $R$ (set of function-free definite rules), $B$ (normal theory, fundamental domain theory in [Richards & Mooney, 1995]). The operators employed are delete rule, hill climbing add condition, relational pathfinding, delete condition, add rule, identification and absorption.

FORTE closely resembles Algorithm 10. The outer cycle starts with the computation of a list of revision points $rp_i$ ordered by decreasing revision potential $pot_i$ (the maximum increase in theory accuracy that could result from a revision at that point). Clauses are annotated as specialisation points when a derivation of a negative example that involves them is successful. Generalisation points annotations are detected in failures of proofs of positive examples. A condition $l$ is annotated if it leads to a failing derivation of a positive example $e$ (this kind of condition is called “failure point”). Failure can be caused also by other conditions that may have bound some variable to incorrect values (“contributing point”).
FIND REVISION POINTS generates a revision point $rp_i$ for every clause $c$ such that: a) $c$ has been annotated for specialisation or b) some of conditions in $c$ have been annotated. The revision point potential $pot_i$ is calculated as the number of distinct examples whose proof caused an annotation on the revision point $rp_i$. Also in this case $pot_j$ is calculated as the sum of all examples that caused an annotation.

The transaction list $tl$ created in GENERATE_REVISIONS is ordered by its associated revision point potentials, thus the revision point with highest potential is handled first by means of its appropriate operators. A score (equivalent to the inverted loss) drives the search. The score for a revision is calculated as the normalised number of examples that are correctly classified ($S = 1$ if all the inconsistencies are corrected). In case of a tie, the revision that results in the smallest theory is preferred. The inner cycle of Algorithm 10 checks if the potential, that is defined as the potential of the revision point of the transaction, is higher then the current best score. In this case the transaction is applied calculating the score. When the theory cannot be improved the transaction with the best score is applied and the outer cycle starts another iteration. FORTE terminates whenever for a certain node it is unable to generate any revision that improves the theory.

[McCluskey & West, 2001] presents a domain specific first-order non interactive theory revision system based on FORTE’s hill-climbing algorithm. The task addressed is the maintenance of a theory describing air traffic rules. Given the nature of the task and preliminary experiments, in order to reduce the search space, only a specific subset of operators has been chosen as particularly useful. A region revision operator revises ordinal parameters and a relation revision operator replaces conditions with conditions of the same sort.

7.4.3 INTHELEX

INTHELEX (INcremental THEory Learner from EXamples) [Esposito et al., 2004] is a fully incremental (i.e. it does not require an initial theory) TR system based on the notion of Object Identity (OI). It adopts a multistrategy approach combining induction and abduction.

The inputs are a background theory $B$ (definite, function free, linked, hierarchical clauses), an initial theory $R$ (definite, function free, linked, hierarchical clauses), a set of examples $E$ (ground definite function free clauses), an abstraction theory $T_{abs}$ (definite, function free, linked, hierarchical clauses), a dependency graph $D$ that defines a hierarchy over the concepts that are learnt, and a set of abducibles. The particular language INTHELEX relies on is a variant of Datalog called $Datalog^{OI}$ where OI stands for object identity. The OI assumption determines that, within a clause $r$, terms denoted with different symbols must be distinct. We denote a clause $r$ under the OI assumption by $r_{OI}$. This, in contrast to what happens in Prolog, forbids the
unification of two different variables in a clause. For example the clause

\[ p(X) \leftarrow q(X, Y), r(a, X), s(Y, a). \]

is associated with the following implicit constraints: \( X \neq Y, X \neq a, Y \neq a. \) Despite this bias \( \text{Datalog}^{OI} \) is as expressive as Datalog (see [Semeraro et al., 1997] for the proof). OI is introduced with the purpose of obtaining a simpler structure of the search space than the one based on \( \theta \)-subsumption. \( \theta \)-subsumption is defined according to OI. A Datalog clause \( d \) \( \theta \)-subsumes a Datalog clause \( c \) under object identity (\( d \theta_{OI} \)-subsumes \( c \)) iff there exists a substitution \( \theta \) such that \( d_{OI} \theta \subseteq c_{OI} \) where \( d_{OI} \) (resp. \( c_{OI} \)) is obtained from \( d \) (resp. \( c \)) by adding the additional inequalities.

