UNIVERSITÀ DEGLI STUDI DI UDINE
HIC SUNT FUTURA

DEPARTMENT OF MATHEMATICS, COMPUTER SCIENCE AND PHYSICS

MASTER THESIS IN
COMPUTER SCIENCE

Learning from Answer Sets via ASP Encodings

CANDIDATE
Roberto Borelli

SUPERVISOR
Prof. Agostino Dovier

Academic Year 2024-2025

Institute Contacts
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine — Italia
+39 0432 558400
https://www.dmif.uniud.it/

"A thing of beauty is a joy forever"
— John Keats

To Sofia, my beauty

Acknowledgements

Prima di iniziare con i risultati tecnici, desidero ringraziare alcune persone che sono state fondamentali in questo viaggio durato cinque anni.

Innanzitutto, ringrazio con tutto il cuore la mia fidanzata, Sofia. Abbiamo vissuto insieme le esperienze più belle e significative. Mi hai sostenuto in ogni circostanza e abbiamo costruito insieme ricordi indelebili, e questo è solo l'inizio. Anche dopo tanti anni passati insieme, continuo ad ammirarti ogni giorno, dal tuo spirito pieno di vita e di energia, fino alla profondità del tuo animo. Senza di te non sarei la persona che sono oggi, e ti devo molti dei miei traguardi. In particolare, ti dedico questa tesi.

Esprimo la mia profonda gratitudine al mio relatore, il Professor Agostino Dovier: la sua professionalità rappresenta per me un modello a cui aspirare come informatico, mentre la sua disponibilità e gentilezza costituiscono un esempio umano di grande valore.

Un sentito ringraziamento va alla mia famiglia, tutta, che è sempre stata fonte di amore e unione. In particolare, ringrazio i miei genitori per il loro costante supporto e per aver reso possibile questo percorso, senza mai farmi mancare nulla.

Ringrazio anche tutti i miei amici, con cui ho condiviso momenti di studio e di svago. In particolare: Giosuè, Ale, Doch, Manuel, Enrico, e tutti gli altri che non menziono solo per brevità, ma a cui sono profondamente grato.

Un ringraziamento speciale va a Luca Geatti e al Professor Montanari, per avermi guidato su diversi fronti e per avermi offerto un'esperienza stimolante, sempre contraddistinta da grande disponibilità e supporto.

Infine, ringrazio il mio professore delle scuole superiori, il Professor Roldo, di cui ricordo ancora con affetto le lezioni sui MAT, per avermi trasmesso la curiosità di "scoprire le cose".

Abstract

In recent years, Learning from Answer Sets (LAS) has gained increasing attention as a symbolic AI technique well-suited for explainable machine learning. The need for explainability has become a key requirement in modern AI applications, making LAS an appealing framework due to its strong logical foundations and interpretability. LAS provides a formal approach to inductive logic programming (ILP), where the goal is to learn hypotheses in the form of logic programs that explain given positive and negative examples under the answer set semantics. This thesis presents a comprehensive study of LAS, analyzing its fundamental algorithms and complexity results. A particular focus is given to the satisfiability problem of an LAS task - that is, an instance of the problem of Learning from Answer Sets - which is known to be Σ_2^P-complete. To address this problem, we develop novel Answer Set Programming (ASP) encodings that translate an LAS task into an ASP program whose stable models correspond one-to-one with the solutions of the original task.

We propose two alternative encodings. The first, simpler but less efficient, relies on an ASP program of exponential size. The second encoding, which introduces disjunctive rules, makes use of the saturation technique by Eiter and Gottlob. This approach ensures a linear encoding size when the background knowledge combined with the hypothesis space forms a tight program.

A key contribution of this work is the development of `lasco` (LAS to ASP Compiler), a tool that implements both encodings and translates LAS tasks into ASP code executable by standard solvers. `lasco` supports ground and non-ground tasks and handles common ASP extensions such as arithmetic expressions and cardinality constraints.

Finally, we evaluate our approach on a suite of representative benchmarks and compare its performance to ILASP, the state-of-the-art LAS solver. While ILASP generally demonstrates superior performance, largely due to its maturity and extensive optimizations, our newly developed tool `lasco` outperforms all versions of ILASP on certain benchmark instances (in particular on tight tasks) and remains competitive in many others. These results highlight the potential of our approach, suggesting that the disjunctive encoding is not only of theoretical interest but also has practical relevance, and motivates further exploration.

Contents

1

Introduction

Motivations and related work

Answer Set Programming (ASP) is a declarative programming paradigm rooted in logic programming and based on the stable model semantics introduced by Gelfond and Lifschitz [1]. ASP is particularly effective for modeling complex combinatorial problems, including classical puzzles such as the n-queens and Sudoku [2], planning tasks [3, 4], and scheduling problems [5]. ASP has also been successfully applied in industrial contexts, as highlighted by Falkner et al. [6]. The solutions of an ASP program are called *answer sets*. A key feature of ASP is its explainability and interpretability [7]: given an answer set containing an atom p, one can trace back the derivation steps to understand why p is included in the solution. In recent years, significant effort has been devoted to enhancing explainability in ASP systems, particularly through the development of tools such as XASP [8, 9, 10], which provide structured explanations for answer set programs.

Inductive Logic Programming (ILP), originally introduced by Muggleton (1991) [11], is a branch of symbolic machine learning that aims to learn logic programs from examples and background knowledge. Formally, an *ILP task* T is defined as a tuple $\langle B, S_M, E\rangle$, where B is a logic program called the *background knowledge*, S_M is a set of rules called the *hypothesis space*, and E is a set of examples. An *inductive solution* $H \subseteq S_M$ is a hypothesis such that the program $B \cup H$ entails all examples in E. Different ILP frameworks can be defined by varying: (i) the semantics adopted for the logic programs in B and S_M; (ii) the nature of the examples (e.g., ground atoms or partial interpretations); and (iii) the notion of entailment used to evaluate hypotheses against the set of examples, such as *brave* or *cautious* entailment.

A significant advance in ILP was introduced by Law (2018) [12], who proposed the Learning from Answer Sets (LAS) framework. In this setting, the background knowledge is represented by an ASP program (without disjunctive heads), and the learning task involves two distinct sets of examples: the set of positive examples E^+ and the set of negative examples E^-. A distinguishing feature of LAS is that positive and negative examples are treated asymmetrically: an inductive solution is required to *bravely* entail the positive examples and *cautiously* avoid entailing the negative ones. Law et al. [13] also introduced the ILASP system (Inductive Learning of Answer Set Programs), which supports a broad family of LAS-based learning frameworks. ILASP algorithms operate by iteratively solving a series of ASP programs until a valid inductive solution is found. The most recent version, ILASP4 [13, 14], improves upon its predecessors by integrating a conflict-driven learning strategy that enhances efficiency

and scalability.

ILP (and in particular frameworks based on Answer Set Programming) is widely used in domains where explainability and symbolic reasoning are essential, such as bioinformatics, robotics, and legal reasoning. Unlike sub-symbolic approaches (such as deep neural networks) which often act as black-box models and require post-hoc interpretability techniques of limited fidelity [15], ILP provides inherently interpretable models grounded in logic. This makes it particularly suitable for safety-critical applications and for scenarios where transparency and human trust are required. The importance of explainable models has been increasingly recognized, especially within the European research agenda. Large-scale initiatives, such as those promoted by the European Commission have explicitly emphasized the need for trustworthy and explainable AI systems [16, 17]. In this context, symbolic approaches like ILP and LAS are considered promising tools to bridge the gap between performance and interpretability. This is further supported by a growing body of recent applications that demonstrate their effectiveness across diverse and practical domains. For instance:

- In [18], Fossemo et al. (2022) investigate the use of ILASP to globally explain neural networks trained on user preferences, exploring dimensionality reduction via PCA to improve scalability while maintaining interpretability.
- In [19], Ieolo et al. (2023) introduce the first ILP-based approach to learn LTLf formulas from example traces using ILASP, showing improved performance over SAT-based and exhaustive methods.
- In [20], Dreossi (2023) uses the ILASP system to learn constraints from logic puzzles like n-queens, 15 puzzle, star battle and flow lines.
- In [21], Chiarello et al. (2024) present an ILASP-based approach to Petri net model repair that integrates domain knowledge and structural constraints.
- In [22], Dovier et al. (2024) present the development of XAI-LAW, a logic programming-based tool based on Answer Set Programming and inductive logic programming to model, and explain legal decisions in the Italian criminal system, aiming to support judicial reasoning through formalized legal knowledge.
- In [23], Dreossi et al. (2024) explore the use of FastLAS, an Inductive Logic Programming system [24], to develop an explainable and accurate weather forecasting tool that learns human-readable rules from small datasets, offering interpretable predictions and competitive performance compared to traditional machine learning models.

Main contributions

Building on the foundations and applications discussed above, this thesis focuses specifically on the *satisfiability problem* for LAS tasks. Given a task T, we are interested in determining whether there exists an inductive solution (more details are given in section 2.2.2). When such a solution exists, we also aim to compute it. This problem has been shown to be Σ_2^P-complete [12], i.e., complete for the second level of the polynomial hierarchy. Notably, the Σ_2^P-hardness arises from the cautious entailment

requirements: if the framework is restricted to brave entailment only, the complexity of the satisfiability problem drops to NP-complete. In the context of ASP, the Σ_2^P complexity can be addressed in two main ways: (i) by constructing an ASP program of exponential size using only normal rules; or (ii) by employing disjunctive logic programming, which allows a more compact encoding by using disjunctions in rule heads. The second approach is more involved and relies on the so-called *saturation technique*, introduced by Eiter and Gottlob in 1995 [25], and further discussed in works such as [26, 27, 28].

In this thesis, we explore both of the aforementioned approaches. First, we present an encoding of exponential size, denoted as P_{exp}, which is inspired by the encoding used in the ILASP1 algorithm [12]. The encoding in ILASP1 is capable of handling only brave entailment, and it addresses cautious requirements through multiple iterative calls to an ASP solver. In contrast, our P_{exp} encoding directly captures cautious entailment by explicitly enumerating all possible answer sets within the encoding itself. Although this first encoding is not practically efficient, it serves as a valuable conceptual baseline that illustrates the theoretical expressiveness required for Σ_2^P-complete reasoning. The second encoding, denoted as P_{dis}, achieves linear size under the assumption that the program $B \cup H$ is tight. It is based on two key ideas: (i) instead of enumerating all answer sets to enforce cautious entailment, we represent them symbolically using a Boolean formula, following the methodology introduced by Lin and Zhao [29]; (ii) we apply the saturation technique from Eiter and Gottlob [25] to translate this formula into an ASP program with disjunctive rules.

The key advantage of our methodology is that it follows a *single-shot* approach, in contrast to the *multi-shot* strategy adopted by the ILASP system. Our encodings allow a LAS task to be solved through a *single* invocation of an ASP solver, whereas ILASP requires multiple iterative calls. This single-shot design enables the ASP solver to fully leverage its internal optimization techniques—such as constraint propagation, intelligent grounding, and conflict-driven learning, by having a complete global view of the problem. Importantly, by offloading the reasoning process to a mature ASP solver, our approach benefits from decades of research, theoretical analysis, and performance engineering embedded in systems like Clingo [30] and DLV [26]. In contrast, ILASP implements its own custom optimization layers, which, while effective in many cases, cannot yet match the robustness, generality, and efficiency of state-of-the-art ASP engines.

To demonstrate the practical viability of our approach, we implemented both encodings in a tool named `lasco` (LAS to ASP Compiler), developed in Haskell. The tool takes as input a LAS task and generates an ASP program that can be executed using standard ASP solvers. `lasco` supports both ground tasks and a broad class of non-ground tasks. In the latter case, before the encoding is applied, the input is carefully grounded using `gringo` [30]. We evaluate the performance of `lasco` by comparing it against ILASP, using the suite of logic puzzles introduced in [20] as benchmark tasks.

Thesis structure

In **Chapter 2**, we introduce the syntax and semantics of ASP, along with key complexity results and the connection between ASP and SAT solving. We formally define several ILP frameworks—focusing in particular on LAS frameworks—analyze their computational complexity, and discuss both a naive algorithm and the ILASP algorithms. We also present a simple running example of a LAS task that will be used throughout the following chapters.

In **Chapter 3**, we present our main contributions. First, we describe the exponential encoding P_{exp}, and then we introduce the polynomial disjunctive encoding P_{dis}. For each encoding, we provide the formal ASP translation, prove correctness and completeness theorems, and illustrate the approach using the running example. Finally, we discuss how the encodings can be adapted to support non-ground tasks.

In **Chapter 4**, we present the `lasco` tool and its main features. We describe the input syntax, the command-line interface, and the supported ASP extensions. We then evaluate the performance of `lasco` by comparing it against ILASP on a set of benchmark tasks.

Finally, in **Chapter 5**, we draw our conclusions and outline several possible directions for future work, including the development of parallel and approximate algorithms for solving LAS tasks.

2

Background

In this chapter, we present the theoretical foundations necessary to support the contributions of this thesis. We begin by reviewing the key concepts of Answer Set Programming (ASP), including its syntax, semantics, and computational complexity. We then introduce several frameworks from Inductive Logic Programming (ILP), with particular focus on Learning from Answer Sets (LAS), and analyze their associated complexity results.

2.1 Answer Set Programming

We begin by formally introducing Answer Set Programming (ASP), the declarative language at the core of this work. This section covers its syntax and semantics, the notation adopted in the thesis, and the computational complexity of standard reasoning tasks, such as consistency checking, brave and cautious entailment. We also outline the connection between ASP and Boolean satisfiability.

2.1.1 Syntax

A *signature* is a tuple $\Sigma = \langle \mathcal{P}, \mathcal{V}, \mathcal{C} \rangle$, where $\mathcal{P}$ is a set of predicate symbols, $\mathcal{V}$ a set of variables, and $\mathcal{C}$ a set of constants. A *term* is either a constant or a variable. An *atomic formula* (or simply, an *atom*) is an expression of the form $p(t_1, \ldots, t_{ar(p)})$, where $p \in \mathcal{P}$ is a predicate symbol, each t_i is a term, and $ar(p)$ denotes the arity (i.e. the number of arguments) of p. An atom is said to be *ground* if it contains no variables. A *literal* is either an atom a or `not` a, where "`not`" denotes *negation as failure (naf)* [31]. An *ASP program* over the signature Σ consists of a finite set of rules. A *general disjunctive rule* R has the form:

$$\underbrace{H_1 \texttt{ | } \ldots \texttt{ | } H_k}_{head(R)} \texttt{ :- } \underbrace{A_1, \ldots, A_n, \texttt{not } B_1, \ldots, \texttt{not } B_m}_{body(R)}.$$

where each H_i, A_j, and B_k is an atom, and $k, n, m \geq 0$.

The *head* of the rule, denoted $head(R)$, is the disjunction of atoms $H_1, \ldots, H_k$. The *body*, denoted $body(R)$, is the conjunction of literals $A_1, \ldots, A_n$ and *not* $B_1, \ldots,$ *not* B_m. For simplicity we denote with $body^+(R)$ (resp. $body^-(R)$) the set of positive (resp. negative) atoms in the body of R. Logically, a rule expresses an implication: *if the body holds, then at least one atom in the head must also hold.*

The symbol "`,`" in the body denotes conjunction (logical AND), meaning all literals must be satisfied for the rule to apply. The symbol "`|`" in the head denotes disjunction (logical OR), indicating that the rule supports any of the head atoms when its body is true. A rule is said to be:

- *normal* if its head contains exactly one atom ($|head(R)| = 1$);
- *definite* if it is a normal rule with the body containing only positive literals ($m = 0$);
- a *fact* if its body is empty ($n = m = 0$), meaning the head is always true;
- a *denial* if its head is empty ($k = 0$), meaning the body must not be satisfied.

We may assume without loss of generality that $k > 0$ for all rules, as denials can be treated as syntactic sugar under the commonly used answer set semantics defined later in section 2.1.2.

We denote by $\mathsf{ASP}^{\mathsf{D}}$ the class of *disjunctive ASP programs* consisting of general disjunctive rules, by $\mathsf{ASP}^{\mathsf{N}}$ the class of *normal ASP programs* consisting only of normal rules and by $\mathsf{ASP}^{\mathsf{def}}$ the class of *definite ASP programs* consisting only of definite rules. Let P be an ASP program. We denote with $vars(P)$ the set of all variables appearing in P and we denote with $atoms(P)$ the set of all atoms in P (i.e. $atoms(P) \coloneqq \bigcup_{R \in P} head(R) \cup body^+(R) \cup body^-(R)$). A program is said to be *ground* if it contains only ground atoms (i.e. $vars(P) = \emptyset$).

2.1.2 Semantics

The semantics of Answer Set Programming (ASP) is designed to model commonsense reasoning, which is inherently non-monotonic. This means that adding new rules to a program P can invalidate previously derived conclusions, potentially causing atoms to disappear from the solution. Moreover, ASP adopts the principle of *negation as failure*, which departs from classical logic: if there is no evidence supporting an atom q, we assume `not` q. This principle enables ASP to represent incomplete knowledge effectively.

The solutions of an ASP program are called *answer sets* or *stable models*, a concept introduced by Gelfond and Lifschitz in 1988 [1, 32]. The set of answer sets of a program P is denoted by $AS(P)$. The *Herbrand universe* of an ASP program P, denoted with HU_P, is the set of ground terms generated from the signature Σ. The *Herbrand base* of an ASP program P, denoted with HB_P, is the set of all ground atoms that can be formed by applying the predicate symbols in $\mathcal{P}$ to all possible terms from the Herbrand universe HU_P. We begin by defining stable models for *ground* programs.

Let P be a ground program. An *Herbrand interpretation* I is a subset of the Herbrand base HB_P, i.e., a truth assignment over atoms. Let a be an atom, let $\{l_1, \ldots, l_n\}$ be a set of literals, and let R be a rule. The satisfaction relation $I \models \cdot$ is defined as follows:

- $I \models a$ if and only if $a \in I$;
- $I \models not\ a$ if and only if $a \notin I$;
- $I \models l_1 \wedge \cdots \wedge l_n$ if and only if $I \models l_i$ for all $i = 1, \ldots, n$;
- $I \models l_1 \vee \cdots \vee l_n$ if and only if $I \models l_i$ for some $i = 1, \ldots, n$;
- $I \models R$ if and only if $I \models head(R)$ or $I \not\models body(R)$;

- $I \models P$ if and only if $I \models R$ for every rule $R \in P$.