\textsc{Inthelex} employs two conceptual specialisation \( \rho_{OI} \) and generalisation \( b_{OI} \) operators. Informally if \( c \) is a Datalog clause, \( d \in \rho_{OI}(c) \) is obtained by substituting a variable in \( c \) with a constant or by adding a literal to the body of \( c \). \( d \in b_{OI}(c) \) is obtained by substituting a constant in \( c \) with a new variable or by deleting a literal from the body of \( c \). The two operators are ideal for Datalog clauses ordered by \( \theta_{OI} \)-subsumption [Semeraro et al., 1996].

The main disadvantage of OI is that it is weaker than \( \theta \)-subsumption and thus it is a worse approximation than logical entailment. Moreover the least general generalisation is not unique under OI assumption [Semeraro et al., 1997]. A background theory \( B \) that is also used in the pre-processing phase of examples for saturation can be provided. Saturation adds new literals to examples, considering the dependency graph if dependencies are defined for the predicate to which the example refers. It is also possible to specify an abstraction theory \( T_{abs} \) that is used to eliminate irrelevant details from examples. Abstraction, in contrast to saturation, does not involve the use of the dependency graph. For a more in depth discussion on these operations see [Saitta & daniel Zucker, 1998].

The algorithm is structured as follows. An abductive procedure is used in \textsc{Inthelex} to check consistency of examples. Two cases trigger the revision of the current theory. If the example is positive and not covered then the theory is generalised, if it is negative and covered then the theory is specialised.

Referring to Algorithm 10, the \textsc{Find_revision_points} procedure selects clauses in the order they are written in the theory in case of generalisation. In case of specialisation \( rp \) is the list of all the clauses involved in the derivation of the inconsistent example \( e \) ordered by decreasing depth in derivation.

In case of generalisation the \( lgg_{OI} \) operator is tried first, then the \textit{generalised exception} and finally the \textit{exception} operator. In case of specialisation, first the \textit{hill climbing positive condition addition under OI} is considered, then the \textit{hill climbing negative condition addition under object-identity} and finally the \textit{exception} operator are applied. We refer to [Esposito et al., 2000] for a definitions of these revision operators.
ILP has been successfully applied to a wide range of problems (for example [Muggleton et al., 1996], [Džeroski & Lavrač, 2000], see [Džeroski & Lavrač, 2001] for a review). The nonmonotonic extension is a recent development of the field and understandably less validated by practical applications. The advantages of nonmonotonic inference have been discussed in previous chapters of this thesis. Notably, from an applied perspective, logic programs with negation as failure are usually shorter and easier to manage and they are an established representation language. Examples where nonmonotonic ILP is applied include the analysis of biological networks [Ray et al., 2010b] and the extraction of requirements in the context of automated software engineering [Alrajeh et al., 2011, Alrajeh et al., 2007, Alrajeh et al., 2009].

7.5 Normative framework design

In this section we show a novel application area, in the context of Normative frameworks (also called institutions or virtual organisations). We detail the formalisation of the problem and presenting a case study that exemplify and motivate the approach. This application takes advantage of some of the distinctive features of the ASPAL system. In particular, as the inductive learning assists a designer, it is essential that all the possible solutions are found within the specified boundaries as we argue later in Section 7.5.4.

Norms and regulations play an important role in the governance of human society. Social rules such as laws, conventions and contracts prescribe and regulate our behaviour. Normative frameworks allow automated reasoning about the consequences of socially acceptable and unacceptable behaviour by monitoring the permissions, empowerment and obligations of the participants and flagging violations when norms are not followed.

The details of a normative framework, e.g. particular cases and unexpected sequences of events, make it relatively easy to miss crucial operations needed to model the intended behaviour. To make an analogy with software engineering, this characterises the gap between requirements and implementation and what we describe here can be seen as an automated mechanism to support the validation of normative frameworks.

We show how to use ASPAL to fill gaps in the rules of an existing normative framework, encoded by an Ans-Prolog program. The designer normally develops a system with a certain behaviour in mind. This intended behaviour can be captured in use cases which comprise two components: a description of a scenario and the expected outcome when executing the scenario. Use cases are added to the program to validate the existence of an answer set. Failure to find an answer set for the program indicates that the specification does not yield the intended behaviour. In this case, the program and the failing use case(s) are given to an inductive learning tool, which will then return suggestions for improving the normative specification such that the use cases are satisfied.
We demonstrate the methodology through a case study and discuss the distinctive challenge involved in this ILP problem.