If $I \models P$, we say that I is a *model* of P. An interpretation I is a *minimal Herbrand model* of P if no proper subset $J \subsetneq I$ is also a model P. For the class $\mathsf{ASP}^{\mathsf{def}}$ of definite programs, the following result holds [33]:

Theorem 2.1 *Let P be a definite program. Then P admits a unique minimal Herbrand model, denoted M_P. Moreover, $M_P = \bigcap_{I \models P} I$.*

To define stable models for general programs, we first introduce the notion of the *Gelfond–Lifschitz reduct.*

Definition 2.2 (Reduct [1]) *Let P be a ground ASP program, and let I be an interpretation. The reduct of P with respect to I, denoted P^I, is defined as:*

$$P^I \coloneqq \left\{ head(R) \ \texttt{:-}\ body^+(R) \;\middle|\; R \in P,\ body^-(R) \cap I = \emptyset \right\}$$

Intuitively, P^I is obtained by:

1. Removing all rules whose body contains a negative literal `not` a with $a \in I$;
2. Removing all negative literals from the remaining rules.

If P is a normal program (i.e., $P \in \mathsf{ASP}^{\mathsf{N}}$), then P^I is definite and hence admits a unique minimal model.

Definition 2.3 (Stable Model [1]) *Let P be a ground ASP program. An interpretation I is a stable model (or answer set) of P if and only if I is a minimal model of P^I. In particular, if $P \in \mathsf{ASP}^{\mathsf{N}}$, then I is a stable model of P if and only if $I = M_{P^I}$.*

So far, we have defined the semantics of ASP for ground programs, i.e., programs in which all terms are constants. In practice, ASP programs are usually written using variables for conciseness and readability. To compute the answer sets of a non-ground program P we first compute a transformation called *grounding*. This operation on the program P produces a ground program, denoted with *ground*(P), on which we can compute $AS(P)$ as already seen.

Definition 2.4 (Ground instantiation) *Let P be a non-ground ASP program over a signature $\Sigma = \langle \mathcal{P}, \mathcal{V}, \mathcal{C} \rangle$. The ground instantiation of P (ground(P)), is the set of all rules obtained by replacing the variables in P with all possible combinations of ground terms from HU_P.*

We can now define the semantics for non ground-programs.

Definition 2.5 (Stable Model for non-ground programs) *Let P be a non-ground ASP program. I is an answer set of P if and only if I is an answer set of ground(P).*

2.1.3 Complexity

We now consider the complexity of various problem on ASP programs. Let P be an ASP program, let I be an interpretation, and let B be an atom. We define the following problems related to P:

- The *answer set checking problem*, denoted with $I \in AS(P)$, asks to decide if I is an answer set of P.
- The *consistency problem*, denoted with $AS(P) \neq \emptyset$, asks to decide whether or not P admits a stable model.
- The *brave entailment problem*, denoted with $P \models_b B$, asks to decide if B belongs to some answer set of P.
- The *cautious entailment problem*, denoted with $P \models_c B$, asks to decide if B belongs to all answer sets of P.

We first show that the class of ground $\mathsf{ASP}^{\mathsf{N}}$ programs corresponds precisely to the complexity class NP.

Theorem 2.6 *The consistency problem for ground* $\mathsf{ASP}^{\mathsf{N}}$ *programs is* NP-complete.

Proof. To show that the problem belongs to NP, consider that for a ground $\mathsf{ASP}^{\mathsf{N}}$ program P, any interpretation I of P satisfies $|I| < |P|$. Verifying whether I is a stable model of P requires computing the reduct P^I and then determining its minimal model M_{P^I}, both of which can be performed in deterministic polynomial time with respect to $|P|$.

We now show the NP-hardness. Consider a 3-CNF formula $\phi \coloneqq C_1 \wedge \cdots \wedge C_m$ where each C_i is a clause of the form $l_i^1 \vee l_i^2 \vee l_i^3$ over the set of variables $X = \{x_1, \ldots, x_n\}$. Consider the following ground $\mathsf{ASP}^{\mathsf{N}}$ program $P(\phi)$:

$$\texttt{x(i) :- not nx(i).}\quad \texttt{nx(i) :- not x(i).} \qquad \forall x_i \in X$$

$$\texttt{c(i) :- } \sigma(l_i^j)\texttt{.} \qquad \forall i = 1 \ldots m \;\; \forall j = 1 \ldots 3$$

$$\texttt{:- not c(i).} \qquad \forall i = 1 \ldots m$$

where $\sigma(l_i^j)$ is `x(k)` if l_i^j is x_k and $\sigma(l_i^j)$ is `nx(k)` if l_i^j is $\neg x_k$. It is now easy to check that $P(\phi)$ admits a stable model if and only if ϕ is satisfiable. □

Eiter and Gottlob [25] proved that ground $\mathsf{ASP}^{\mathsf{D}}$ programs (i.e., ASP programs with disjunctive heads) are strictly more expressive than ground $\mathsf{ASP}^{\mathsf{N}}$ programs. In particular, the class of ground $\mathsf{ASP}^{\mathsf{D}}$ programs captures the second level of the polynomial hierarchy, namely the class Σ_2^P. The proof of Σ_2^P-hardness is based on a reduction from the *validity* problem for quantified Boolean formulas (QBF) of the form $\exists X \forall Y \; \psi(X, Y)$, which is known to be Σ_2^P-complete.

Theorem 2.7 ([25]) *The consistency problem for ground* $\mathsf{ASP}^{\mathsf{D}}$ *programs is* Σ_2^P-complete.

Proof. We first show membership in Σ_2^P. Let P be an $\mathsf{ASP}^{\mathsf{D}}$ program, and let S be a given interpretation. To verify whether S is a stable model of P, we proceed as follows. First, we compute the reduct P^S,

which can be obtained in polynomial time and results in an $\mathsf{ASP}^{\mathsf{D}}$ program without negative literals. Then, we check in polynomial time whether S is a model of P^S. It remains to verify the minimality of S, namely that there is no proper subset $S' \subset S$ which is also a model of P^S. Determining the existence of such an S' is clearly an NP problem, as we can guess a candidate subset and verify it in polynomial time. Therefore, we employ an NP oracle to decide this subproblem. If the oracle answers negatively, then S is indeed a stable model of P, establishing membership of the problem in Σ_2^P.

We now prove the Σ_2^P-hardness. Let ϕ be the quantified Boolean formula defined as follows:

$$\phi \coloneqq \exists x_1 \dots \exists x_n \forall y_1 \dots \forall y_m \, \psi,$$

where ψ is a Boolean formula in disjunctive normal form (DNF) over the set of variables $\{x_1, \dots, x_n, y_1, \dots, y_m\}$. In particular, ψ has the form $D_1 \vee \dots \vee D_k$, where each disjunct D_i is defined as $D_i \coloneqq l_{i,1} \wedge l_{i,2} \wedge l_{i,3}$ with each $l_{i,j}$ being a literal over the variables. It is well known that deciding the validity of the formula ϕ is a Σ_2^P-complete problem. Therefore, to establish the Σ_2^P-hardness of the consistency problem for ground $\mathsf{ASP}^{\mathsf{D}}$ programs, it suffices to construct a program $P(\phi)$ that admits a stable model if and only if ϕ is valid. The program $P(\phi)$ is defined as:

`x(i) | nx(i).`	$\forall i = 1 \dots n$
`y(j) | ny(j).  y_j :- w.  ny(j) :- w.`	$\forall j = 1 \dots m$
`w :-` $\sigma(l_{h,1}), \sigma(l_{h,2}), \sigma(l_{h,3})$.	$\forall h = 1 \dots .k$
`w :- not w.`	

where $\sigma(l)$ is defined as:

$$\sigma(l) = \begin{cases} \texttt{x(i)} & \text{if } l \text{ is } x_i \\ \texttt{nx(i)} & \text{if } l \text{ is } \neg x_i \\ \texttt{y(i)} & \text{if } l \text{ is } y_i \\ \texttt{ny(i)} & \text{if } l \text{ is } \neg y_i \end{cases}$$

The rule `w :- not w` ensures that `w` cannot be false in any stable model of $P(\phi)$. Let M be a stable model of $P(\phi)$. Since `w` must be true, it must be supported by at least one rule of the form `w :-` $\sigma(l_{h,1}), \sigma(l_{h,2}), \sigma(l_{h,3})$. Moreover, as `w` is included in M, both atoms `y(i)` and `ny(i)` belong to M. From the rule `x(i) | nx(i)`, it follows that exactly one of the two atoms `x(i)` or `nx(i)` belongs to M; indeed, having both `x(i)` and `nx(i)` in M would violate the minimality condition of stable models. We now show that the program $P(\phi)$ has a stable model if and only if the formula ϕ is valid.

Suppose that S is a stable model of $P(\phi)$. Consider any interpretation I that agrees with S on the assignment of the atoms `x(i)` and `nx(i)` for every i, does not include `w`, and contains exactly one of `y(j)` and `ny(j)` for each j. Such an interpretation I cannot be a stable model, as otherwise it would contradict the minimality of S. Since I is not a stable model, there must exist some some h (with $1 \leq h \leq k$) such that $\sigma(l_{h,1}) \in I$, $\sigma(l_{h,2}) \in I$ and $\sigma(l_{h,3}) \in I$ and therefore (with some abuse of notation) $I \models D_h$. Hence, we have shown that there exists a truth assignment for the existential variables $x_1, \dots, x_n$ such that, for every possible assignment to the universal variables $y_1, \dots, y_m$, the

	$I \in AS(P)$	$AS(P) \neq \emptyset$	$P \models_b B$	$P \models_c B$
$\mathsf{ASP}^{\mathsf{N}}$	P-complete	NP-complete	NP-complete	coNP-complete
$\mathsf{ASP}^{\mathsf{D}}$	coNP-complete	Σ_2^P-complete	Σ_2^P-complete	Π_2^P-complete

Table 2.1: Recap of problem complexities for ground $\mathsf{ASP}^{\mathsf{N}}$ and ground $\mathsf{ASP}^{\mathsf{D}}$ programs [26].

formula ψ is satisfied. This directly implies that ϕ is valid.

Conversely, assume that the formula ϕ is valid. Then there exists a truth assignment τ for the existential variables $x_1, \ldots, x_n$ that satisfies ψ for every possible assignment of the universal variables $y_1, \ldots, y_m$. Consider the interpretation S defined as follows:

$$S \coloneqq \{\texttt{x(i)} \mid \tau(x_i) \text{ is true}\} \cup \{\texttt{nx(i)} \mid \tau(x_i) \text{ is false}\} \cup \{\texttt{y(j)}, \texttt{ny(j)} \mid j = 1, \ldots, m\} \cup \{\texttt{w}\}\,.$$

It is straightforward to verify that S is indeed a stable model of $P(\phi)$. This completes the proof. □

This construction illustrates the so-called *saturation technique* [25, 26, 27, 28], a standard method for encoding universal quantification in ASP using disjunctive rules. This technique forces a designated atom, called the *saturation predicate* (in this case, `w`), to be true in all stable models. This is achieved via the rule `w :- not w`, which requires `w` to be included. However, under stable model semantics, `w` must also be *supported*, i.e., derivable through some other rule in the program. This implies that `w` can appear in a stable model *only if* at least one disjunct D_h of the formula ψ is satisfied under the current assignment to the existential variables $x_1, \ldots, x_n$ and *for all possible assignments* to the universal variables $y_1, \ldots, y_m$. To simulate this universal quantification, the program forces both `y(j)` and `ny(j)` to be true for every j whenever `w` is true. If no such disjunct D_h is satisfied under all assignments to the y_j, then none of the support rules for `w` will fire, and the program becomes inconsistent.

Incorporating the proofs for theorems 2.6 and 2.7 was essential; the first theorem is crucial for foundational understanding, while the second serves as an introduction to the *saturation technique*, which we will apply in section 3.2. We do not report here the formal proofs for the complexity of the other fundamental ASP reasoning tasks, but we summarize the known results in Table 2.1 that can be read in [26]. The table reports the complexity classes for answer set membership, program consistency, brave entailment, and cautious entailment for both ground $\mathsf{ASP}^{\mathsf{N}}$ and ground $\mathsf{ASP}^{\mathsf{D}}$ programs.

It is worth noticing that the complexity of brave reasoning coincides with that of consistency checking. Given a program P and an atom `a`, deciding whether `a` is bravely entailed amounts to checking whether *there exists* a stable model of the program $P \cup \{$ `:- not a.`$\}$. That is, brave reasoning can be reduced to a standard consistency test on an augmented program. In contrast, cautious reasoning is computationally harder, as it requires verifying that the target atom holds in *all* stable models of the program. Several algorithmic approaches have been proposed to compute cautious consequences in ASP, including model-checking strategies, minimality-based techniques, and methods based on unsatisfiable core extraction and backbone computation [34, 35, 36].

2.1.4 Relation with SAT solving

We now show that answer sets of $\mathsf{ASP}^{\mathsf{N}}$ programs can be computed by means of SAT solvers. In particular, for every ground $\mathsf{ASP}^{\mathsf{N}}$ program P, we show that there exists a formula $\phi_{as}(P)$ such that

an interpretation I is an answer set of P if and only if I is a model of $\phi_{as}(P)$. We follow the approach presented by Lin and Zhao [29]. Before stating the theorem that characterizes $\phi_{as}(P)$, we need to define the notions of *program completion* and *loop formulas*.

Definition 2.8 (Program completion) *Let P be a ground $\mathsf{ASP}^{\mathsf{N}}$ program. The completion of P, denoted as $\phi_{iff}(P)$, is the following Boolean formula:*

$$\phi_{iff}(P) \coloneqq \bigwedge_{a \in atoms(P)} \left(a \leftrightarrow \bigvee_{R \in P, head(R)=a} \left(\bigwedge_{b \in body^{+}(R)} b \wedge \bigwedge_{b \in body^{-}(R)} \neg b \right) \right).$$

.

A model of $\phi_{iff}(P)$ is called a *supported model*. It holds that every stable model of P is also a supported model of $\phi_{iff}(P)$, but the converse does not hold in general. In particular, there exist ground $\mathsf{ASP}^{\mathsf{N}}$ programs P for which an interpretation I satisfies $\phi_{iff}(P)$ but $I \notin AS(P)$.

The reason why a supported model may not be a stable model is the presence of loops. For example, consider the program P [29] defined as: `a :- b. b :- a.` In this example, atom `a` supports `b`, and vice versa. It is easy to verify that $AS(P) = \{\emptyset\}$, but the supported models of $\phi_{iff}(P)$ are $\{\emptyset, \{\mathtt{a}, \mathtt{b}\}\}$. The issue is that, under answer set semantics, loops like the one between `a` and `b` cannot support themselves: their justification must come from outside the loop. For instance, consider the extended program $P' \coloneqq P \cup \{\mathtt{a} \texttt{ :- } \mathtt{c}.\ \mathtt{c}.\}$. Here, the atom `c` externally supports the loop between `a` and `b`, and the only answer set of P' is $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$.

We now describe precisely what we mean by the word "loop".

Definition 2.9 (Positive atom dependency graph) *Let P be an ASP program. The positive atom dependency graph, denoted as $G(P)$, is defined as:*

$$G(P) \coloneqq (atoms(P), \{(b, h) \mid R \in P, b \in body^{+}(R), h \in head(R)\})$$

.

Definition 2.10 (Loops and tight programs) *Let P be an APS program, and let E be the set of arcs of the graph $G(P)$. A set of atoms L is a loop of P if it induces a non-trivial strongly connected subgraph of $G(P)$, namely, each pair of atoms in L is connected by a non-empty path in the graph $(L, E \cap (L \times L))$. We denote with $loops(P)$ the set of loops of P. We say that P is tight if P has no loops (i.e. $loops(P) = \emptyset$).*

For the class of ground tight $\mathsf{ASP}^{\mathsf{N}}$ programs, the formula $\phi_{iff}(\cdot)$ fully characterizes the answer set semantics, as shown by Fages [37, 38].

Theorem 2.11 (Fages' theorem [37]) *Let P be a tight ground $\mathsf{ASP}^{\mathsf{N}}$ program and let $I \subseteq atoms(P)$. It holds that $I \in AS(P)$ if and only if I is a model of $\phi_{iff}(P)$.*

To fill the gap between the completion formula and answer set semantics for non-tight programs, we describe the *loop formula* (denoted as $\phi_{loop}(P)$) which removes the possibility that loops support themselves.

Definition 2.12 (Loop formula) *Let P be a ground $\mathsf{ASP}^{\mathsf{N}}$ program. The loop formula of P, denoted as $\phi_{loop}(P)$, is defined as follows:*

$$\phi_{loop}(P) \coloneqq \bigwedge_{L \in loops(P)} \left(\left(\bigvee_{a \in L} a \right) \rightarrow \left(\bigvee_{R \in ES(L)} \left(\bigwedge_{b \in body^{+}(R)} b \wedge \bigwedge_{b \in body^{-}(R)} \neg b \right) \right) \right)$$

where $ES(L)$ is a set of rules called external support (with respect to the loop L) and is defined as :

$$ES(L) \coloneqq \left\{ R \in P \mid head(R) \in L, body^{+}(R) \cap L = \emptyset \right\}$$

Intuitively the external support of a loop L, is the set of rules that supports atoms in L without using atoms in L. The loop formula says that, for each loop L in P, if an atom in L is true, then it must be supported by the body of some rule in its external support. Notice that the loop formula ensures all atoms of a loop L to be false whenever L is not externally supported [2].

The following theorem from Lin and Zhao fully characterizes the answer set semantics using a Boolean formula.

Theorem 2.13 (Lin-Zhao Theorem [29]) *Let P be a ground $\mathsf{ASP}^{\mathsf{N}}$ program and let $I \subseteq atoms(P)$. It holds that $I \in AS(P)$ if and only if I is a model of $\phi_{iff}(P) \wedge \phi_{loop}(P)$.*

Through the rest of this work, we denote with $\phi_{as}(P)$ the formula $\phi_{iff}(P) \wedge \phi_{loop}(P)$. Note that the number of loops in a program P can be exponential in the size of P, and thus the formula $\phi_{as}(P)$ may grow exponentially. Lifschitz [39] in a paper entitled "*Why are there so many loop formulas?*" showed that, under a widely believed assumption in complexity theory ($\mathsf{P} \not\subseteq \mathsf{NC}^1/poly$, that is, not all languages in P can be recognized by polynomial size propositional formulas), there is no polynomial-size propositional encoding that fully captures the answer set semantics for all ground $\mathsf{ASP}^{\mathsf{N}}$ programs. This result formally explains why translations such as $\phi_{as}(P)$ may necessarily involve an exponential blow-up.

Despite this limitation, the translation to propositional logic is not only of theoretical interest, but also of practical relevance: it enables the use of SAT-based techniques and modern SAT solvers for computing answer sets. This connection underlies the design of conflict-driven ASP solvers, as demonstrated in the seminal work by Gebser et al. [40].

2.2 Inductive Logic Programming

After introducing the fundamental concepts and complexity results of Answer Set Programming (ASP), we now turn our attention to Inductive Logic Programming (ILP). In this section, we present several ILP frameworks based on answer set semantics and analyze the complexity of key reasoning tasks such as satisfiability, verification, and optimum verification. We also describe a naive algorithm for solving ILP tasks and briefly outline the ILASP1 algorithm. Most of the content in this section is adapted from [12].

2.2.1 Frameworks

We begin by defining a set of ILP frameworks based on answer set semantics. We do not cover more traditional settings such as *Learning from entailment* [41], *Learning from interpretations* [42], and *Learning from satisfiability* [43]. Each learning task considered in this work is defined by:

- a background knowledge B;
- a hypothesis space S_M;
- a set of positive examples E^+; and
- a set of negative examples E^-.

An inductive solution (usually denoted with H) is a subset of S_M satisfying the required conditions. Throughout the remainder of this thesis, we assume that B is an ASP^N program and that S_M is a set of ASP^N rules, that is, we focus on learning ASP programs without disjunctive heads. We say that a learning task T is ground if it uses only ground rules. Let X be an ILP learning framework and let T be a learning task. We say that T is am ILP_X task and we denote with $ILP_X(T)$ the set of solutions of T. The length of a solution H is the number of literals appearing in H. We denote with ${}^*ILP_X(T)$ the set of optimal inductive solutions of T, where optimality is defined in terms of the length of the hypotheses. We begin by introducing the brave induction framework.

Definition 2.14 (Brave Induction) *A brave induction (ILP_b) task T is a tuple $\langle B, S_M, E^+, E^- \rangle$ where E^+ and E^- are sets of ground atoms. A hypothesis $H \subseteq S_M$ is an inductive solution of T if and only if $\exists A \in AS(B \cup H)$ such that $E^+ \subseteq A$ and $E^- \cap A = \emptyset$.*

The cautious induction framework is similar except that existential quantification is replaced by universal quantification.

Definition 2.15 (Cautious Induction) *A cautious induction (ILP_c) task T is a tuple $\langle B, S_M, E^+, E^- \rangle$ where E^+ and E^- are sets of ground atoms. A hypothesis $H \subseteq S_M$ is an inductive solution of T if and only if*

1. *$AS(B \cup H) \neq \emptyset$;*
2. *$\forall A \in AS(B \cup H)$ it holds $E^+ \subseteq A$ and $E^- \cap A = \emptyset$.*

Both brave induction and cautious induction frameworks were first formalized by Sakama et al. [44]. Here, we defined the versions presented and refined by Law [12].

Cautious induction requires *all* answer sets to meet some condition while brave induction requires just *one* answer set to satisfy the requirements. We can generalize both frameworks by introducing a new parameter p that represents the fraction of answer sets that need to meet the required conditions. We define probabilistic induction.

Definition 2.16 (Probabilistic Induction) *A probabilistic induction (ILP_p) task T is a tuple $\langle p, B, S_M, E^+, E^- \rangle$ where $p \in [0, 1]$ and E^+ and E^- are sets of ground atoms. A hypothesis $H \subseteq S_M$ is an inductive solution of T if and only if the number of answer sets of $B \cup H$ satisfying $E^+ \subseteq A \wedge E^- \cap A = \emptyset$ divided by the total number of answer sets of $B \cup H$ is at least p.*

Notice that probabilistic induction corresponds to cautious induction when $p = 1$ and corresponds to brave induction when p goes near 0.

Law [12] observes that there are cases in which brave induction is too weak, while cautious induction is too strong. In particular, he presents a Sudoku example where neither an ILP_b nor an ILP_c task admits an inductive solution, meaning that these frameworks are unable to learn the standard rules of Sudoku. To address this issue, Law introduces a new framework called *Learning from Answer Sets* (LAS), which combines elements of both ILP_b and ILP_c. Before giving the definition of the LAS framework, we need to define *partial interpretations*.

Definition 2.17 (Partial interpretation) *A partial interpretation e is a pair of sets of ground atoms $\langle e^{incl}, e^{excl}\rangle$. An interpretation I extends e if and only if $e^{incl} \subseteq I$ and $e^{excl} \cap I = \emptyset$.*

The LAS framework represents examples as sets of partial interpretations. We now formally define the Learning From Answer Sets (LAS) framework.

Definition 2.18 (Learning From Answer Sets) *A learning from answer sets (ILP_{LAS}) task T is a tuple $\langle B, S_M, E^+, E^-\rangle$ where E^+ and E^- are sets of partial interpretations. A hypothesis $H \subseteq S_M$ is an inductive solution of T if and only if*

1. $\forall e^+ \in E^+ \; \exists A \in AS(B \cup H)$ *A extends e^+ ;*
2. $\forall e^- \in E^- \; \forall A \in AS(B \cup H)$ *A does not extend e^- .*

Before introducing further definitions, we present a simple example (taken from [12]) that will also be used in Chapter 3 to illustrate the output of the proposed encodings. Here, we define the task T_e and discuss its inductive solutions.

Running example: definition of the task T_e

Let $T_e = \langle B, S_M, E^+, E^-\rangle$ be the ILP_{LAS} task such that:

- $B := \{$`p :- not q.`$\}$
- $S_M := \{h_1$ `q.` $\quad h_2$ `q :- not p.`$\}$
- $E^+ := \{\langle \{p.\}, \{q.\}\rangle\}$
- $E^- := \{\langle \{q.\}, \{p.\}\rangle\}$

There are 4 possible subsets of the hypothesis space. For each subset H, we give the set of answer sets of $B \cup H$.

1. $AS(B \cup \emptyset) = \{1 : \{p\}\}$
2. $AS(B \cup \{h_1\}) = \{1 : \{q\}\}$
3. $AS(B \cup \{h_2\}) = \{1 : \{p\}, \;\; 2 : \{q\}\}$
4. $AS(B \cup \{h_1, h_2\}) = \{1 : \{q\}\}$

The hypotheses $H = \{h_1\}$ and $H = \{h_1, h_2\}$ are not inductive solutions, as the only answer set of $B \cup H$ extends the negative example. Moreover, $H = \{h_2\}$ is also not an inductive solution: while the answer set containing only the atom `p` does not extend the negative example, the answer set containing only the atom `q` does, contradicting the definition of an inductive solution. Thus, we conclude that $H = \emptyset$ is the only inductive solution of T_e. In this case, we say that we cannot learn any rule for task T_e using LAS semantics.
Consequently, we examine a slightly altered example where we can successfully learn a non-empty hypothesis. Define T_e^+ as the version of T_e including only its positive example and excluding any negative example. Given that $H = \emptyset$ is an inductive solution of T_e, it is also an inductive solution of T_e^+. Furthermore, $H = \{h2\}$ is an inductive solution for T_e^+, as the first answer set of $B \cup \{h_2\}$ extends the positive example, containing `p` but excluding `q`.

Law [12] also introduces a framework called *Context-Dependent Learning from Ordered Answer Sets*, here we present a simplification that we call *Context-Dependent Learning from Answer Sets*. The difference between the LAS frameworks is that examples are associated to a context. We first introduce *context-dependent partial interpretations.*

Definition 2.19 (Context-dependent partial interpretation) *A context-dependent partial interpretation (CDPI) e is an object of type $\langle\langle e^{incl}, e^{excl}\rangle, e^{ctx}\rangle$ where $\langle e^{incl}, e^{excl}\rangle$ is a partial interpretation and e^{ctx} is an $\mathsf{ASP}^{\mathsf{N}}$ program called context. Given a program P, an interpretation I is an accepting answer set of e (with respect to P) if and only if $I \in AS(P \cup e^{ctx})$ and I extends $\langle e^{incl}, e^{excl}\rangle$. A program P accepts e if there is at least one accepting answer set of e w.r.t. to P.*

The Context-Dependent Learning from Answer Sets framework is defined as follows.

Definition 2.20 (Context-Dependent Learning from Answer Sets) *A context-dependent learning from answer sets (ILP^{ctx}_{LAS}) task T is a tuple $\langle B, S_M, E^+, E^-\rangle$ where E^+ and E^- are sets of CDPI. A hypothesis $H \subseteq S_M$ is an inductive solution of T if and only if*

1. *$\forall e^+ \in E^+$ $B \cup H$ A accepts e^+ ;*

2. *$\forall e^- \in E^-$ $B \cup H$ A does not accept e^- .*

Notice that conditions 1. and 2. can be expanded as:

1. $\forall\ \langle\langle e^{incl}, e^{excl}\rangle, e^{ctx}\rangle \in E^+\ \exists I \in AS(B \cup H \cup e^{ctx})$ A extends $\langle e^{incl}, e^{excl}\rangle$;

2. $\forall\ \langle\langle e^{incl}, e^{excl}\rangle, e^{ctx}\rangle \in E^-\ \forall I \in AS(B \cup H \cup e^{ctx})$ A does not extend $\langle e^{incl}, e^{excl}\rangle$.

Notably, Law [12] proves that ILP^{ctx}_{LAS} is as expressive as ILP_{LAS} and so the context does not increase expressiveness but just provides a more natural way to write tasks.

Theorem 2.21 ([12]) *Let T be an ILP^{ctx}_{LAS} task. Then, there exists an ILP_{LAS} task T' such that $ILP^{ctx}_{LAS}(T) = ILP_{LAS}(T')$.*

Proof. Let $T = \langle B_1, S_M, E_1^+, E_1^- \rangle$ be an ILP^{ctx}_{LAS} task. We begin by introducing some auxiliary definitions to translate ILP^{ctx}_{LAS} tasks into ILP_{LAS} tasks. Let $e_i \in E_1^+ \cup E_1^-$ be the CDPI $e_i = \langle \langle e_i^{incl}, e_i^{excl} \rangle, e_i^{ctx} \rangle$, indexed by i. We define $pi(e_i)$ as the partial interpretation $\langle e_i^{incl} \cup \{\texttt{ctx(e}_\texttt{i}\texttt{)}\},\ e_i^{excl} \rangle$, where `ctx` is a fresh predicate symbol not occurring in T and $\texttt{e}_\texttt{i}$ is a constant. Let P be a logic program and let a be an atom. We define:

$$activate(P, a) := \{head(R)\ \texttt{:-}\ body(R),\ a. \mid R \in P\}$$

That is, $activate(P, a)$ is the program P where each rule is "guarded" by the atom a.

We now define the ILP_{LAS} task $T' = \langle B_2, S_M, E_2^+, E_2^- \rangle$ as follows:

- $B_2 = B_1 \cup \bigcup_{e_i \in E_1^+ \cup E_1^-} activate(e_i^{ctx}, \texttt{ctx(e}_\texttt{i}\texttt{)}) \ \cup\ \{1\{\texttt{ctx(e}_\texttt{1}\texttt{)}; \ldots; \texttt{ctx(e}_\texttt{n}\texttt{)}\}1.\}$
  where $\{e_1, \ldots, e_n\} = E_1^+ \cup E_1^-$;
- $E_2^+ = \{pi(e) \mid e \in E_1^+\}$;
- $E_2^- = \{pi(e) \mid e \in E_1^-\}$.

It is now a straightforward verification that $ILP^{ctx}_{LAS}(T) = ILP_{LAS}(T')$. □

The intuition behind the translation from ILP^{ctx}_{LAS} to ILP_{LAS} is that the context of each example is moved into the background knowledge. Each rule in the context e_i^{ctx} of a CDPI e_i is guarded by the atom $\texttt{ctx}(e_i)$, which ensures that the rule is active only when the corresponding context is selected. The background knowledge includes a choice rule enforcing that exactly one context is active at a time. Each CDPI is then converted into a standard partial interpretation and we also add $\texttt{ctx}(e_i)$ to its inclusion set, so that only the relevant context is considered during evaluation.

Another result proven by Law [12] is that the LAS framework is strictly more general than the previously defined frameworks. In particular, he introduces three notions of generality and shows that: (i) ILP_b and ILP_c are incomparable, and (ii) ILP_{LAS} is strictly more general than both ILP_b and ILP_c under all three measures. This result provides a formal justification for preferring ILP_{LAS} over ILP_b and ILP_c in general settings.

We conclude this section by proposing a new ILP framework, which we call *probabilistic Learning from Answer Sets* (pLAS). Although we do not formally analyze or employ this framework in the remainder of the thesis, we believe it offers a natural generalization of both ILP_{LAS} and ILP_p.

Definition 2.22 (Probabilistic Learning From Answer Sets) *A probabilistic learning from answer sets (ILP_{pLAS}) task T is a tuple $\langle B, S_M, E \rangle$, where E is a set of pairs $\langle p_i, e_i \rangle$ such that $p_i \in [0, 1]$ and e_i is a partial interpretation. A hypothesis $H \subseteq S_M$ is an inductive solution of T if and only if, for each example $\langle p_i, e_i \rangle \in E$, the number of answer sets of $B \cup H$ that extend e_i, divided by the total number of answer sets of $B \cup H$, is at least p_i.*

The justification for pLAS is as follows. The framework ILP_p generalizes ILP_b and ILP_c by introducing a global threshold $p \in [0, 1]$, representing the minimum fraction of answer sets in which all examples must be satisfied. Values close to 0 correspond to brave reasoning, while $p = 1$ corresponds to cautious reasoning. In contrast, ILP_{LAS} generalizes ILP_b and ILP_c by distinguishing between positive

and negative examples, interpreted respectively under brave and cautious semantics. This allows each example to be evaluated with respect to either a single answer set or all of them. The ILP_{pLAS} framework combines both generalizations: for each example e_i, we can specify a threshold p_i indicating the minimum fraction of answer sets in which e_i must be satisfied. This allows for fine-grained control over the inductive constraints associated with each example.

We leave the formal investigation of ILP_{pLAS} as a direction for future work. In particular, it would be interesting to analyze its position within the formal measures of generality introduced by Law [12].

2.2.2 Complexity

We now consider the complexity of various problems related to ground ILP_b, ILP_c, and ILP_{LAS} tasks. In particular, we will study the satisfiability problem for ILP_{LAS} in greater detail, as it will be the focus of the encodings proposed in Chapter 3. Let T be an ILP task and let $H \subseteq S_M$ be a candidate solution.

- The *verification problem* asks whether H is an inductive solution of T.
- The *satisfiability problem* asks whether T admits at least one inductive solution.
- The *optimum verification problem* asks whether H is an optimal inductive solution of T.

In this section, we show that the satisfiability problem for ground ILP_{LAS} tasks is Σ_2^P-complete. To this end, (i) we prove that the satisfiability problem for ILP_c can be polynomially reduced to that of ILP_{LAS}, (ii) we establish that satisfiability for ILP_c is Σ_2^P-hard and finally (iii) we prove that satisfiability problem for ground ILP_{LAS} tasks is a member of Σ_2^P.

Theorem 2.23 ([12]) *Deciding verification, satisfiability and optimum verification for ILP_c each reduce polynomially to the corresponding ILP_{LAS} decision problem.*

Proof. The proof consists in translating ILP_c tasks into ILP_{LAS} tasks. Let $T = \langle H, S_M, E^+, E^- \rangle$ be an ILP_c task. We show that there exists an ILP_{LAS} task $M(T)$ such that H is an inductive solution of T if and only if H is an inductive solution of $M(T)$. We define $M(T)$ as follows:

$$M(T) \coloneqq \langle B, S_M, \{\langle \emptyset, \emptyset \rangle\}, \{\langle \emptyset, \{e^+\} \rangle \mid e^+ \in E^+\} \cup \{\langle \{e^-\}, \emptyset \rangle \mid e^- \in E^-\} \rangle.$$

Intuitively, the single positive example in $M(T)$ ensures that the program $B \cup H$ is satisfiable, while the positive and negative examples of T are encoded as negative examples in $M(T)$. It is then a straightforward calculation to verify that H is an inductive solution of T if and only if it is an inductive solution of $M(T)$, which concludes the proof. □

We now establish the Σ_2^P-hardness for ILP_c. The proof is based on theorem 2.7 which establishes that consistency checking for ground ASP^D programs is Σ_2^P-complete.

Theorem 2.24 ([12]) *Deciding satisfiability for ground ILP_c tasks is Σ_2^P-hard.*

Proof. The proof consists of a reduction from the consistency problem of a ground ASP^D program to the satisfiability problem of a ground ILP_c task. Let P be a ground ASP^D program. We define an ILP_c task $T(P)$ such that $T(P)$ has an inductive solution if and only if the program P has an answer set.

First, we consider the program P', which is obtained as a copy of P where the following transformations are applied:

- Every disjunction in rule heads of the form $\texttt{h}_1$ `|` ... `|` $\texttt{h}_\texttt{m}$ is replaced with a choice rule $\texttt{1}\{\texttt{h}_1\texttt{;...;h}_\texttt{m}\}\texttt{m}$.
- Every negative literal `not a` is replaced with an atom embedded in a new predicate `not inAs(a).`

That is, disjunctions in rule heads are rewritten as choice rules, and negative literals are encapsulated within the new predicate `inAs`.

Now, we define the task $T(P) = \{B, S_M, E^+, E^-\}$ as follows:

- $B = P' \cup \{$`:- a, not inAs(a).  notMin :- not a, inAs(a).` $\mid a \in HB_P\}$;
- $S_M = \{$`inAs(a).` $\mid a \in HB_P\}$;
- $E^+ = \emptyset$, and $E^- = \{$`notMin.`$\}$

We now establish some useful properties of the reduction.

Fact 2.25 *The following properties hold:*

1. *If P is satisfiable, then so is P'.*
2. *If there exists $A \neq \emptyset$ such that $A \in AS(P')$, then B is not satisfiable.*
3. *$B \cup H$ is satisfiable if and only if there exists $A \in AS(P'^H)$ such that $A \subseteq \{$`a` $\mid$ `inAs(a)` $\in H\}$.*

Proof of the facts. The first fact is immediate. For the second fact, observe that no rule in B provides support for any atom of the form `inAs(a)`. Consequently, if an atom `a` is derivable, then the constraint `:- a, not inAs(a).` is violated, making B unsatisfiable. Finally, we prove the third fact. Suppose that $B \cup H$ is satisfiable, and let A be an answer set of P'^H. In $B \cup H$, the constraints of the form `:- a, not inAs(a).` are not violated, implying that for every `a` $\in A$, it must hold that `inAs(a)` $\in H$. The converse follows similarly. Additionally, observe that $A' \in AS(B \cup H)$ if and only if $A' = A \cup H \cup M$ where $M = \{$`notMin`$\}$ if there exists `inAs(a)` $\in H$ such that `a` $\notin A$, and $M = \emptyset$ otherwise.