### 7.5.1 Normative frameworks and formal model

The concept of normative framework has become firmly embedded in the agent community as a necessary complement to the essential autonomy of agents, in just the same way as societal conventions and legal frameworks have developed to constrain the behaviour of people. While the concept remains attractive, its realisation in a computational setting remains a subject for research, with a wide range of existing logics [Sergot, 2004, Artikis et al., 2003, Boella & Van der Torre, 2005, Cliffe et al., 2006, Singh, 2000] and tools [Rodriguez-Aguilar, 2001, Dignum, 2004, Hopton et al., 2009] that support it.

The essential idea of normative frameworks is a (consistent) collection of rules whose purpose is to describe a principle of right action binding upon the members of a group and serving to guide, control, or regulate proper and acceptable behaviour [Merriam-Webster dictionary]. These rules may be stated in terms of events, specifically the events that matter for the functioning of the normative framework.

To provide the context, we give an outline of a formal event-based model for the specification of normative frameworks that captures all the essential concepts, namely empowerment, permission, obligation and violation. We adopt the formalisation from [Cliffe et al., 2006].

The essential elements of the normative framework are events which bring about changes in state, and fluents, which characterise the state at a given instant. The function of the framework is to define the interplay between these concepts over time, in order to capture the evolution of a particular institution through the interaction of its participants. Normative frameworks distinguish two kinds of events: normative events, that are the events defined by the framework, and exogenous events, some of whose occurrence may trigger normative events in a direct reflection of “counts-as” [Jones & Sergot, 1996]. Normative events are further partitioned into normative actions, that denote changes in normative state, and violation events, that signal the occurrence of violations. Violations may arise either from explicit generation, (i.e. from the occurrence of a non-permitted event), or from the non-fulfilment of an obligation. We also distinguish two kinds of fluents: normative fluents that denote normative properties of the state such as permissions, powers and obligations, and domain fluents that correspond to properties specific to a particular normative framework. A normative state is represented by the fluents that hold true in this state. Fluents that are not present are considered to be false. Conditions on a state are expressed by a set of fluents that should be true or false.

Changes in a normative state are achieved through the definition of two relations: (i) the generation relation, which specifies how the occurrence of one (exogenous or normative) event generates another (normative) event,
subject to the empowerment of the actor and the conditions on the state, and (ii) the consequence relation, which specifies the initiation and termination of fluents, subject to the performance of some action in a state matching some condition.

The semantics of a normative framework is defined over a sequence, called a trace, of exogenous events. Starting from the initial state, each exogenous event is responsible for a state change, through initiation and termination of fluents. Given the definition of the normative framework, for each trace, we can compute a sequence of states that constitutes the model of the normative framework for that trace. This process is realised as a computational model through ASP and it is this representation that is used in the learning process described here.

The mapping to AnsProlog described more in detail in [Corapi et al., 2011a] and [Cliffe et al., 2006] is such that ASP can be used to reason about properties of the framework and the models of the normative framework for given traces. The following atoms are used in the AnsProlog translation: instant(i) for time instances, next(i1, i2) to establish time ordering, occurred(e, i) to indicate that the (normative) event happened at time i, observed(e, i) that the (exogenous) event was observed at time i, holdsat(p, i) to state that the normative fluent p holds at i, and finally initiated(p, i) and terminated(p, i) for fluents that are initiated and terminated at i.

The generation relation is modelled through rules of the type:

\[
\text{occurred}(g, T) \leftarrow \\
\text{occurred}(e, T), \\
\text{holdsat}(	ext{pow}(e), T), \\
\ldots 
\]

where e represents the normative event occurring and holdsat(pow(e), T) models the condition that the event e is empowered.

The consequence relation is modelled through rules of the type:

\[
\text{terminated}(p, T) \leftarrow \\
\text{occurred}(e, T), \\
\ldots 
\]
Figure 7.1: Iterative design driven by use cases.

\[
\text{initiated}(p, T) \leftarrow \\
\text{occurred}(e, T),
\]

\[\ldots\] (7.4)

which model the initiation (resp. termination) of a fluent \( p \) given the occurrence of an event \( e \).