We now return to the main proof. It holds that:

$T(P)$ has a solution

$\leftrightarrow^1$ $\exists H \subseteq S_M$ such that $B \cup H$ is satisfiable **and** $\nexists A' \in AS(B \cup H)$ such that `notMin` $\in A'$

$\leftrightarrow^2$ $\exists H \subseteq S_M$ such that $\exists A \in AS(P'^H)$ with $A \subseteq \{$`a` $\mid$ `inAs(a)` $\in H\}$ **and** $\nexists A' \in AS(B \cup H)$ such that `notMin` $\in A'$

$\leftrightarrow^3$ $\exists H \subseteq S_M$ such that $\exists A \in AS(P'^H)$ with $A = \{$`a` $\mid$ `inAs(a)` $\in H\}$ **and** there is no strict subset of A which is also an answer set

$\leftrightarrow^4$ $\exists H \subseteq S_M$ such that $\{$`a` $\mid$ `inAs(a)` $\in H\}$ is a **minimal model** of P'^H

$\leftrightarrow^5$ $\exists A \subseteq HB_P$ such that A is a minimal model of P^A

$\leftrightarrow^6$ P is satisfiable .

The first equivalence $\leftrightarrow^1$ holds by the definition of an inductive solution to an ILP_c task. The second equivalence $\leftrightarrow^2$ follows from point 3 of Fact 2.25. The third equivalence $\leftrightarrow^3$ is justified by the characterization of answer sets of $B \cup H$ as $A' = A \cup H \cup M$, where M accounts for the presence or absence of `notMin`. Equivalences $\leftrightarrow^4$, $\leftrightarrow^5$, and $\leftrightarrow^6$ follow directly from the definitions of minimal model and answer set. This concludes the proof. □

Finally, we establish the Σ_2^P membership of the satisfiability problem for ground ILP_{LAS} tasks. This result was originally proved by Law [12]; here, however, we present a reformulated — and arguably simpler — proof.

Theorem 2.26 *Deciding satisfiability for ground ILP_{LAS} tasks is in Σ_2^P.*

Proof. Let $T = \langle B, S_M, E^+, E^- \rangle$ be a ground ILP_{LAS} task, where each example $e \in E^+ \cup E^-$ has the form $e = \langle \{e_1^{incl}, \ldots, e_n^{incl}\}, \{e_1^{excl}, \ldots, e_m^{excl}\} \rangle$. Given a candidate hypothesis $H \subseteq S_M$, verifying whether H is an inductive solution requires checking two conditions. First, we must determine if there exists at least one answer set of $B \cup H$ that satisfies the formula:

$$\bigwedge_{e \in E^+} \left(e_1^{incl} \wedge \cdots \wedge e_n^{incl} \wedge \neg e_1^{excl} \wedge \cdots \wedge \neg e_m^{excl} \right).$$

Second, we must verify that all answer sets of $B \cup H$ satisfy the formula:

$$\bigwedge_{e \in E^-} \left(\neg e_1^{incl} \vee \cdots \vee \neg e_n^{incl} \vee e_1^{excl} \vee \cdots \vee e_m^{excl} \right).$$

These checks correspond, respectively, to solving $poly(|E^+|)$ brave entailment problems and $poly(|E^-|)$ cautious entailment problems for the ground $\mathsf{ASP}^{\mathsf{N}}$ program $B \cup H$, each of which can be solved using an NP oracle. Since there are $|S_M|$ non-deterministic choices for guessing the hypothesis H, we conclude that the satisfiability problem for ILP_{LAS} tasks belongs to the complexity class Σ_2^P. □

Theorems 2.23, 2.24 and 2.26 allow us to conclude that the satisfiability problem for ground ILP_{LAS} tasks is Σ_2^P-complete. We do not provide formal proofs for the complexity of the other fundamental ILP problems, but we summarize the known results in Table 2.2. The table reports the complexity classes for verification, satisfiability, and optimum verification in the case of ground ILP_b, ILP_c, and ILP_{LAS} frameworks [12]. As expected, the complexity results for ILP_c align with those for ILP_{LAS}, while the corresponding problems for ILP_b are consistently easier. This is analogous to what we observed in Table 2.1, where cautious reasoning in ASP is computationally harder than brave reasoning. The reason is structural: cautious reasoning requires universal quantification over answer sets, whereas brave reasoning requires only existential quantification.

In Chapter 3, we will address the satisfiability problem for ground ILP_{LAS} tasks by encoding it into ASP programs. In particular, we will propose a polynomial-size encoding using disjunctive ASP programs ($\mathsf{ASP}^{\mathsf{D}}$). In a sense, this construction goes in the "opposite direction" of theorem 2.24, which reduces consistency checking for ground $\mathsf{ASP}^{\mathsf{D}}$ programs to the satisfiability of ILP_c tasks. Unsurprisingly, the set of negative examples — which enforces cautious entailment — plays a central role in this reduction, as it is essential to achieve the target Σ_2^P complexity.

	Verification	Satisfiability	Optimum Verification
ILP_b	NP-complete	NP-complete	DP-complete
ILP_c	DP-complete	Σ_2^P-complete	Π_2^P-complete
ILP_{LAS}	DP-complete	Σ_2^P-complete	Π_2^P-complete

Table 2.2: Recap of problem complexities for various ILP frameworks [12].

2.2.3 Naive algorithms

In this section, we present naive algorithms for computing an inductive solution to ILP_b, ILP_c, and ILP_{LAS} tasks. It is worth noting that the proposed algorithms are applicable to both ground and non-ground tasks. The algorithms rely on a brute-force enumeration of all possible hypotheses $H \subseteq S_M$, verifying for each whether it satisfies the required semantic conditions. We do not provide explicit algorithms for ILP_c, ILP^{ctx}_{LAS}, and ILP_{pLAS}, as they are straightforward generalizations of the ones presented here.

Algorithm 1 Naive Algorithm for ILP_b

Input: $T = \langle B, S_M, E^+, E^- \rangle$
Output: An inductive solution $H \subseteq S_M$
for $H \subseteq S_M$ **do**
  $E' \leftarrow \{$ `:- not` $A \mid A \in E^+\} \cup \{$ `:-` $A \mid A \in E^-\}$
  if $AS(B \cup H \cup E') \neq \phi$ **then**
    return H
  end if
end for
return False

Algorithm 2 Naive Algorithm for ILP_c

Input: $T = \langle B, S_M, E^+, E^- \rangle$
Output: An inductive solution $H \subseteq S_M$
for $H \subseteq S_M$ **do**
  $AS \leftarrow AS(B \cup H)$
  $i \leftarrow 0$
  for $A \in AS$ **do**
    if $E^+ \not\subseteq A \vee E^- \cap A \neq \emptyset$ **then**
      break
    end if
    $i \leftarrow i + 1$
  end for
  if $i == |AS|$ **then**
    return H
  end if
end for
return False

Algorithm 1 describes the computation of an inductive solution for an ILP_b task. Given a candidate hypothesis $H \subseteq S_M$, we check the consistency of the program $B \cup H \cup E'$, where E' is an auxiliary ASP program that enforces the condition that all atoms in E^+ must be supported, and no atom in E^- may be supported. In other words, E' encodes the requirement that there must exist an answer set in which

Algorithm 3 Naive Algorithm for ILP_{LAS}

Input: $T = \langle B, S_M, E^+, E^- \rangle$
Output: An inductive solution $H \subseteq S_M$
for $H \subseteq S_M$ **do**
  $AS \leftarrow AS(B \cup H)$
  $brave \leftarrow array[\ |E^+|\ ]$
  for $A \in AS$ **do**
    for $e_j^- \in E^-$ **do**
      if $B \cup H$ *extends* e_j^- **then**
        return False
      end if
    end for
    for $e_i^+ \in E^+$ **do**
      if $B \cup H$ *extends* e_i^+ **then**
        $brave[i] \leftarrow True$
      end if
    end for
  end for
  if $\forall i\ brave[i] == True$ **then**
    return H
  end if
end for
return False

all positive examples appear and all negative examples are absent.

Algorithm 2 describes the computation of an inductive solution for an ILP_c task. Given a candidate hypothesis $H \subseteq S_M$, we enumerate all the possible answer sets of $B \cup H$ and if one of them does not meet the condition $E^+ \subseteq A \wedge E^- \cap A = \emptyset$ we discard H and we consider the next hypothesis (until there is one). Otherwise if all the answer sets of $B \cup H$ satisfy $E^+ \subseteq A \wedge E^- \cap A = \emptyset$ we conclude H is an inductive solution.

Finally, Algorithm 3 describes the computation of an inductive solution for an ILP_{LAS} task. It is easy to see that this algorithm generalizes both Algorithm 1 and Algorithm 2, as it incorporates elements of both brave and cautious induction.

It is interesting to notice that if we aim to compute all inductive solutions of an ILP_{LAS} task $T = \langle B, S_M, E^+, E^- \rangle$, we can proceed in two steps. First, we consider the simplified task $T' = \langle B, S_M, E^+, \emptyset \rangle$, which includes only positive examples, and compute the set of its inductive solutions, denoted $ILP_{LAS}(T')$. Then, we define a new task $T'' = \langle B, ILP_{LAS}(T'), \emptyset, E^- \rangle$, which includes only negative examples and uses the solutions from T' as its hypothesis space. The inductive solutions of T'' correspond exactly to the set of all inductive solutions of the original task T.

Before concluding and moving on to the description of ILASP in the next subsection, we remark that the algorithms presented here compute generic inductive solutions, but not necessarily optimal ones. Moreover, they do not apply any form of reasoning or optimization; instead, they rely purely on brute-force techniques.

2.2.4 ILASP

We conclude the background chapter by providing a high-level overview of the ILASP1 algorithm and discussing the main optimizations introduced in ILASP2, ILASP2i, ILASP3, and ILASP4.

ILASP1 is based on the notions of *positive* and *violating* hypotheses.

Definition 2.27 (Positive and Violating Hypotheses [12]) *Let $T = \langle B, S_M, E^+, E^- \rangle$ be an ILP_{LAS} task. We say that $H \subseteq S_M$ is a* positive hypothesis *if $\forall e \in E^+$ there exists $A \in AS(B \cup H)$ such that A extends e. A positive hypothesis H is said to be* violating *if $\exists e \in E^-$ such that there exists $A \in AS(B \cup H)$ extending e. We denote with $P(T)$ (respectively, $V(T)$) the set of positive (respectively, violating) hypotheses for task T. We write $P(T)^n$ and $V(T)^n$ to denote the subsets of positive and violating hypotheses of length n.*

Law [12] shows that the set of inductive solutions of a task is exactly the set of positive hypotheses that are not violating.

Theorem 2.28 ([12]) *Let T be an ILP_{LAS} task. Then: $ILP_{LAS}(T) = P(T) \setminus V(T)$*

Algorithm 4 ILASP1 for ILP_{LAS}

Input: $T = \langle B, S_M, E^+, E^- \rangle$
Output: The set ${}^*ILP_{LAS}(T)$
$solutions \leftarrow \emptyset$
for $n \leftarrow 0$ **up to** n_{max} **do**
  $AS_1 \leftarrow AS(P^n_{ilasp}(T) \cup \{\texttt{checkViolating}\})$
  $vs \leftarrow \{\{\texttt{h(i)} \mid \texttt{h(i)} \in A\} \mid A \in AS_1\}$
  $AS_2 \leftarrow AS(P^n_{ilasp}(T) \cup \{constraint(V) \mid V \in vs\})$
  $solutions \leftarrow \{\{\texttt{h(i)} \mid \texttt{h(i)} \in A\} \mid A \in AS_2\}$
  if $solutions \neq \emptyset$ **then**
    break
  end if
end for
return *solutions*

Algorithm 4 implements the ILASP1 algorithm. The goal is to compute the set of optimal inductive solutions by incrementally exploring candidate hypotheses of increasing size, starting from $n = 0$. The first time a non-empty set of solutions is found, it is returned. Law [12] proves that this set corresponds exactly to ${}^*ILP_{LAS}(T)$. The algorithm relies on an ASP program P^n_{ilasp} that encodes the task T at hypothesis size n. For each rule $h_i \in S_M$, P^n_{ilasp} includes a choice rule of the form $0\{\texttt{h(i)}\}1$, allowing each rule to be optionally included in a candidate hypothesis. For every example $e \in E^+ \cup E^-$, the program contains an atom `cov(e, j)`, which is supported if the answer set (labelled by j) of $B \cup H$ extends e. For positive examples $e \in E^+$, P^n_{ilasp} includes constraints of the form `:- not cov(e, j)`, enforcing that each such example must be covered. In contrast, if a negative example $e \in E^-$ is covered by some answer set of $B \cup H$ and the atom `checkViolating` is present, then the current hypothesis (represented by the set of atoms `h(i)` selected) is marked as violating. The program P^n_{ilasp} is first used together with the fact `checkViolating` to compute the set vs of violating hypotheses. Then, in a second run, the program is used again—this time with added constraints of the form: $\texttt{constraint}(V) :=$ `:- not` $\texttt{h(i}_1\texttt{)}, \ldots, \texttt{h(i}_n\texttt{)}$

for each violating hypothesis $V = \{h_{i_1}, \ldots, h_{i_n}\} \in vs$. This second step filters out violating hypotheses, effectively computing $P(T)^n \setminus V(T)$. Having described ILASP1 in detail, we now briefly outline the main improvements introduced in subsequent versions of the ILASP algorithm.

ILASP2 improves upon ILASP1 by introducing the notion of violating reasons, which enables more compact and efficient ASP encodings. Both ILASP1 and ILASP2 are examples of batch learners, meaning they consider all examples simultaneously. ILASP2i introduces an iterative approach, where only a subset of examples (called relevant examples) is considered at each step. This refinement often leads to a significant reduction in the size of the ASP grounding.

ILASP3 targets learning in the presence of noisy examples by incrementally constructing a function that approximates the coverage of hypotheses.

Finally, ILASP4 formalizes Conflict-Driven Inductive Logic Programming (CDILP). It is an iterative procedure composed of four main steps: Hypothesis Search, Counterexample Search, Conflict Analysis, and Constraint Propagation.

In the next chapter, we move from existing ILP algorithms to our proposed encodings. In particular, we present two ASP-based translations of the satisfiability problem for ILP_{LAS} tasks, one based on exponential $\mathsf{ASP}^{\mathsf{N}}$ programs and one based on $\mathsf{ASP}^{\mathsf{D}}$ programs of polynomial size.

3

Contributions and Methods

In this chapter, we present our main theoretical contributions. We show that for any ground task T, there exist two ASP encodings: an $\mathsf{ASP}^{\mathsf{N}}$ program $P_{exp}(T)$ of size exponential in $|T|$, and an $\mathsf{ASP}^{\mathsf{D}}$ program $P_{dis}(T)$ of size polynomial in $|T|$, provided that the combined program $B \cup S_M$ is tight. Here, $|T|$ denotes the size of the task $T = \langle B, S_M, E^+, E^- \rangle$, measured as the sum of the sizes of B, S_M, E^+, and E^-.

Both encodings are constructed in such a way that computing their answer sets allows us to recover the set of inductive solutions $ILP_{LAS}(T)$.

Each of these programs translates the ILP task T into the ASP formalism. For both encodings, we prove a correctness theorem, stating that every answer set of the encoding corresponds to an inductive solution of the original task. We also prove a completeness theorem, showing that every inductive solution is captured by some answer set of the encoding.

In addition, we analyze the size and structural complexity of both encodings. Finally, we extend our approach to compute optimal inductive solutions with respect to the size of the hypothesis and we show how to support non ground tasks.

3.1 Exponential $\mathsf{ASP}^{\mathsf{N}}$ encoding

Let $T = \langle B, S_M, E^+, E^- \rangle$ be a ground ILP_{LAS} task. We define an $\mathsf{ASP}^{\mathsf{N}}$ program $P_{exp}(T)$, such that $P_{exp}(T)$ is satisfiable if and only if T is satisfiable, and the set $AS(P_{exp}(T))$ corresponds one-to-one with the set of solutions of T. First, we present the encoding and the main ideas, followed the correctness and completeness theorems.

3.1.1 Encoding

As discussed in section 2.2.4, the ILASP1 algorithm iteratively solves an $\mathsf{ASP}^{\mathsf{N}}$ program P^n_{ilasp} (of polynomial size in $|T|$), which can only capture brave entailment. The main idea behind the definition of P_{exp} is to modify P^n_{ilasp} to capture cautious entailment. This modification eliminates the loop of ILASP1, allowing task T to be solved directly by solving the ASP program $P_{exp}(T)$. Specifically, this is achieved by enumerating all possible answer sets of $B \cup H$ inside $P_{exp}(T)$. Since the number of answer sets is typically exponential, the program $P_{exp}(T)$ will require exponential space. The program $P_{exp}(T)$

can be viewed as a *guess-and-verify* method: we guess a solution H, and then verify that H satisfies both the brave and cautious requirements, allowing us to conclude that H is an inductive solution. To enumerate all possible answer sets of $B \cup H$, we first enumerate all possible interpretations of $B \cup S_M$ and then check which of them are answer sets. To this end, we need to simulate the notion of reduct and so we encapsulate each negative literal from $B \cup S_M$ within a predicate `reduct(` $\cdot$ `)`. Given an interpretation M, the *answer set checking* phase computes the reduct $(B \cup H)^M$ and checks if it is the minimal Herbrand model of $(B \cup H)$. If this is the case, we mark the interpretation as `isAS(`M`)`; otherwise, we mark it as `notAS(`M`)`. At this point, it is straightforward to encode the notion of a covered example into the program and constrain the solutions of $P_{exp}(T)$ to include only hypotheses H such that each positive example is bravely covered by $B \cup H$, and all negative examples are cautiously not covered by $B \cup H$.

We now present how to define the program $P_{exp}(T)$. The encoding is structured into the following modules: *background knowledge*, *hypothesis space*, *interpretations enumeration*, *answer set checking* and *examples*. We now describe each of these modules in detail.

Background knowledge

For each rule $R_i \in B$ of the background knowledge of the form:

$$H \texttt{ :- } A_1, \ldots, A_n, \texttt{not } B_1, \ldots, \texttt{not } B_m.$$

the program $P_{exp}(T)$ contains the rule:

$$\texttt{inAs}(H, M) \texttt{ :- } \texttt{inAs}(A_1, M), \ldots, \texttt{inAs}(A_n, M), \quad (Bg(i))$$
$$\texttt{reduct}(\texttt{ninAs}(B_1, M)), \ldots, \texttt{reduct}(\texttt{ninAs}(B_m, M)), \texttt{interpretation}(M).$$

Hypothesis space

Let $S_M = \{h_1, \ldots, h_s\}$ be a finite hypothesis space. The program $P_{exp}(T)$ contains a choice rule over the hypothesis:

$$\texttt{0\{h(1);...;h(s)\}s.} \quad (Hypo)$$

For each hypothesis $h_i \in S_M$ in the hypothesis space of the form:

$$H \texttt{ :- } A_1, \ldots, A_n, \texttt{not } B_1, \ldots \texttt{not } B_m.$$

the program $P_{exp}(T)$ contains the rule:

$$\texttt{inAs}(H, M) \texttt{ :- } \texttt{inAs}(A_1, M), \ldots, \texttt{inAs}(A_n, M), \quad (Hypo(i))$$
$$\texttt{reduct}(\texttt{ninAs}(B_1, M)), \ldots, \texttt{reduct}(\texttt{ninAs}(B_m, M)), \texttt{interpretation}(M), \texttt{h(i)}.$$

The choice rule *Hypo* encodes the fact that each candidate answer set of $P_{exp}(T)$ corresponds to exactly one hypothesis $H \subseteq S_M$. The rule $Hypo(i)$ is similar to $Bg(i)$, except that $Hypo(i)$ is activated only if

`h(i)` appears in the answer set. As mentioned in the introduction of this section, both rules $Bg(i)$ and $Hypo(i)$ do not contain negative literals but instead include predicates that facilitate the computation of the reduct of $B \cup H$ with respect to an interpretation M.

Interpretations enumeration

For a learning task T, the program $P_{exp}(T)$ enumerates all possible interpretations of $B \cup S_M$ and checks which ones are stable models of $B \cup H$. Let HB be the Herbrand base of $B \cup S_M$. Consider an enumeration of all the possible $2^{|HB|}$ interpretations, and let M_j be the interpretation of index j. The program $P_{exp}(T)$ contains the following atoms:

$$\texttt{interpretation}(j). \qquad (Int_1(j))$$

$$\texttt{inInterpretation}(p, j). \qquad \forall p \in M_j \qquad (Int_2(j))$$

$$\texttt{ninInterpretation}(p, j). \qquad \forall p \notin M_j \qquad (Int_3(j))$$

Answer set checking

Let M be an interpretation, the *answer set checking* stage consists of checking whether or not M is a stable model of $B \cup H$. To this end, we include in $P_{exp}(T)$ the following rules:

$$\texttt{:- reduct(ninAs}(X, M)\texttt{), inInterpretation}(X, M)\texttt{, interpretation}(M)\texttt{.} \qquad (As_1)$$

$$\texttt{reduct(ninAs}(X, M)\texttt{) :- ninInterpretation}(X, M)\texttt{, interpretation}(M)\texttt{.} \qquad (As_2)$$

$$\texttt{notAS}(M)\texttt{ :- inAs}(X, M)\texttt{, ninInterpretation}(X, M)\texttt{, interpretation}(M)\texttt{.} \qquad (As_3)$$

$$\texttt{notAS}(M)\texttt{ :- not inAs}(X, M)\texttt{, inInterpretation}(X, M)\texttt{, interpretation}(M)\texttt{.} \qquad (As_4)$$

$$\texttt{isAS}(M)\texttt{ :- not notAS}(M)\texttt{, interpretation}(M)\texttt{.} \qquad (As_5)$$

Recall that the interpretation M_j is encoded by predicates of the form `inInterpretation(`$\cdot, j$`)` and `ninInterpretation(`$\cdot, j$`)`. Rules As_1 and As_2 are designed to simulate the computation of the reduct $(B \cup H)^{M_j}$. Let R_i be a rule in $B \cup H$; in particular, consider the case where $R_i \in B$ (the case $R_i \in H$ is analogous). Suppose that the rule R_i contains a negative literal `not p` in its body. In the encoding of R_i (i.e., in rule $Bg(i)$), the literal `not p` is replaced by `reduct(ninAs(`p, j`))`. We now analyze two cases:

Case $p \in M_j$. By the definition of the reduct, the rule R_i must not appear in $(B \cup H)^{M_j}$ and therefore does not contribute to $M_{(B \cup H)^{M_j}}$, the minimal Herbrand model of $(B \cup H)^{M_j}$. Since `inInterpretation(`p, j`)` holds, rule As_1 ensures that `reduct(ninAs(`p, j`))` is false. Consequently, rule $Bg(i)$ is discarded whenever `interpretation(`j`)` holds (i.e., whenever we consider the interpretation M_j).

Case $p \notin M_j$. By the definition of the reduct, the literal `not p` must be removed from the body of R_i in $(B \cup H)^{M_j}$. Since `ninInterpretation(`p, j`)` holds, rule As_2 ensures that `reduct(ninAs(`p, j`))` is true whenever `interpretation(`j`)` holds.

Rules As_3, As_4, and As_5 verify whether the minimal Herbrand model of $B \cup H$ (encoded by literals `inAs(`$\cdot, j$`)`) coincides with the interpretation M (encoded by literals `inInterpretation(`$\cdot, j$`)`). If they match, the atom `isAS(`j`)` is set; otherwise, the atom `notAS(`j`)` is set.

Lemma 3.1 in section 3.1.2 will formalize these intuitions.

Examples

Let $e_i \in E^+ \cup E^-$ be an example of the form $\langle \{e_1^{incl}, \ldots, e_n^{incl}\}, \{e_1^{excl}, \ldots, e_m^{excl}\}\rangle$. The program $P_{exp}(T)$ includes the following rule:

$$\texttt{cov(}i\texttt{) :- inInterpretation(}e_1^{incl}\texttt{,}M\texttt{),}\ldots\texttt{,inInterpretation(}e_n^{incl}\texttt{,}M\texttt{),} \quad (Ex_1(i))$$
$$\texttt{not inInterpretation(}e_1^{excl}\texttt{,}M\texttt{),}\ldots\texttt{,not inInterpretation(}e_m^{excl}\texttt{,}M\texttt{),isAS(}M\texttt{).}$$

If $e_i \in E^+$ is a positive example, we include

$$\texttt{positive(}i\texttt{).} \quad (Ex_2(i))$$

otherwise if $e_i \in E^-$ is a negative example, we include

$$\texttt{negative(}i\texttt{).} \quad (Ex_2(i))$$

To ensure brave entailment of positive examples and cautious entailment of negative examples, we include the following rules:

$$\texttt{bad :- not cov(}X\texttt{),positive(}X\texttt{).} \quad (Ent_1)$$
$$\texttt{bad :- cov(}X\texttt{),negative(}X\texttt{).} \quad (Ent_2)$$
$$\texttt{:- bad.} \quad (Ent_3)$$

Final encoding

Given the task T the program $P_{exp}(T)$ is the following $\mathsf{ASP}^{\mathsf{N}}$ program:

$$\begin{aligned} P_{exp}(T) \coloneqq\ & \{Bg(i) \mid R_i \in B\} \cup \\ & \{Hypo(i) \mid h_i \in S_M\} \cup \{Hypo\} \cup \\ & \{Int_1(j),\ Int_2(j),\ Int_3(j) \mid M_j \subseteq HB_{B \cup S_M}\} \cup \\ & \{As_1,\ As_2,\ As_3,\ As_4,\ As_5\} \cup \\ & \{Ex_1(i),\ Ex_2(i) \mid e_i \in E^+ \cup E^-\} \cup \\ & \{Ent_1,\ Ent_2,\ Ent_3\} \end{aligned}$$

To illustrate the encoding, we consider the example task T_e presented in section 2.2.1 (page 14).

Running example: the program $P_{exp}(T_e)$

We show the full program P_{exp} for the task T_e.

```
%% Background knowledge B = {p :- not q}
inAs(p,M) :- reduct(ninAs(q,M)),interpretation(M).

%% Hypothesis SM = { h1:  q., h2:  q :- not p.  }
0{h(1); h(2)}2.
inAs(q,M) :- interpretation(M),h(1).
inAs(q,M) :- reduct(ninAs(p,M)),interpretation(M),h(2).

%% Enumerating all possible interpretations
interpretation(1).ninInterpretation(q,1).ninInterpretation(p,1). %% K1 = ∅
interpretation(2).ninInterpretation(q,2).inInterpretation(p,2).  %% K2 = {p}
interpretation(3).inInterpretation(q,3).ninInterpretation(p,3).  %% K3 = {q}
interpretation(4).inInterpretation(q,4).inInterpretation(p,4).   %% K4 = {p,q}

%% Reduction rules
 :- reduct(ninAs(X,M)),inInterpretation(X,M),interpretation(M).
reduct(ninAs(X,M)) :- ninInterpretation(X,M),interpretation(M).
notAS(M) :- inAs(X,M),ninInterpretation(X,M),interpretation(M).
notAS(M) :- not inAs(X,M),inInterpretation(X,M),interpretation(M).
isAS(M) :- not notAS(M),interpretation(M).

%% Positive example ({p.}, {q.})
positive(1).
cov(1) :- inInterpretation(p,M),not inInterpretation(q,M),isAS(M).

%% Negative example ({q.}, {p.})
negative(2).
cov(2) :- inInterpretation(q,M),not inInterpretation(p,M),isAS(M).

%% Entailment conditions
bad :- not cov(X),positive(X).
bad :- cov(X),negative(X).
 :- bad.
```

The program has exactly one answer set A_1:

```
interpretation(1) interpretation(2) interpretation(3) interpretation(4) reduct(ninAs(p,1))
reduct(ninAs(p,3)) reduct(ninAs(q,1)) reduct(ninAs(q,2)) inAs(p,1) inAs(p,2) atom(p)
```

```
atom(q) inInterpretaion(p,2) inInterpretaion(q,3) inInterpretaion(p,4) inInterpretaion(q,4)
ninInterpretaion(p,1) ninInterpretaion(p,3) ninInterpretaion(q,1) ninInterpretaion(q,2) notAS(1)
notAS(4) positive(1) negative(2) notAS(3) isAS(2) cov(1).
```

Since $|AS(P_{exp}(T_e))| = 1$, the task T_e has exactly one inductive solution, denoted by H.
The set $\{h_i \mid \texttt{h(i)} \in A_1\}$ is empty, which implies that the only inductive solution is $H = \emptyset$.
The only atom in A_1 of the form `isAS( · )` is `isAS(2)`, meaning that the sole answer set A of $B \cup H$ contains only the atom `p` (as indicated by the atoms `inInterpretaion(p,2)` and `ninInterpretaion(q,2)`).
This answer set A extends the positive example (indicated by the presence of `cov(1)` in A_1) and does not extend the negative example (indicated by the absence of `cov(2)` in A_1).
This confirms that H is indeed an inductive solution of T_e.

3.1.2 Correctness, completeness and complexity

We now analyze the correctness, completeness, and complexity of the encoding $P_{exp}(T)$. Before stating the main theorem on correctness and completeness, we first characterize the structure of the answer sets of $P_{exp}(T)$ when the entailment rules are omitted.

Lemma 3.1 *Let T be a ground ILP_{LAS} task. Let $P'_{exp}(T)$ be the program $P_{exp}(T)$ without the entailment rules (i.e. without the set of rules $\{Ent_1, Ent_2, Ent_3\}$). It holds that:*

1. *the number of answer sets of $P'_{exp}(T)$ is $2^{|S_M|}$;*
2. *for every answer set A of $P'_{exp}(T)$, let $H = \{h_i \mid \texttt{h(i)} \in A\}$. The set $AS(B \cup H)$ corresponds to the set $\{\{\texttt{p} \mid \texttt{inInterpretation}(p, i) \in A\} \mid \texttt{isAS}(i) \in A\}$.*

Proof. We begin with point 1. Consider the program $P''_{exp}(T)$, obtained from $P'_{exp}(T)$ by removing rules *Hypo*, As_1, As_4 and As_5. The resulting program is definite (i.e., it belongs to the class $\mathsf{ASP}^{\mathsf{def}}$ and therefore admits a unique answer set. Now, adding back rule *Hypo*, which is a choice rule over the hypothesis space S_M, results in $2^{|S_M|}$ answer sets, one for each possible subset $H \subseteq S_M$. Finally, reintroducing rules As_1, As_4, and As_5 does not change the number of answer sets. These rules neither introduce additional non-determinism nor eliminate any of the existing answer sets—since no constraints or denials are added that could invalidate previously valid models.

We now turn to point 2. Consider an answer set A of $P'_{exp}(T)$ and let $H = \{h_i \mid \texttt{h(i)} \in A\}$. We first show that the following inclusion holds:

$$AS(B \cup H) \subseteq \{\{\texttt{p} \mid \texttt{inInterpretation}(p, i) \in A\} \mid \texttt{isAS}(i) \in A\}\,. \tag{3.1}$$

Let C be an answer set of $B \cup H$. By construction, the program $P'_{exp}(T)$ enumerates all possible interpretations of $B \cup S_M$, and in particular, there must exist some index j such that the answer set A contains the encoding of C, that is:

$$(\{\texttt{inInterpretation}(p, j) \mid p \in C\} \cup \{\texttt{ninInterpretation}(q, j) \mid q \notin C\} \cup \texttt{interpretation}(j)) \subset A \tag{3.2}$$

To prove equation 3.1, we have to prove that $\texttt{isAS}(j) \in A$. This is guaranteed if the following equality holds:

$$\{\texttt{inInterpretation}(p, j) \mid \texttt{inInterpretation}(p, j) \in A\} = \{\texttt{inAs}(p, j) \mid \texttt{inAs}(p, j) \in A\} \quad (3.3)$$

In this case, rules As_3, As_4 and As_5 ensure that the atom $\texttt{isAS}(j)$ is supported and hence $\texttt{isAS}(j) \in A$.

Let R be a rule in the program $B \cup H$, and let $enc(R)$ denote its encoding in $P'_{exp}(T)$. By construction, we have $head(enc(R)) = \texttt{inAs}(head(R), A)$.

If $B \cup H$ is an ASP$^{\mathsf{def}}$ program (i.e., it contains no negated literals), then it is straightforward to verify equation (3.3). Each atom p in the body of R corresponds to an atom $\texttt{inAs}(p, j)$ in the grounding of $enc(R)$. Moreover, rules in the set $\{enc(R) \mid R \in S_M \setminus H\}$ are never activated since the corresponding guard $\texttt{h(i)}$ does not belong to A.

If $B \cup H$ is an ASP$^{\mathsf{N}}$ program and R contains a negated literal $\texttt{not}\ p$, then $ground(enc(R))$ contains the positive literal $\texttt{reduct}(\texttt{ninAs}(p, j))$. We distinguish two cases:

(i) If $p \in C$, then $head(R)$ cannot be supported by R itself by definition of the reduct $(B \cup H)^C$. Rule As_1 ensures that $\texttt{reduct}(\texttt{ninAs}(p, j)) \notin A$, hence $head(enc(R))$ cannot be supported by $enc(R)$ itself.

(ii) If $p \notin C$, then R appears in the reduct without the negated literal $\texttt{not}\ p$ in its body. Rule As_2 ensures that $\texttt{reduct}(\texttt{ninAs}(p, j)) \in A$ and so the rule $enc(R)$ is not discarded.

In other words, the reduct behavior of negated literals is correctly simulated by $P'_{exp}(T)$, which implies that both equations (3.3) and (3.1) hold.

It remains to prove the opposite inclusion of equation (3.1), that is:

$$AS(B \cup H) \supseteq \{\{\texttt{p} \mid \texttt{inInterpretation}(p, i) \in A\} \mid \texttt{isAS}(i) \in A\}\,. \quad (3.4)$$

Let j be an index such that $\texttt{isAS}(j) \in A$ and let $C = \{\texttt{p} \mid \texttt{inInterpretation}(p, j) \in A\}$. We must show that C is an answer set of $B \cup H$, i.e., that $M_{(B \cup H)^C} = C$. This follows directly from the fact that $P'_{exp}(T)$ correctly simulates reduct evaluation. This concludes the proof of the lemma.

□

We now turn our attention to the full program $P_{exp}(T)$, as opposed to the simplified version $P'_{exp}(T)$. Given the characterization provided by lemma 3.1, the correctness and completeness theorem for $P_{exp}(T)$ becomes almost immediate: since we have shown that $P_{exp}(T)$ correctly enumerates all answer sets of $B \cup H$, it remains only to verify that the entailment rules faithfully reflect the LAS semantics.

Theorem 3.2 (Correctness and completeness) *Let T be a ground ILP_{LAS} task.*
It holds that $ILP_{LAS}(T) = \{\{h_i \mid \texttt{h(i)} \in A\} \mid A \in AS(P_{exp}(T))\}$.

Proof. Let A be an answer set of $P_{exp}(T)$, and define $H = \{h_i \mid \texttt{h(i)} \in A\}$. By lemma 3.1 it holds that

$$AS(B \cup H) = \{\{\texttt{p} \mid \texttt{inInterpretation}(p, i) \in A\} \mid \texttt{isAS}(i) \in A\} \quad (3.5)$$

This characterization holds because rules Ent_1, Ent_2, and Ent_3 — which are present in $P_{exp}(T)$ but not in $P'_{exp}(T)$ — do not affect the structure of answer sets. Instead, they act as global constraints that may eliminate answer sets of $P'_{exp}(T)$ which do not satisfy the inductive conditions, i.e., brave entailment of all positive examples and cautious rejection of all negative examples. Moreover, from rule $Ex_1(i)$, it follows that $B \cup H$ extends an example e_i if and only if it holds $\texttt{cov(}i\texttt{)} \in A$.

To prove the correctness of the encoding, consider an answer set A of $P_{exp}(T)$. Since rule Ent_3 enforces that $\texttt{bad} \notin H$, it must be the case that $\texttt{cov(}i\texttt{)} \in A$ for every positive example e_i, and $\texttt{cov(}i\texttt{)} \notin A$ for every negative example. This implies that every positive example is extended by some answer set of $B \cup H$, and no negative example is extended — hence, H is an inductive solution of T.

To prove completeness, let $H \subseteq S_M$ be an inductive solution of T. By lemma 3.1, the program $P'_{exp}(T)$ admits one answer set A corresponding to H, i.e., such that $\texttt{h(i)} \in A$ if and only if $h_i \in H$. Since H satisfies the inductive constraints by assumption, all relevant $\texttt{cov(}i\texttt{)}$ atoms are correctly supported, and the constraint in rule Ent_3 is not violated. Therefore, A is a stable model of $P_{exp}(T)$, completing the proof. □

To conclude this section, we analyze the size of the encoding $P_{exp}(T)$. Although the result is almost immediate, it formally confirms that the program has exponential size. Since $P_{exp}(T)$ is non-ground, we also estimate the size of its grounding.