Traces of events can be specified as facts (e.g. \( \text{observed}(e, t) \)). We refer to a complete trace when all exogenous events for a given time interval are specified. A trace is incomplete when the model needs to determine the missing exogenous events. When the model is supplemented with the \textit{AnsProlog} specification of a complete trace, we obtain a single answer set corresponding to the model matching the trace\(^2\).

### 7.5.2 Methodology

Use cases represent instances of executions that are known to the designer and that drive the elaboration of a normative system. If the current formalisation of a normative system does not match the intended behaviour in the use cases then the formalisation is not complete or is incorrect, and an extension or revision is required.

Each use case \( u \in U \) is a tuple \( (T, O) \) where \( T \), a trace, specifies a set of exogenous events (\( \text{observed}(e, t) \)), and \( O \) is a set of \textit{holdsat} and \textit{occurred} literals that represent the \textit{expected output} of the use case. Given a set \( U \) of use cases, \( T_U \) and \( O_U \) denote, respectively, the set of all the traces and expected outputs in all the use cases in \( U \). The time points of the different use cases relate to different instances of executions of the normative system to avoid the effect of events in one use case on the fluents of another use case. The use cases can, but do not have to, be complete traces (i.e. an exogenous event for each time instance) and expected output can contain positive as well as negative literals.

For a given translation of a normative framework \( N \), the designer must specify what part of the theory is subject to revision. The theory is split into two parts: a “revisable” part, \( N_T \), and a “fixed” part, \( N_B \). By default

\(^2\)The structure of the program (the stratified base part and observed events as facts) guarantees that the program has exactly one answer set. See [Cliffe, 2007] for further details and proofs.
the former includes rules that are domain independent while the latter includes the rules that are specific to the normative framework considered.

Given a set $U$ of use cases, a TR task for a normative framework $N$ is defined as the tuple $\langle O_U, N_B \cup T_U, N_T, M \rangle$, where $M$ includes by default a body declaration for any static relation declared in $N_B$, and the following mode declarations (where the schema is formed by substituting arguments with input place-markers): $mode_h(occurred(e^*, +instant))$, for each $e \in E_{norm}$; $mode_h(initiated(f^*, +instant))$ and $mode_h(terminated(f^*, +instant))$, for each $f \in F$; $mode_b(holdsat(f^*, +instant))$, for each $f \in F$; $mode_b(occurred(e^*, +instant))$, for each $e \in E$.

The choice of the set of mode declarations $M$ is crucial and is ultimately the responsibility of the designer. Many mode declarations ensure higher coverage of the specification but increase the computation time. Conversely, fewer mode declarations improve performance but may result in partial solutions. The choice may be driven, for example, by previous design cycles, or by interest in more problematic parts of the specification.

As shown in Figure 7.1 the design of a normative system is an iterative process. The representation $N$ in ASP of a system described by the designer using a normative language is tested against a set of use cases also provided by the designer. This analysis step is performed by running an ASP solver over $N$, extended with the observed events included in the use cases, and a constraint indicating that no answer set that does not satisfy $O$ is acceptable. Conceptually, if the solver is not able to find an answer set (i.e. returns unsatisfiable), then some of the given use cases are not satisfied in the answer sets of $N$ and a revision step is performed. Possible revisions are provided to the designer who ultimately chooses the most appropriate one.

7.5.3 Case Study

We illustrate the methodology with a small but sufficiently rich case study that demonstrates the key properties and benefits of our proposed approach. The following is a description of a reciprocal file sharing normative framework.

The active parties—agents—of the scenario find themselves initially in the situation of having ownership of several (digital) objects—the blocks—that form part of some larger composite (digital) entity—a file. An agent is required to share a copy of a block they hold before they can download a copy of block they are missing. Initially each agent holds the only copy of a given block and there is only one copy of each block in the agent population. Some vip agents are able to download blocks without any restriction. Agents that request a download and have not shared a block after a previous download generate a violation for the download action and a misuse violation for the agent. A misuse terminates the empowerment of the agent to download blocks.
Following the specification, the designer can produce the following use case \((T,O)\):