Lemma 3.3 *Let $T = \langle B, S_M, E^+, E^- \rangle$ be a ground ILP_{LAS} task. Then the following holds:*

1. $|P_{exp}(T)| \in \Theta\left(|T| + 2^{|HB_{B \cup S_M}|}\right)$
2. $|ground(P_{exp}(T))| \in \Theta\left(|E^+| + |E^-| + |B \cup S_M| \times 2^{|HB_{B \cup S_M}|}\right)$

Proof. The size of the program $P_{exp}(T)$ consists of: (i) a linear number of rules encoding B, S_M, and the examples, plus (ii) a set of rules that explicitly enumerate all interpretations of the Herbrand base $HB_{B \cup S_M}$. Since each interpretation M is encoded using a linear number of facts, the total number of such facts is $\Theta\left(2^{|HB_{B \cup S_M}|}\right)$. This establishes the bound for $|P_{exp}(T)|$.

We now turn to the grounding. The number of rules generated by grounding rules Ent_1 and Ent_2 is linear in the size of $E^+ \cup E^-$. The number of grounded rules resulting from As_1 through As_5 is linear in the number of interpretations. Finally, grounding rules $Bg(i)$ and $Hypo(i)$ yields a number of rules bounded by $(|B| + |S_M|) \cdot 2^{|HB_{B \cup S_M}|}$, as each rule is instantiated once for every interpretation.

This proves the claimed bound for $|ground(P_{exp}(T))|$.

□

3.2 Polynomial $\mathsf{ASP}^{\mathsf{D}}$ encoding

In section 3.1, we presented an exponential-size encoding. The main bottleneck in that construction lies in the handling of cautious entailment, which requires enumerating all answer sets of $B \cup H$. This is achieved by explicitly generating all interpretations and checking whether they are stable models. To overcome this limitation, in the present encoding we avoid the explicit enumeration of answer sets. Instead, we characterize them symbolically using a Boolean formula whose models correspond to stable models of $B \cup H$. This distinction mirrors the classical difference between model checking and symbolic

model checking [45] in formal verification: rather than enumerating states one by one, we encode the entire state space symbolically and reason about it more compactly.

3.2.1 Encoding

Let $T = \langle B, S_M, E^+, E^- \rangle$ be a ground ILP_{LAS} task. We define the ASP$^{\mathsf{D}}$ program $P_{dis}(T)$ such that $P_{dis}(T)$ is consistent if and only if T is satisfiable (i.e. admits an inductive solution) and the set $AS(P_{dis}(T))$ corresponds one-to-one with the set of inductive solutions of T. If the program $B \cup S_M$ is tight, the encoding is linear in the size of T; otherwise, the size of the encoding may become exponential due to the inclusion of loop formulas. The program $P_{dis}(T)$ is composed of two main components: (i) $P^+_{dis}(T)$, which ensures that all positive examples are bravely entailed; (ii) $P^-_{dis}(T)$, which enforces that all negative examples are cautiously avoided.

As discussed throughout this thesis, the main technical difficulty lies in handling the cautious entailment constraints, which are responsible for the intrinsic Σ^P_2-hardness of the satisfiability problem.

We now proceed to describe the two components of $P_{dis}(T)$ in detail.

Cautious entailment

Given a ground ILP_{LAS} task T, the cautious entailment component $P^-_{dis}(T)$ is constructed in two main stages:

1. The task T is first translated into a quantified Boolean formula $\phi(T)$ of the form $\exists X \forall Y\, \psi(T)$.
2. The formula $\phi(T)$ is then compiled into a ground ASP$^{\mathsf{D}}$ program $P^-_{dis}(T)$.

The first stage builds upon the notions introduced in section 2.1.4, and in particular on theorem 2.13, which illustrates how to translate an ASP program into a Boolean formula. The second stage relies on concepts from section 2.1.3, and specifically on theorem 2.7, which shows how to compile a quantified Boolean formula into a ground ASP$^{\mathsf{D}}$ program using the saturation technique from Eiter and Gottlob [25].

We now describe each of these stages in detail.

Satge 1. We begin by defining the guarded hypothesis space S'_M as follows. Given a rule $h_i \in S_M$ of the form:

$$H \texttt{ :- } A_1, \ldots, A_n, \texttt{not } B_1, \ldots, \texttt{not } B_m.$$

we define the corresponding rule $h'_i \in S'_M$ as:

$$H \texttt{ :- } A_1, \ldots, A_n, \texttt{not } B_1, \ldots, \texttt{not } B_m, \texttt{h(i)}.$$

where `h(i)` is a fresh propositional atom associated uniquely with h_i. In other words, each rule in S'_M may be activated only if its corresponding guard `h(i)` is.

We define the programs $P(T) \coloneqq B \cup S_M$ and $P'(T) \coloneqq B \cup S'_M$. Our goal is to describe the answer sets of $P'(T)$ by means of a Boolean formula, once the values of atoms `h(i)` have been fixed. A natural candidate would be the formula $\phi_{as}(P'(T)) \coloneqq \phi_{iff}(P'(T)) \wedge \phi_{loop}(P'(T))$, as introduced in section 2.1.4.

However, this is not suitable in our case: since atoms `h(i)` do not appear in the head of any rule, they would be automatically forced to be false in every model of $\phi_{as}(P'(T))$ due to the completion semantics (we will see an example in a moment). Instead, we want these atoms to be left externally unconstrained — as if an oracle could freely assign their values. To this end, we define modified formulas $\phi^*_{\mathit{iff}}(P'(T))$, $\phi^*_{loop}(P'(T))$, and $\phi^*_{as}(P'(T))$, which are adapted variants of the classical notions introduced earlier.

Definition 3.4 (Variants of program completion and loop formula)
Given $P'(T)$, let $S = \{\texttt{h(i)} \mid h_i \in S_M\}$ be the set of guard atoms. We define:

$$\phi^*_{\mathit{iff}}(P'(T)) \coloneqq \bigwedge_{a \in atoms(P'(T)) \setminus S} \left(a \leftrightarrow \bigvee_{R \in P'(T), head(R)=a} \left(\bigwedge_{b \in body^+(R)} b \wedge \bigwedge_{b \in body^-(R)} \neg b \right) \right)$$

$$\phi^*_{loop}(P'(T)) \coloneqq \bigwedge_{L \in loops(P'(T))} \left(\left(\bigvee_{a \in L \setminus S} a \right) \rightarrow \left(\bigvee_{R \in ES(L)} \left(\bigwedge_{b \in body^+(R)} b \wedge \bigwedge_{b \in body^-(R)} \neg b \right) \right) \right)$$

$$\phi^*_{as}(P'(T)) \coloneqq \phi^*_{\mathit{iff}}(P'(T)) \wedge \phi^*_{loop}(P'(T))$$

The following fact — a direct consequence of theorem 2.13 — establishes the correspondence between models of $\phi^*_{as}(P'(T))$ and the answer sets of $P'(T)$ extended with fixed guards.

Fact 3.5 *Let A be an interpretation of $\phi^*_{as}(P'(T))$, and define the program $P^A \coloneqq \bigcup_{h(i) \in A} \{\texttt{h(i).}\}$. Then:*

$$A \models \phi^*_{as}(P'(T)) \leftrightarrow A \in AS(P'(T) \cup P^A).$$

To better understand the difference between these formulas, we illustrate the distinction between $\phi_{as}(P'(T_e))$ and $\phi^*_{as}(P'(T_e))$, where T_e is the example task defined in section 2.2.1 (page 14).

Running example: difference between $\phi_{as}(P'(T_e))$ and $\phi^*_{as}(P'(T_e))$

The program $P'(T_e)$ is the following ASP program:

```
p :- not q.
q :- h(1).
q :- not p, h(2).
```

Since $P'(T_e)$ is tight, we can apply the definitions from section 2.1.4 and definition 3.4, obtaining:

$$\phi^*_{as}(P'(T_e)) \coloneqq \phi^*_{loop}(P'(T_e)) = (p \leftrightarrow \neg q) \wedge (q \leftrightarrow ((\neg p \wedge h(2)) \vee h(1)))$$
$$\phi_{as}(P'(T_e)) \coloneqq \phi_{loop}(P'(T_e)) = \phi^*_{loop}(P'(T_e)) \wedge (h(1) \leftrightarrow \mathit{false}) \wedge (h(2) \leftrightarrow \mathit{false})$$

Observe that since the atoms `h(1)` and `h(2)` do not appear in the head of any rule in $P'(T_e)$, the standard completion formula $\phi_{loop}(P'(T_e))$ forces them to be false. In contrast, the variables $h(1)$ and $h(2)$ are left unconstrained in the relaxed formula $\phi^*_{loop}(P'(T_e))$. As an application of fact 3.5, consider:

- For $A_1 = \{h(2), p\}$, since $A_1 \models \phi^*_{as}(P'(T_e))$, it follows that $A_1 \in AS(P'(T_e) \cup \{\texttt{h(2).}\})$.

- For $A_2 = \{h(2), q\}$, since $A_2 \in AS(P'(T_e) \cup \{\texttt{h(2).}\})$, it follows that $A_2 \models \phi^*_{as}(P'(T_e))$.

Let $V_{P'(T)}$ denote the set of Boolean variables appearing in $\phi^*_{as}(P'(T))$, i.e., one for each ground atom in the Herbrand base of $P'(T)$. Since the only difference between $P(T)$ and $P'(T)$ lies in the presence of the auxiliary atoms `h(i)`, it holds that:

$$HB_{P'(T)} = HB_{P(T)} \cup \{\texttt{h(i)} \mid h_i \in S_M\}\,.$$

Accordingly, we partition the variable set as $V_{P'(T)} = V_{P(T)} \cup V^H_{P'(T)}$, where: (i) $V_{P(T)}$ is the set of free variables of $\phi_{as}(P(T))$, and (ii) $V^H_{P'(T)} = \{h(i) \mid h_i \in S_M\}$ represents the set of guard atoms for the hypotheses.

We now introduce a propositional formula that characterizes the entailment of the negative example set. Recall that each negative example $e_j \in E^-$ is a partial interpretation of the form:

$$e_j \coloneqq \langle \left\{e^{incl}_{j,1}, \ldots, e^{incl}_{j,m^1_j}\right\}, \left\{e^{excl}_{j,1}, \ldots, e^{excl}_{j,m^2_j}\right\} \rangle.$$

We define the formula $\phi_{neg}(E^-)$ as:

$$\phi_{neg}(E^-) \coloneqq \bigwedge_{e_j \in E^-} \left(\neg e^{incl}_{j,1} \vee \cdots \vee \neg e^{incl}_{j,m^1_j} \vee e^{excl}_{j,1} \vee \cdots \vee e^{excl}_{j,m^2_j}\right).$$

This formula ensures that, for each negative example, at least one inclusion atom is absent or one exclusion atom is present—i.e., an interpretation satisfying $\phi_{neg}(E^-)$ fails to extend every example in E^-. This is formalized by the following fact.

Fact 3.6 *Let A be an interpretation of $B \cup S_M$. Then A satisfies $\phi_{neg}(E^-)$ if and only if for every $e^- \in E^-$, A does not extend e^-.*

We now consider the formula $\psi(T)$ defined as:

$$\psi(T) \coloneqq \mathsf{NNF}(\phi^*_{as}(P'(T)) \rightarrow \phi_{neg}(E^-))$$

where $\mathsf{NNF}(F)$ denotes the *Negation Normal Form* (NNF) of the formula F, that is, a logically equivalent formula in which negation is applied only to atomic propositions, and the only connectives allowed are $\wedge$, $\vee$, and $\neg$. For a formal definition, see e.g., [46].

Notice that the set of free variables of $\psi(T)$ is $V_{P'(T)}$.

Let $\{y_1, \ldots, y_v\}$ be the set of variables $V_{P(T)}$ and let s denote the number of rules in S_M. We define the formula $\phi(T)$ as follows:

$$\begin{aligned}\phi(T) \coloneqq &\exists h(1) \ldots \exists h(s) \forall y_1 \ldots \forall y_v\ \psi(T).\\ = &\exists V^H_{P'(T)}\ \forall V_{P(T)}\ \psi(T)\end{aligned}$$

The idea behind $\phi(T)$ is the following: once a hypothesis $H \subseteq S_M$ is fixed (i.e., we assign truth values to the set of existential variables $V^H_{P'(T)} = \{h(1), \ldots, h(s)\}$), we then consider all possible truth

assignments to the universal variables $V_{P(T)} = \{y_1, \ldots, y_v\}$. Whenever the complete assignment to the variables $V_{P'(T)} = V^H_{P'(T)} \cup V_{P(T)}$ satisfies the formula $\phi^*_{as}(P'(T))$, it corresponds to an answer set $A \in AS(B \cup H)$. In this case, we require that A does not extend any negative example, that is, the formula $\phi_{neg}(E^-)$ must also be satisfied. Lemma 3.7 in section 3.2.2 will formalize this intuition.

Satge 2. From the quantified Boolean formula $\phi(T)$, we construct the program $P^-_{dis}(T)$ by applying a translation similar to the one used by Eiter and Gottlob in the proof of theorem 2.7. The resulting $\mathsf{ASP}^{\mathsf{D}}$ program encodes the formula $\phi(T) = \exists V^H_{P'(T)}\ \forall V_{P(T)}\ \psi(T)$ via saturation.

In particular the program $P^-_{dis}(T)$ is defined as:

`h(i) | nh(i).`	$\forall h(i) \in V^H_{P'(T)}$
`y | ny.  y :- w.  ny :- w.`	$\forall y \in V_{P(T)}$
`w :-` $\texttt{formula}_{\psi(T)}$`.`	
expansion$(\psi(T))$	
`w :- not w.`	

The auxiliary predicate $\texttt{formula}_{\psi(T)}$ is used to evaluate the formula $\psi(T)$ in a bottom-up fashion over its syntax tree. The rule `w :- not w` enforces that `w` must be true in every stable model; however, due to the stable model semantics, `w` must also be supported by $\texttt{formula}_{\psi(T)}$. The subroutine *expansion*(F) encodes the semantics of the Boolean formula F (assumed to be in NNF) as ASP rules, using auxiliary atoms of the form $\texttt{formula}_F$.

The expansion procedure is defined as follows.

- if F is a variable $p \in V_{P'(T)}$, then *expansion*(F) is

  $\texttt{formula}_F$ `:- p.`

- if F is $\neg p$ for a variable $p \in V_{P'(T)}$, then *expansion*(F) is

  $\texttt{formula}_F$ `:- np.`

- if F is $F_1 \wedge F_2$, then *expansion*(F) is

  $\texttt{formula}_F$ `:-` $\texttt{formula}_{F_1}$`,`$\texttt{formula}_{F_2}$`.`
  expansion(F_1)
  expansion(F_2)

- if F is $F_1 \vee F_2$, then $\mathit{expansion}(F)$ is

$$\texttt{formula}_F \texttt{ :- formula}_{F_1}\texttt{.}$$
$$\texttt{formula}_F \texttt{ :- formula}_{F_2}\texttt{.}$$
$$\mathit{expansion}(F_1)$$
$$\mathit{expansion}(F_2)$$

To illustrate the program P^{-}_{dis}, we reconsider the example task T_e introduced in section 2.2.1 (page 14).

Running example: the program $P^{-}_{\boldsymbol{dis}}(T_e)$

We now present the program $P^{-}_{dis}(T_e)$ for the task T_e, explicitly illustrating the intermediate steps leading to its construction. The program $B \cup S_M$ is tight, so the formula $\phi^{*}_{as}(P'(T_e))$ coincides with the formula completion formula $\phi^{*}_{\mathit{iff}}(P'(T_e))$:

$$\phi^{*}_{as}(P'(T_e)) \coloneqq (p \leftrightarrow \neg q) \wedge (q \leftrightarrow ((\neg p \wedge h(2)) \vee h(1))).$$

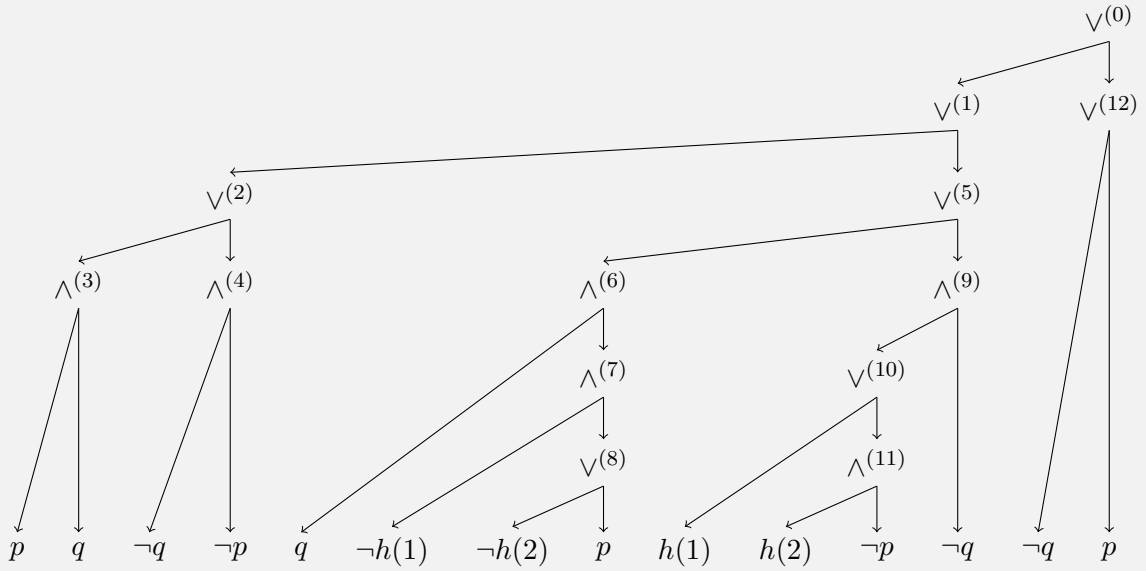
The set of free variables of $\phi(T_e)$ is $V_{P'(T_e)} = V_{P(T_e)} \cup V^{H}_{P'(T_e)} = \{p, q\} \cup \{h(1), h(2)\}$
In the task T_e, the only negative example is $\langle \{\texttt{q.}\}, \{\texttt{p.}\} \rangle$, hence:

$$\phi_{neg}(E^{-}) \coloneqq (\neg q \vee p).$$

The formula $\psi(T_e)$, obtained by transforming into NNF the formula $\phi^{*}_{as}(P'(T_e)) \rightarrow \phi_{neg}(E^{-})$, is:

$$((((p \wedge q) \vee (\neg q \wedge \neg p)) \vee ((q \wedge (\neg h(1) \wedge (\neg h(2) \vee p))) \vee ((h(1) \vee (h(2) \wedge \neg p)) \wedge \neg q))) \vee (\neg q \vee p))$$

To better visualize the structure of $\psi(T_e)$ we show the syntax tree. Each internal node is annotated with a unique index:



The final formula from Stage 1 is:

$$\phi(T_e) \coloneqq \exists h(1)\ \exists h(2)\ \forall q\ \forall p\ \ \psi(T_e).$$

We now proceed with the encoding of $\phi(T_e)$ into the disjunctive ASP program $P^{-}_{dis}(T_e)$:

```
%% Existential variables
h(1) | nh(1). h(2) | nh(2).
%% Universal variables
p | np. q | nq.
p  :- w. np  :- w. q  :- w. nq  :- w.
%% Formuala encoding
w :- formula_0.
formula_0 :- formula_1.
formula_0 :- formula_12.
formula_1 :- formula_2.
formula_1 :- formula_5.
formula_2 :- formula_3.
formula_2 :- formula_4.
formula_3 :- p, q.
formula_4 :- nq, np.
formula_5 :- formula_6.
formula_5 :- formula_9.
formula_6 :- q, formula_7.
formula_7 :- nh(1), formula_8.
formula_8 :- nh(2).
formula_8 :- p.
formula_9 :- formula_10, nq.
formula_10 :- h(1).
formula_10 :- formula_11.
formula_11 :- h(2), np.
formula_12 :- nq.
formula_12 :- p.
%% Saturation predicate
w  :- not w.
```

To briefly explain the program *expansion*($\psi(T_e)$), consider that the atom $\texttt{formula}_0$ represents the root of the syntax tree of $\psi(T_e)$. Since $\psi(T_e)$ is a disjunction, the program includes the rules

$$\texttt{formula}_0 \texttt{ :- formula}_1\texttt{.} \quad \texttt{formula}_0 \texttt{ :- formula}_{12}\texttt{.}$$

The atom $\texttt{formula}_{12}$ represents the disjunction $\neg q \vee p$, so it is encoded as:

$$\texttt{formula}_{12} \texttt{ :- nq.} \quad \texttt{formula}_{12} \texttt{ :- p.}$$

The only answer set of the program $P^{-}_{dis}(T_e)$ is:
$\texttt{p np q nq nh(1) nh(2) w formula}_0 \texttt{ formula}_1 \texttt{ formula}_2 \texttt{ formula}_3 \texttt{ formula}_4 \texttt{ formula}_5$
$\texttt{formula}_6 \texttt{ formula}_7 \texttt{ formula}_8 \texttt{ formula}_{12}$
This correctly indicates that every answer set of $B \cup \emptyset$ do not extend the negative example, satisfying the cautious entailment requirement.

Brave entailment

Given a ground ILP_{LAS} task T, the brave entailment component $P^{+}_{dis}(T)$ uses elements already introduced and is similar to the encoding used in the ILASP1 algorithm. In particular, for each rule $R_i \in B$ of the background knowledge of the form:

$$H \texttt{ :- } A_1, \ldots, A_n, \texttt{not } B_1, \ldots, \texttt{not } B_m.$$

the program $P^{+}_{dis}(T)$ contains the rule:

$$\texttt{inAs}(H, M) \texttt{ :- inAs}(A_1, M), \ldots, \texttt{inAs}(A_n, M), \quad (Bg(i))$$
$$\texttt{not inAs}(B_1, M), \ldots, \texttt{not inAs}(B_m, M), \texttt{interpretation}(M).$$

Similarly, for each hypothesis $h_i \in S_M$ in the hypothesis space of the form:

$$H \texttt{ :- } A_1, \ldots, A_n, \texttt{not } B_1, \ldots \texttt{not } B_m.$$

the program $P^{+}_{dis}(T)$ contains the rule:

$$\texttt{inAs}(H, M) \texttt{ :- inAs}(A_1, M), \ldots, \texttt{inAs}(A_n, M), \quad (Hypo(i))$$
$$\texttt{not inAs}(B_1, M), \ldots, \texttt{not inAs}(B_m, M), \texttt{interpretation}(M), \texttt{h(i)}.$$

Notice that for rules $R \in B \cup H$, the encoding is the same as the one used in the program $P_{exp}(T)$ (introduced in section 3.1) except for the treatment of negated literals: here we don't use predicates $\texttt{reduct(ninAs}(\cdot,\cdot)\texttt{)}$, instead we map negated literals into negated literals enclosed in the predicate $\texttt{inAs}(\cdot,\cdot)$.

Finally, for each positive example $e_i \in E^+$ of the form $\langle \{e^{incl}_1, \ldots, e^{incl}_n\}, \{e^{excl}_1, \ldots, e^{excl}_m\} \rangle$, the

program $P^{+}_{dis}(T)$ includes the following rules:

$$\texttt{interpretation(}i\texttt{).} \tag{$Ex_1(i)$}$$

$$\texttt{cov(}i\texttt{) :- inAs(}e^{incl}_1,i\texttt{),}\ldots\texttt{,inAs(}e^{incl}_n,i\texttt{),} \tag{$Ex_2(i)$}$$
$$\texttt{not inAs(}e^{excl}_1,i\texttt{),}\ldots\texttt{,not inAs(}e^{excl}_m,i\texttt{).}$$

$$\texttt{:- not cov(}i\texttt{).} \tag{$Ex_3(i)$}$$

Given the task T the program $P^{+}_{dis}(T)$ is the following $\mathsf{ASP}^{\mathsf{D}}$ program:

$$\begin{aligned} P^{+}_{dis}(T) \coloneqq &\{Bg(i) \mid R_i \in B\} \cup \\ &\{Hypo(i) \mid h_i \in S_M\} \cup \\ &\{Ex_1(i),\ Ex_2(i),\ Ex_2(i) \mid e_i \in E^{+}\} \end{aligned}$$

Notice that the program $P^{+}_{dis}(T)$ is not self-contained: atoms of the form `h(i)` do not appear in the head of any rule. Therefore, in order to use the $P^{+}_{dis}(T)$ encoding as a standalone module, it is necessary to add rules that explicitly encode the fact that each `h(i)` may or may not belong to an answer set. We define the *completed version* of $P^{+}_{dis}(T)$ (denoted $P^{+}_{dis}{}'(T)$) as the program:

$$P^{+}_{dis}{}'(T) \coloneqq P^{+}_{dis}(T) \cup \{\texttt{h(i) | nh(i).} \mid h_i \in S_M\}$$

As usual, to illustrate the effect of the brave entailment part, we consider the program $P^{+}_{dis}(T_e)$ where the task T_e is the one introduced in section 2.2.1 (page 14).

Running example: the program $P^{+}_{dis}(T_e)$

After presenting the program $P^{-}_{dis}(T_e)$ in the last paragraph, we now see the program $P^{+}_{dis}(T_e)$.

```
%% Background knowledge B = {p :- not q}
inAs(p,M) :- not inAs(q,M),interpretation(M).

%% Hypothesis SM = { h1:  q., h2:  q :- not p.  }
inAs(q,M) :- interpretation(M),h(1).
inAs(q,M) :- not inAs(p,M),interpretation(M),h(2).

%% Positive example ({p.}, {q.})
interpretation(1).
cov(1) :- inAs(p,1),not inAs(q,1).
 :- not cov(1).
```

The program $P^{+}_{dis}(T_e)$ is not self-contained since atoms `h(1)` and `h(2)` do not occur in any rule head. We thus consider the program $P^{+}_{dis}{}'(T_e) \coloneqq P^{+}_{dis}(T_e) \cup \{\texttt{h(1) | nh(1). h(2) | nh(2).}\}$. The program $P^{+}_{dis}{}'(T_e)$ has two answer sets:

A_1: `interpretation(1) nh(1) nh(2) inAs(p,1) cov(1)`

A_2: `interpretation(1) nh(1) h(2) inAs(p,1) cov(1)`

This correctly indicates that:

- there exists an answer set of $B \cup \emptyset$ which extends the positive example ;
- there exists an answer set of $B \cup \{h_2\}$ which extends the positive example.

In fact, as seen in the presentation of the task T_e (in section 2.2.1, at page 14), it holds that $AS(B \cup \emptyset) = \{\{p\}\}$ and $AS(B \cup \{h_2\}) = \{\{p\}, \{q\}\}$.

Final encoding

The final encoding $P_{dis}(T)$ is defined as the union of the two modules:

$$P_{dis}(T) \coloneqq P^{-}_{dis}(T) \cup P^{+}_{dis}(T).$$

Similarly to the $P_{exp}(T)$ encoding, this disjunctive encoding also follows a guess-and-verify approach. The module $P^{-}_{dis}(T)$ guesses a hypothesis $H \subseteq S_M$ that satisfies the constraints imposed by the negative examples. Then, the module $P^{+}_{dis}(T)$ uses the guessed hypothesis and checks whether it also satisfies all the positive examples. In practice, the set of inductive solutions for T is given by:

$$\{\{h_i \mid \texttt{h(i)} \in A\} \mid A \in AS(P_{dis}(T))\}.$$

Alternatively, this set can be obtained by intersecting the inductive solutions computed by $P^{-}_{dis}(T)$ with those computed by $P^{+}_{dis}{}'(T)$, i.e., the *completed version* of $P^{+}_{dis}(T)$.

To illustrate this construction, we consider the task T_e one more time.

Running example: the program $P_{dis}(T_e)$

The only answer set A_1 of the full program $P_{dis}(T_e) = P^{-}_{dis}(T_e) \cup P^{+}_{dis}(T_e)$ is:
`interpretation(1) nh(1) inAs(p,1) cov(1) p np q nq nh(2) w formula`$_0$ `formula`$_1$ `formula`$_2$ `formula`$_3$ `formula`$_4$ `formula`$_5$ `formula`$_6$ `formula`$_7$ `formula`$_8$ `formula`$_{12}$
This indicates that the only inductive solution of T_e is $H = \{h_i \mid \texttt{h(i)} \in A_1\} = \emptyset$. This is the intersection between the set of solutions found by $P^{-}_{dis}(T_e)$ and $P^{+}_{dis}{}'(T_e)$ In particular, the set of inductive solutions is the set $\left\{\{h_i \mid \texttt{h(i)} \in A\} \mid A \in AS(P^{-}_{dis}(T_e))\right\} \cap \left\{\{h_i \mid \texttt{h(i)} \in A\} \mid A \in AS(P^{+}_{dis}{}'(T_e))\right\}$ which evaluates to $\{\emptyset\} \cap \{\emptyset, h_2\} = \{\emptyset\}$.

3.2.2 Correctness, completeness and complexity

We now discuss correctness, completeness and complexity for the disjunctive encoding $P_{dis}(T)$.

We begin by analysing properties of the cautious entailment module $P^{-}_{dis}(T)$. In particular we show that given a model of the formula $\forall V_{P(T)}\ \psi(T)$ we can compute an hypothesis satisfying cautious

requirements. The vice-versa also holds: given such an hypothesis we are able to find a model of $\forall V_{P(T)}\ \psi(T)$.

Lemma 3.7 *Let T be a ground ILP_{LAS} task, let $M \subseteq V^H_{P'(T)}$ be an interpretation of $\forall V_{P(T)}\ \psi(T)$, and let $H \coloneqq \{h_i \mid h(i) \in M\}$. Then the following are equivalent:*

1. *$M \models \forall V_{P(T)}\ \psi(T)$;*
2. *For every $A \in AS(B \cup H)$ and every $e^- \in E^-$, it holds that A does not extend e^-.*

Proof. We have to prove two implications.

$\boxed{1 \to 2}$

Since M is a model of $\forall V_{P(T)}\ \psi(T)$, for every assignment $\sigma : V_{P(T)} \to \{True, False\}$ to the universal variables, following the definition of $\psi(T)$, it must hold $M, \sigma \models \phi^*_{as}(P'(T)) \to \phi_{neg}(E^-)$.

For a specific assignment σ, let us consider A'_σ as the interpretation of $\psi(T)$ defined as $A'_\sigma \coloneqq M \cup A_\sigma$, where A_σ is defined as:

$$A_\sigma \coloneqq \{\mathtt{y} \mid y \in V_{P(T)}, \sigma(y) = True\} \tag{3.6}$$

By fact 3.5, it holds $A'_\sigma \models \phi^*_{as}(P'(T))$ if and only if A'_σ is an answer set of $P'(T) \cup P^{A'_\sigma} = B \cup S'_M \cup \bigcup_{h(i) \in A'_\sigma} \{\mathtt{h(i).}\}$. By definition of the guarded hypothesis space S'_M we have that A'_σ is also an answer set of $B \cup H' \cup P^{A'_\sigma}$ where $H' = \{h'_i \mid \sigma(h(i)) = True\}$. This means that A_σ is an answer set of $B \cup H$ if and only if $A'_\sigma \models \phi^*_{as}(P'(T))$. For a specific σ, suppose that it holds $M, \sigma \models \phi^*_{as}(P'(T))$, then, it must also hold $M, \sigma \models \phi_{neg}(E^-)$. This implies that $A_\sigma \models \phi_{neg}(E^-)$. By fact 3.6, we conclude that for every $e^- \in E^-$, A_σ does not extend e^-. Since σ was arbitrary, this holds for every answer set $A_\sigma \in AS(B \cup H)$, completing this part of the proof.

$\boxed{2 \to 1}$

Let A be an interpretation of $B \cup H$ and let us define $A' \coloneqq M \cup A$. We now divide our discussion into two cases:

- If $A \notin AS(B \cup H)$, by theorem 2.13, $A \not\models \phi^*_{as}(P(T))$ which implies $A' \not\models \phi^*_{as}(P'(T))$.
- If $A \in AS(B \cup H)$, by theorem 2.13, $A \models \phi^*_{as}(P(T))$ and so $A' \models \phi^*_{as}(P'(T))$. Furthermore, by assumption, for every $e^- \in E^-$, it holds that A does not extend e^-. By fact 3.6 $A \models \phi_{neg}(E^-)$ and thus $A' \models \phi_{neg}(E^-)$.

In either case, we obtain $A' \models \psi(T)$. Since $A \subseteq HB_{B \cup S_M}$ was chosen arbitrary, it follows that $M \models \forall V_{P(T)}\ \psi(T)$. □

Notice that lemma 3.7 can be equivalently be rewritten in terms of the validity of the formula $\phi(T) = \exists V^H_{P'(T)}\ \forall V_{P(T)}\ \psi(T)$ as stated by the following fact.

Fact 3.8 *The formula $\phi(T)$ is valid if and only if there exists a model satisfying the formula $\forall V_{P(T)}\ \psi(T)$.*

Our goal is now to reformulate lemma 3.7 in terms of the ASP^D program $P^-_{dis}(T)$. To this end, we define two notions of *consistency* of an interpretation of $P^-_{dis}(T)$.

Definition 3.9 *Let T be a ground ILP_{LAS} task and consider the program $P^{-}_{dis}(T)$. Consider the following conditions:*

1. *for every Boolean formula F such that F is in* NNF *and expansion(F) is included in $P^{-}_{dis}(T)$:*
   - *if F is the conjunction $F = F_1 \wedge F_2$, then, $\texttt{formula}_F \in A$ if and only if $\texttt{formula}_{F_1} \in A$ and $\texttt{formula}_{F_2} \in A$;*
   - *if F is the disjunction $F = F_1 \vee F_2$, then, $\texttt{formula}_F \in A$ if and only if $\texttt{formula}_{F_1} \in A$ or $\texttt{formula}_{F_2} \in A$;*
   - *if F is an atomic formula p, then, $\texttt{formula}_F \in A$ if and only if $\texttt{p} \in A$;*
   - *if F is a negated literal $\neg p$, then, $\texttt{formula}_F \in A$ if and only if $\texttt{np} \in A$;*
2. *for every $y \in V_{P(T)}$ it holds that $\texttt{y} \in A'$ and $\texttt{ny} \in A'$;*
3. *for every $h(i) \in V^{H}_{P'(T)}$ it holds that exactly one of $\texttt{h(i)}$ and $\texttt{nh(i)}$ is included in A' ;*
4. *$\texttt{w} \in A$;*
5. *$\texttt{formula}_{\psi(T)} \in A$;*
6. *for every $y \in V_{P'(T)}$ it holds that exactly one of $\texttt{y}$ and $\texttt{ny}$ is included in A' .*

We say that an interpretation A of $P^{-}_{dis}(T)$ is:

- *formula-consistent if satisfies satisfies conditions 1 and 6 ;*
- *consistent if satisfies conditions 1,2,3,4 and 5.*

The formula-consistency property allows to threat interpretations of $P^{-}_{dis}(T)$ as interpretations of Boolean formulae.

Lemma 3.10 *Let M be a formula-consistent interpretation of $P^{-}_{dis}(T)$. It holds that $\texttt{formula}_F \in M$ if and only if $M \models F$.*

Proof. Point 6 of definition 3.9 allow us to use M as an interpretation for Boolean formulas: for every variable $p \in V_{P'(T)}$ exactly one of $\texttt{p}$ and $\texttt{np}$ belongs to M. In the case $\texttt{p} \in M$ we write $M \models p$. In the case $\texttt{np} \in M$ we write $M \models \neg p$.

The proof of the claim is a very trivial induction on F by using point 1 of definition 3.9. As an example, consider the case $F = F_2 \wedge F_2$. By inductive hypothesis we have (i) $\texttt{formula}_{F_1} \in M \leftrightarrow M \models F_1$ and (ii) $\texttt{formula}_{F_2} \in M \leftrightarrow M \models F_2$. Suppose $\texttt{formula}_F \in M$, by point 1 of definition 3.9 we have (iii) $\texttt{formula}_{F_1} \in M$ and (iv) $\texttt{formula}_{F_2} \in M$. From points (i) and (iii) we obtain (v) $M \models F_1$; from points (ii) and (iv) we obtain (vi) $M \models F_2$. From (v) and (vi) we obtain $M \models F_1 \wedge F_2 = F$. Conversely, starting from $M \models F$ we can easily establish that $\texttt{formula}_F \in M$. □

We also prove that the consistency property is a necessary condition for an interpretation A to be an answer set of $P^{-}_{dis}(T)$. This reflects the proof of theorem 2.7.