\[
T = \begin{cases}
  \text{observed}(\text{start}, i00).
  \\
  \text{observed}(\text{download}(\text{alice}, \text{bob}, x3), i01).
  \\
  \text{observed}(\text{download}(\text{charlie}, \text{bob}, x3), i02).
  \\
  \text{observed}(\text{download}(\text{bob}, \text{alice}, x1), i03).
  \\
  \text{observed}(\text{download}(\text{charlie}, \text{alice}, x1), i04).
  \\
  \text{observed}(\text{download}(\text{alice}, \text{charlie}, x5), i05).
  \\
  \text{observed}(\text{download}(\text{alice}, \text{bob}, x4), i06).
\end{cases}
\]

\[
O = \begin{cases}
  \text{not viol}(\text{myDownload}(\text{alice}, x3), i01).
  \\
  \text{not viol}(\text{myDownload}(\text{charlie}, x3), i02).
  \\
  \text{not viol}(\text{myDownload}(\text{bob}, x1), i03).
  \\
  \text{not viol}(\text{myDownload}(\text{charlie}, x1), i04).
  \\
  \text{not viol}(\text{myDownload}(\text{alice}, x5), i05).
  \\
  \text{viol}(\text{myDownload}(\text{alice}, x4), i06).
\end{cases}
\]

The use case models a sequence of events (e.g. alice downloads the block \(x3\) from bob at the time point \(i01\)) that includes a violation at the time point \(i06\), while the download events at the other time points do not generate violations. The normative framework maps the exogenous events download into normative events myDownload that are used in the specification of the use cases. In the trace, charlie performs a download at time point \(i04\) without sharing a block after the last download. This is not expected to generate a violation since charlie is defined as vip \((\text{isVIP}(\text{charlie}) \in N)\).

The initial normative system includes the domain component and type definitions that are fixed for every problem instance and define the generic semantics of normative frameworks and a specific component given by the following revisable theory \(N_T\):
Some of these rules may not model the intended normative framework correctly. Given the use case and the above formalisation of the normative system, the first iteration of our approach proposes as one of the possible solutions the deletion of a condition in rule 5 \( r_5 \) and addition of a condition to rule 4 as shown below (leaving the other rules unaltered):

\[
N_T = \begin{cases} 
  r_1 : & \text{initiated}(\text{hasblock}(X, B), I) \leftarrow \\
  & \text{occurred}(\text{myDownload}(X, B), I). \\
  r_2 : & \text{initiated}(\text{perm}(\text{myDownload}(X, B)), I) \leftarrow \\
  & \text{occurred}(\text{myShare}(X), I). \\
  r_3 : & \text{terminated}(\text{pow}(\text{extendedfilesharing}, \text{myDownload}(X, B)), I) \leftarrow \\
  & \text{occurred}(\text{misuse}(X), I). \\
  r_4 : & \text{terminated}(\text{perm}(\text{myDownload}(X, B_2)), I) \leftarrow \\
  & \text{occurred}(\text{myDownload}(X, B), I). \\
  r_5 : & \text{occurred}(\text{myDownload}(X, B), I) \leftarrow \\
  & \text{occurred}(\text{download}(Y, Y, B), I), \text{holdsat}(\text{hasblock}(Y, B), I). \\
  r_6 : & \text{occurred}(\text{myShare}(X), I) \leftarrow \\
  & \text{occurred}(\text{download}(Y, X, B), I), \text{holdsat}(\text{hasblock}(X, B), I). 
\end{cases}
\]

(7.5)


\[
N'_T = \begin{cases} 
  r_4 : & \text{terminated}(\text{perm}(\text{myDownload}(X, B_2)), I) \leftarrow \\
  & \text{not isVIP}(X), \text{occurred}(\text{myDownload}(X, B), I). \\
  r_5 : & \text{occurred}(\text{myDownload}(X, B), I) \leftarrow \\
  & \text{holdsat}(\text{hasblock}(Y, B), I). 
\end{cases}
\]

(7.6)

However, this is not yet the intended formalisation – rule 5 in particular makes no intuitive sense. As an additional debugging facility the designer can analyse the answer sets associated with the revisions. For the previous revision, unwanted violations are generated at each time point. This feedback can be used to refine the use case provided. In fact the use case specifies the single specific violations that must not occur but it does not request explicitly that no violations should occur in the first five time points (e.g. \text{viol}(\text{myDownload}(alice,x3),i02), \text{viol}(\text{myDownload}(alice,x4),i02)). These violations can be observed in the answer set associated with the revision. The designer can then improve the use case by modifying the set of expected outputs:
\[ O = \begin{cases} 
\text{viol} (\text{myDownload}(\text{alice}, x4), i06), \\
\text{not viol} (\text{myDownload}(A, B), T), T \neq i06, \\
\text{occurred} (\text{misuse}(\text{alice}), i06), \\
\text{not occurred} (\text{misuse}(X), T), T \neq i06. 
\end{cases} \] (7.7)

In the subsequent iteration, the revision process suggests changes that include those identified in the previous iteration (i.e. addition of a condition in rule 4 and deletion of a condition in rule 5), and the addition of a further condition in the body of rule 5. The combined effect of these changes fixes the original error in the specification, by also changing the name of one of the variables. Furthermore, since the output \( O \) of the use case includes a desired \textit{misuse} event, which is not currently formalised in the system, the revision also suggests the new rule 7 given below. The final theory \( N'_T \) includes the following rules (leaving rules 1, 2, 3, 6 untouched):

\[
N'_T = \begin{cases} 
\text{r4:} \text{terminated} (\text{perm} (\text{myDownload}(X, B2), I)) \leftarrow \\
\text{not isVIP}(X), \text{occurred}(\text{myDownload}(X, B), I). \\
\text{r5:} \text{occurred}(\text{myDownload}(X, B), I) \leftarrow \\
\text{occurred}(\text{download}(X, Y, B), I), \text{holdsat}(\text{hasblock}(Y, B), I). \\
\text{r7:} \text{occurred}(\text{misuse}(X), I) \leftarrow \\
\text{occurred}(\text{viol}(\text{myDownload}(X, B)), I). 
\end{cases} \] (7.8)

In summary, rule 4 is corrected by adding an exception \textit{not isVIP}(X), rule 5 is revised by correcting a typographical error in its condition (i.e. the name of a variable was not the intended one – \textit{occurred}(\text{download}(Y, Y, B), I)), and finally, a new rule is learnt that defines \textit{misuse} coherently with respect to the provided use case. The computation time and other information are reported in Table 6.1 in Chapter 6.

### 7.5.4 Discussion

From an ILP perspective, the task is particularly challenging as it involves the revision of a given theory and in general produces a number of alternative solutions. Also, full accuracy over the examples is required, since they are provided by the designer and assumed to be correct. In this case examples are provided under a general form containing variables (see the expected outputs (7.7)), which are instantiated by the ASP grounder. Completeness is a crucial requirement as the output returned is given to a supervisor. If the system is not able to find any solution within a certain distance, the supervisor must be sure that no revisions are possible. An
incomplete system will always leave the undesirable option that a revision exists but the system was not able to find it.

The learning involves complex dependencies. The revision of a certain rule can have an effect on the observations regarding both the predicate occurred and viol. In the case study the following rules are contained in the fixed theory:

\[
N_B = \left\{ \begin{align*}
\ldots \\
\text{viol}(E, T) &\leftarrow \\
&\quad \text{occured}(\text{viol}(E), T).
\text{occured}(\text{viol}(E), I) &\leftarrow \\
&\quad \text{occured}(E, I), \\
&\quad \text{not holdsat}(\text{perm}(E), I).
\text{holdsat}(P, J) &\leftarrow \\
&\quad \text{holdsat}(P, I), \\
&\quad \text{not terminated}(P, I), \\
&\quad \text{next}(I, J).
\text{holdsat}(P, J) &\leftarrow \\
&\quad \text{initiated}(P, I), \\
&\quad \text{next}(I, J).
\ldots
\end{align*} \right. 
\] (7.9)

Revising the rules in the original \(N_T\) implies a different definition for the predicates terminated and initiated, which affects the predicates holdsat and thus the occurrence of violations. The undefined concept of occurrence of a misuse can be learned and defined by the ILP system using any of the conditions that are compatible with the mode declarations. These are also affected by the revisions in rules defining terminated and initiated, thus extending the potential set of dependencies in the final theory.

The solution reported in the theory (7.8), as shown in Table 6.1, is one of the 12 suboptimal solutions, i.e. with minimal depth, computed by ASPAL. In addition to the ability to generate hypotheses that revise the given theory based on complex dependencies, the possibility of deriving all possible solutions and of defining domain-dependent preference criteria is a peculiarity of ASPAL that makes it the only system able to support in full the normative framework design process shown here.
7.6 Conclusions

TR systems, similarly to ILP systems, attempt to find a hypothesis that, together with some available background knowledge, improves the quality of a theory (measured by some scoring function) with respect to a set of given examples. The main difference with ILP is that a given revisable theory is provided that can be modified. The revisions to this theory must be minimal in the sense that it is preferrable to obtain solutions that are as close as possible to it.

Traditionally this is done by applying revision operators to the theory, like adding a condition to a rule, and searching for a set of applications that improve the quality of the theory. The search is usually local in the sense that the theory is revised iteratively by applying the locally best revision. This prevents the available system to derive complex revisions that are not locally optimal. In general the optimality of the revision is not guaranteed.

We have proposed in this Chapter a different approach that does not consist of an ad hoc algorithm but maps the TR problem into a nonmonotonic ILP problem. This shows that TR for nonmonotonic theories and nonmonotonic ILP are able to solve the same class of problems. The advantages of this approach are: (1) a nonmonotonic ILP system can be employed, thus an available system can be used for TR; (2) completeness and thus optimality is ensured; (3) TR can be applied to normal theories, allowing for more expressiveness compared to systems that support only definite clauses or function-free languages.

We demonstrate how this approach can be applied to the revision of normative frameworks. In this scenario it is important that whenever a solution exists, this is presented to the designer, thus making it clear on whether the current formalisation admits a revision or simply the learning system was not able to find it. The application also shows that ASPAL can be used successfully in concrete nonmonotonic ILP problems, where complex solutions involving negation and mutual dependencies amongst the target concepts are expected.
8 Conclusions

In this thesis we have introduced a novel ILP approach based on “meta-abductive” inference over the structure of logic rules. The approach consists of a general transformation that produces from a given ILP task an equivalent ALP task that can be used to derive inductive hypotheses. The transformation preserves the semantics and, conditionally to a sound and complete underlying ALP system, guarantees soundness and completeness for the initial ILP task. We have then presented two systems, based on this transformation, that are the only nonmonotonic ILP system with a result of completeness with respect to the whole set of input examples.

The transformation can be used, in principle, with any ALP system. We have developed two alternative solutions. One is based on a goal-driven ALP system called SLDNFA. In order to gain more control over the search we have extended the top theory representation generated by the transformation with control abducible predicates that can be used to evaluate partial solutions and prune unnecessary states from the search. The resulting system, called TAL, can be used to derive complex hypotheses involving recursion and multiple interdependent concepts. We have showed how TAL is able to derive the expected solution for a problem, in which the system learns the concepts of even and odd numbers, which highlights some shortcomings of other ILP systems that are unable to derive the desired hypothesis or perform a less constrained, potentially non-terminating search.

TAL also can be used on monotonic ILP problems, by constraining the mode declarations provided and by executing it within a sequential covering cycle. We showed that the predictive accuracy of TAL is comparable to that of PROGOL and TOPLLOG by evaluating the hypotheses learnt on four well-known ILP tasks.

In the second implementation solution, we have adapted the transformation by explicitly generating a theory, called a skeleton theory, that represents the rule space for the ILP task as an ASP program. A solver can be used to extract inductive solutions under this representation of the problem. The use of ASP greatly improves the performance compared to TAL in those problems that make intensive use of negation and integrity constraints. The computation time is greatly affected by the maximum number of conditions allowed in the rules, that is, the depth of the hypothesis. From this consideration we have designed a system called ASPAL that searches for hypotheses by iterating over increasing depth.

Notably, the use of a flexible preference criterion for the hypothesis is left as future work in [Ray, 2005] and
not addressed in [Ray, 2009b]. The solution we provided natively supports the use of optimisation features that translate directly to preference criteria over the hypotheses. Furthermore we allow the distinction between integrity constraints that can be used as hard constraints over the solution space and negative examples that can act as soft constraints, biasing the learning process through the preference criterion.

The techniques provided are flexible enough to operate on a range of applications, from well-known ILP problems to more recent tasks that model problems that involve nonmonotonicity. We have introduced a general technique for theory revision, which is based on the transformation of the given task into a nonmonotonic ILP problem. This opens a new application area for nonmonotonic ILP, as shown in a novel applications discussed, where the ASPAL is applied to revise normative frameworks.

Both TAL and ASPAL, given their completeness properties, provide an integrated solution to recursive, multi-predicate, non observational predicate learning. Our study showed how these properties emerged from the application of abductive reasoning, the use of conjunction of the positive and negative examples in the derivation and the construction of a set of rules that does not take place iteratively but within the same reasoning process.

The two solutions are not immune from shortcomings, but they are complementary as they can be applied in different settings. TAL is currently implemented as a prototype in Prolog. The current implementation can be greatly improved, in particular by optimising the underlying abductive system specifically for ILP tasks. The main limit of TAL is the amount of redundant computations that are performed for hard classes of problems. Since complex dependencies amongst the concepts being learnt cannot be ruled out, a complete example-driven search for all but the smallest problems is in practice unfeasible. These issues have led to the development of ASPAL, where the underlying ASP solver, which uses a different type of search based on SAT solving techniques and heuristics, manages to reduce the computation time considerably. On the other hand the limitation for ASPAL lies in the necessity for the solver to ground the input program. Problems with a particularly large domain and that require hypotheses that contain rules with a large number of conditions result in higher execution times, due to the costly grounding. The limitations on performance are a direct consequence of the increased expressiveness of the language supported for the background knowledge and for the hypotheses in TAL and ASPAL. We believe that based on the theoretical background provided improved implementations can be applied successfully in a larger range of real world problems than those that have been addressed so far in the field of ILP.
8.1 Future Work

Starting from the two prototypes developed, further studies should be pursued on performance. The problem of reducing the computation time in ASPAL can be addressed using approximations. In particular we developed a prototype, not discussed in this thesis, that derives a solution by constructing an initial hypothesis with depth $d_0$ (a set of rules where each rule has at most $d_0$ conditions) and by extending it incrementally, adding at each step $d$ conditions to the current set of rules. The learning steps use the theory revision technique introduced in this thesis to extend the rules. This method sets a bound to the computation time based on $d$ at the cost of favouring locally optimal solutions. In TAL, future work could address the impact of different types of heuristics and restrictions on the trade-off between performance and accuracy. Also, an optimised implementation would lower the average computation time. In particular we believe the use of tabling techniques [Swift & Warren, 2010] would speed up the derivation process by avoiding repeated computations. In fact, portions of the derivation are often replicated and can be saved during the process.

Another direction of work should address the use of ASPAL with different classes of logic programs, making use of the language supported by the underlying ASP solver. In principle, ASPAL can be used to learn disjunctive logic programs and integrity constraints. In particular the learning of integrity constraints can be tackled both in TAL and ASPAL, using the same principles introduced in this thesis, adding integrity constraints derived from mode declarations in the top theory in the first case and as skeleton rules in the second. We refer to future work to study the effectiveness of this method. Mode declarations are not the only possible representation mechanism for the language bias. More in general, similarly to TOPLOG, a grammar can be used to derive the top theory and to define more complex constraints on the structure of target hypotheses.

The mapping can be used to exploit probabilistic semantics for logic programs and the associated learning mechanisms. In [Corapi et al., 2011b], for example, we apply a modified transformation and use gradient descent to derive locally optimal probabilities associated with the hypothesis. The semantics adopted is the distribution semantics [Sato, 1995], extended to abductive programs similarly to [Poole, 1991]. Clearly, by using our abductive transformation that relates the structure of the rule to atoms, and ultimately to a set of propositional variables, other estimation techniques can be used.

Nonmonotonic ILP is a relatively young field and applications have been proposed only recently. We collected a set of problems to experiment on the applicability of the systems proposed. This set can be extended and used as a benchmark for future comparisons since, in general, traditional ILP problems are not adequate to evaluate nonmonotonic ILP systems.

Conclusively, we believe our work not only produced a solid framework for nonmonotonic ILP, supported
both by theoretical results and preliminary experiments on two alternative implementations, but also presented a radically different perspective on the problem. By devising an approach based on problem reduction we gained insight on the connections between abductive reasoning and different types of rule learning problems. We observed how problems that have been addressed by ad hoc systems in the past like learning of recursive concepts are incorporated in this approach and how computational steps performed in other ILP systems are closely related to the derivation that results from the encoding discussed in this thesis. We are confident that our work will be used as a reference for a deeper theoretical understanding of the problem of learning logic rules under nonmonotonicity and as a starting point for future efficient implementations.
Bibliography