Lemma 3.11 *Let T be a ground ILP_{LAS} task. Let $A \subseteq HB_{P^{-}_{dis}(T)}$ be an interpretation. If $A \in AS(P^{-}_{dis}(T))$ then A is consistent.*

Proof. Let us analyze the structure of a generic answer set $A \in AS(P^-_{dis}(T))$ using the intuitions of the proof of theorem 2.7. First, the predicate `w` (the saturation predicate), must belong to A (point 4 of definition 3.9) otherwise the rule `w :- not w` would not be satisfied. In order for `w` to be included in A, it must be supported by the rule $\texttt{w :- formula}_{\psi(T)}$, which in turn requires that $\texttt{formula}_{\psi(T)} \in A$ (point 5 of definition 3.9). For every $y \in V_{P(T)}$ it must hold that $\texttt{y} \in A$ and $\texttt{ny} \in A$ (point 2 of definition 3.9), otherwise the corresponding rules `y :- w.  ny :- w.` would not be satisfied. Moreover, for every $h(i) \in V^H_{P'(T)}$ it must hold that exactly one of `h(i)` and `nh(i)` is included in A (point 3 of definition 3.9). If neither is included, then the rule `h(i) | nh(i)` is violated, and if both of them are included, then A is not an answer set because it contradicts the minimality requirement. Regarding point 1 of definition 3.9, we analyze the case such that *expansion*(F) is in $P^-_{dis}(T)$ where $F = F_1 \wedge F_2$ (the other cases are addressed in a similar manner). By definition of the *expansion* procedure we have that the rule R_F defined as $\texttt{formula}_F$ `:-` $\texttt{formula}_{F_1}$, $\texttt{formula}_{F_2}$ is included in the program $P_{dis}(T)$. If it holds $\texttt{formula}_F \in A$, then, the only possibility is that it is supported by the rule R_F and so it must be $\texttt{formula}_{F_1} \in A$ and $\texttt{formula}_{F_2} \in A$. Conversely, if it holds $\texttt{formula}_{F_1} \in A$ and $\texttt{formula}_{F_2} \in A$, then the atom $\texttt{formula}_F$ is supported by the rule R_F and so $\texttt{formula}_F \in A$.

Since A satisfies conditions 1,2,3,4 and 5 of definition 3.9, we conclude that A is consistent. □

Notice that the opposite direction of lemma 3.11 does not hold. In particular, not every consistent interpretation is an answer set of $P^-_{dis}(T)$. This is illustrated on the usual running example T_e.

Running example: a consistent interpretation of $P^-_{dis}(T_e)$

We have already seen that the only answer set of $P^-_{dis}(T_e)$ is:
A_1 : `p np q nq nh(1) nh(2) w` $\texttt{formula}_0$ $\texttt{formula}_1$ $\texttt{formula}_2$ $\texttt{formula}_3$ $\texttt{formula}_4$ $\texttt{formula}_5$ $\texttt{formula}_6$ $\texttt{formula}_7$ $\texttt{formula}_8$ $\texttt{formula}_{12}$
By lemma 3.11, it holds that A_1 is consistent. Consider now the following interpretation:
I_2 : `p np q nq h(2) h(1) w` $\texttt{formula}_0$ $\texttt{formula}_1$ $\texttt{formula}_2$ $\texttt{formula}_3$ $\texttt{formula}_4$ $\texttt{formula}_5$ $\texttt{formula}_8$ $\texttt{formula}_9$ $\texttt{formula}_{10}$ $\texttt{formula}_{11}$ $\texttt{formula}_{12}$
I_2 is not a an answer set of $P^-_{dis}(T_e)$ because is not a model of the reduct $P^-_{dis}(T_e)^{I_2}$ (which is exactly the program $P^-_{dis}(T_e)$ without the rule `w :- not w`). Even if $I_2 \notin AS(P^-_{dis}(T_e))$ it is straightforward to verify that it satisfies points 1,2,3,4 and 5 of definition 3.9 and so I_2 is consistent.

We now reformulate lemma 3.7 in terms of the $\mathsf{ASP}^{\mathsf{D}}$ program $P^-_{dis}(T)$. In particular, given a stable model of the program $P^-_{dis}(T)$ we can compute an hypothesis satisfying cautious conditions and vice-versa.

Lemma 3.12 *Let T be a ground ILP_{LAS} task, let $A' \subseteq HB_{P^-_{dis}(T)}$ be a consistent Herbrand interpretation of $P^-_{dis}(T)$, and define $H \coloneqq \{h_i \mid \texttt{h(i)} \in A'\}$. Then the following are equivalent:*

1. *$A' \in AS(P^-_{dis}(T))$;*
2. *For every $A \in AS(B \cup H)$ and every $e^- \in E^-$, it holds that A does not extend e^- .*

Proof. We have to prove two implications. Again, the proof uses concepts from theorem 2.7 by Eiter and Gottlob [25].

$\boxed{1 \rightarrow 2}$

Suppose that $A' \in AS(P^-_{dis}(T))$. Consider any formula-consistent interpretation I that agrees with A' on the assignment of the atoms `h(i)` and `nh(i)` for every i, does not include `w`, and contains exactly one of `y` and `ny` for every $y \in V_{P(T)}$. Such an interpretation I cannot be a stable model, as otherwise it would contradict the minimality of A'. Since I is not a stable model, the only possibility is that $\texttt{formula}_{\psi(T)} \in I$. By lemma 3.10 it holds that $I \models \psi(T)$. Since the choice over variables `y` and `ny` for every $y \in V_{P(T)}$ was arbitrary, this proves that $M \models \forall\ V_{P(T)}\ \psi(T)$ where M is defined as $M \coloneqq \{h(i) \mid \texttt{h(i)} \in A'\}$. By lemma 3.7 it follows directly that for every $A \in AS(B \cup H)$ and every $e^- \in E^-$, it holds that A does not extend e^- .

$\boxed{2 \rightarrow 1}$

From lemma 3.7 it follows that (i) $M \models \forall\ V_{P(T)}\ \psi(T)$ where $M = \{\texttt{h(i)} \mid h(i) \in A'\}$. Since A' is consistent, and point (i) holds, it follows that A' is a model of $P^-_{dis}(T)^{A'}$. We have to prove that A' is minimal. Notice that $P^-_{dis}(T)^{A'}$ is the program $P^-_{dis}(T)$ without the rule `w :- not w`. Suppose by contradiction that there exists a model J of $P^-_{dis}(T)^{A'}$ such that $J \subseteq A'$. J must coincide with A' on atoms `h(i)` and `nh(i)` for every $h(i) \in V^H_{P'(T)}$. It must be $\texttt{w} \notin J$ and for every $y \in V_{P(T)}$ exactly one atom between `y` and `ny` is included in J. Notice that such a model J is by definition formula-consistent. By point (i) it must be $J \models \psi(T)$ and by lemma 3.10 it follows that $\texttt{formula}_{\psi(T)} \in J$. However this implies $\texttt{w} \in J$ which leads to a contradiction. This proves that A' must be a minimal model of $P^-_{dis}(T)^{A'}$ and so $A' \in AS(P^-_{dis}(T))$. □

From lemma 3.12, it follows that the program $P^-_{dis}(T)$ is correct and complete for tasks T with only negative examples.

Lemma 3.13 *Let T be a ground ILP_{LAS} task. Let T' be the task T without positive examples. It holds that $ILP_{LAS}(T') = \big\{\{h_i \mid \texttt{h(i)} \in A\} \mid A \in AS(P^-_{dis}(T))\big\}$.*

Proof. Let $T = \langle B, S_M, E^+, E^-\rangle$ be a ground ILP_{LAS} task, and let $T' = \langle B, S_M, \emptyset, E^-\rangle$ denote the corresponding task with all positive examples removed. By the definition of the encoding $P^-_{dis}(\cdot)$, it holds that $P^-_{dis}(T) = P^-_{dis}(T')$, since the construction only depends on negative examples. Let $A \in AS(P^-_{dis}(T))$ be an answer set of the encoding. Define the hypothesis $H \coloneqq \{h_i \mid \texttt{h(i)} \in A\}$. By lemma 3.12 (using the implication $1 \rightarrow 2$), we know that for every $A' \in AS(B \cup H)$ and for every $e^- \in E^-$, it holds that A' does not extend e^-. Therefore, by definition of T', H is an inductive solution for T'. Since this holds for every $A \in AS(P^-_{dis}(T))$, it follows that $ILP_{LAS}(T') \supseteq \big\{\{h_i \mid \texttt{h(i)} \in A\} \mid A \in AS(P^-_{dis}(T))\big\}$. Furthermore, following an analogous reasoning using the implication $2 \rightarrow 1$ of lemma 3.12, we prove that $ILP_{LAS}(T') \subseteq \big\{\{h_i \mid \texttt{h(i)} \in A\} \mid A \in AS(P^-_{dis}(T))\big\}$. This concludes the proof. □

We now consider properties of the brave entailment module $P^+_{dis}(T)$.

Lemma 3.14 *Let $A \in AS(P^{+\,'}_{dis}(T))$, let $A_i = \{\texttt{q} \mid \texttt{inAs(q,i)} \in A\}$ and let $H = \{h_i \mid \texttt{h(i)} \in A\}$. It holds that:*

1. *If `interpretation(i)` $\in A$, then $A_i \in AS(B \cup H)$;*
2. *A_i extends the example e_i if and only if it holds `cov(i)` $\in A$.*

Proof. The proof is trivial and similar to the ones by Law [12] for the ILASP1 algorithm. Point 1 follows since each atom `q` in a rule $R \in B \cup S_M$ is encoded by the (non-ground) atom `inAs(q,`M`)` and the encoding of R contains the atom `interpretation(`M`)` in its body. Moreover each rule h_i in S_M is encoded by adding the atom `h(i)` in its body.

We now prove point 2. Consider an example e_i. By applying point 1 to rule $Ex_1(i)$ we have that $A_i \in AS(B \cup H)$. It holds `cov(i)` $\in A$ if and only if for every atom e_j^{incl} in the inclusion set of e_i it holds `inAs(`e_j^{incl}`,i)` $\in A$ and for every atom `inAs(`e_j^{excl}`,i)` in the exclusion set of e_i it holds $e_j^{excl} \notin A_i$. This holds if and only if A_i extends the example e_i.

□

Lemma 3.13 establishes correctness and completeness for the module $P^{-}_{dis}(\cdot)$ when used with tasks with only negative examples. We now establish the counterpart for the module $P^{+}_{dis}(\cdot)$. In particular, from lemma 3.14 it follows that the program $P^{+}_{dis}(T)$ is correct and complete for tasks T with only positive examples.

Lemma 3.15 *Let T be a ground ILP_{LAS} task. Let T' be the task T without negative examples. It holds that* $ILP_{LAS}(T') = \Big\{\{h_i \mid \texttt{h(i)} \in A\} \mid A \in AS(P^{+}_{dis}{}'(T))\Big\}$.

Proof. Let $H \in ILP_{LAS}(T')$ be an inductive solution of T'. We construct an interpretation A of $P^{+}_{dis}{}'(T)$ as follows:

1. `h(i)` $\in A$ if $h_i \in H$;
2. `nh(i)` $\in A$ if $h_i \notin H$;
3. `cov(i)` $\in A$ for every $e_i \in E^+$;
4. `inAs(p,i)` $\in A$ for every atom p in the inclusion set of $e_i \in E^+$;
5. `inAs(q,i)` $\notin A$ for every q in the exclusion set of e_i.

It can be verified that A is a stable model of $P^{+}_{dis}{}'(T)$. In particular, note that for each $h_i \in S_M$, exactly one of `h(i)` and `nh(i)` must belong to A, otherwise, A would not satisfy minimality.

Conversely, suppose $A \in AS(P^{+}_{dis}{}'(T))$, and define the hypothesis $H = \{h_i \mid \texttt{h(i)} \in A\}$. By construction of the encoding (specifically, rule $Ex_1(i)$), we have `interpretation(i)` $\in A$ for every $e_i \in E^+$. Rule $Ex_3(i)$ then ensures that `cov(i)` $\in A$. By lemma 3.14, the interpretation $A_i = \{p \mid$ `inAs(p,i)` $\in A\}$ is a stable model of $B \cup H$ and extends the example e_i. Since this holds for every $e_i \in E^+$, we conclude that H is an inductive solution of T'. □

Lemmas 3.13 and 3.15 establish the correctness and completeness of the two submodules that compose $P_{dis}(\cdot)$. We now show that their combination yields a full encoding that is correct and complete with respect to the inductive solutions of ground ILP_{LAS} tasks.

Theorem 3.16 (Correctness and completeness) *Let $T = \langle B, S_M, E^+, E^- \rangle$ be a ground ILP_{LAS} task. It holds that* $ILP_{LAS}(T) = \{\{h_i \mid \texttt{h(i)} \in A\} \mid A \in AS(P_{dis}(T))\}$.

Proof. Let $T^+ = \langle B, S_M, E^+, \emptyset \rangle$ and $T^- = \langle B, S_M, \emptyset, E^- \rangle$ denote the positive-only and negative-only variants of T, respectively. Then:

$$T^+ = \langle B, S_M, E^+, \emptyset \rangle \quad T^- = \langle B, S_M, \emptyset, E^- \rangle$$

We have that :

$$
\begin{aligned}
ILP_{LAS}(T) &= ILP_{LAS}(T^+) \cap ILP_{LAS}(T^-) && (3.7)\\
&= \left\{\{h_i \mid \texttt{h(i)} \in A\} \mid A \in AS(P_{dis}^{+\prime}(T^+))\right\} \cap \left\{\{h_i \mid \texttt{h(i)} \in A\} \mid A \in AS(P_{dis}^{-}(T^-))\right\} && (3.8)\\
&= \left\{\{h_i \mid \texttt{h(i)} \in A\} \mid A \in AS(P_{dis}^{+\prime}(T^+)) \cap AS(P_{dis}^{-}(T^-))\right\} && (3.9)\\
&= \left\{\{h_i \mid \texttt{h(i)} \in A\} \mid A \in AS(P_{dis}^{+\prime}(T^+) \cup P_{dis}^{-}(T^-))\right\} && (3.10)\\
&= \left\{\{h_i \mid \texttt{h(i)} \in A\} \mid A \in AS(P_{dis}^{+\prime}(T) \cup P_{dis}^{-}(T))\right\} && (3.11)\\
&= \left\{\{h_i \mid \texttt{h(i)} \in A\} \mid A \in AS(P_{dis}^{+}(T) \cup P_{dis}^{-}(T))\right\} && (3.12)\\
&= \{\{h_i \mid \texttt{h(i)} \in A\} \mid A \in AS(P_{dis}(T))\} && (3.13)
\end{aligned}
$$

Equation (3.7) follows directly from the definition of $ILP_{LAS}(T)$ as requiring both brave coverage of positive examples and cautious rejection of negative ones. Equations (3.8) and (3.9) follow from lemmas 3.13 and 3.15, which establish the correctness of the individual positive and negative encodings. To justify Equation (3.10), observe that: (i) $atoms(P_{dis}^{+\prime}(T^+)) \cap atoms(P_{dis}^{-}(T^-)) = \{\texttt{h(i), nh(i)} \mid h_i \in S_M\}$ and (ii) every answer set of each module contains exactly one of `h(i)` or `nh(i)` for each $h_i \in S_M$. As a consequence, the intersection of answer sets from the two modules determines the same hypothesis (i.e., the same assignment to `h(i)` atoms) as any answer set of their union. Equation (3.11) holds because the modules $P_{dis}^{+\prime}(T^+)$ and $P_{dis}^{-}(T^-)$ are constructed to handle only positive and negative examples, respectively. Therefore, their semantics remain unaffected when both modules are used with the full task $T = \langle B, S_M, E^+, E^-\rangle$. Equation (3.12) holds because the rules of the form `h(i) | nh(i)` do not need to appear in both modules. Since they are already included in $P_{dis}^{-}(T)$, they can be safely omitted from $P_{dis}^{+\prime}(T)$, allowing us to use $P_{dis}^{+}(T)$ in its place. Finally, equation (3.13) holds by definition of $P_{dis}(T)$ as the union $P_{dis}^{+}(T) \cup P_{dis}^{-}(T)$. This concludes the proof. □

So far, we have established the correctness and completeness of $P_{dis}(\cdot)$, as previously done for $P_{exp}(\cdot)$ in Section 3.1.2. We now focus on the main advantage of $P_{dis}(\cdot)$: its compact size under suitable conditions. This is formalized in the following lemma.

Lemma 3.17 *Let $T = \langle B, S_M, E^+, E^-\rangle$ be a ground ILP_{LAS} task. Then the following holds:*

1. $|P_{dis}(T)| \in \Theta\left(|T| + |\phi^*_{loop}(P'(T))|\right)$;
2. $|ground(P_{dis}(T))| \in \Theta\left(|B \cup S_M| \times |E^+| + |E^-| + |B| + |S_M| + |\phi^*_{loop}(P'(T))|\right)$;
3. *If $B \cup S_M$ is tight then* $|P_{dis}(T)| \in \Theta\left(|T|\right)$ *and* $|ground(P_{dis}(T))| \in \Theta\left(|B \cup S_M| \times |E^+| + |E^-| + |B| + |S_M|\right)$

.

Proof. The total size of $P_{dis}(T)$ is the sum of the sizes of $P_{dis}^{-}(T)$ and $P_{dis}^{+}(T)$. The number of rules in $P_{dis}^{+}(T)$ is in $\Theta(|B| + |S_M| + |E^+|)$, since each rule and each positive example contributes a constant number of clauses. The module $P_{dis}^{-}(T)$ is obtained by translating the formula

$$\phi(T) = \exists V^H_{P'(T)}\ \forall V_{P(T)}\left(\mathsf{NNF}\left(\phi^*_{as}(P'(T)) \rightarrow \phi_{neg}(E^-)\right)\right)$$

into a ground ASP$^{\mathsf{D}}$ program.

Observe that the NNF transformation is linear in the size of its argument. However, the presence of bi-implications may cause an exponential blow-up during the transformation. We therefore examine the size of $\phi(T)$ more carefully:

$$
\begin{aligned}
|\phi(T)| &= |\mathsf{NNF}(\phi^*_{as}(P'(T)))| + |\mathsf{NNF}(\phi_{neg}(E^-))| && (3.14)\\
&= |\mathsf{NNF}(\phi^*_{iff}(P'(T)))| + |\mathsf{NNF}(\phi^*_{loop}(P'(T)))| + |\mathsf{NNF}(\phi_{neg}(E^-))| && (3.15)\\
&= |\mathsf{NNF}(\phi^*_{iff}(P'(T)))| + \Theta(|\phi^*_{loop}(P'(T))|) + \Theta(|\phi_{neg}(E^-)|) && (3.16)\\
&= |\mathsf{NNF}(\phi^*_{iff}(P'(T)))| + \Theta(|\phi^*_{loop}(P'(T))|) + \Theta(|E^-|) && (3.17)\\
&= \Theta(|\phi^*_{iff}(P'(T))|) + \Theta(|\phi^*_{loop}(P'(T))|) + \Theta(|E^-|) && (3.18)\\
&= \Theta(|B \cup S_M|) + |\phi^*_{loop}(P'(T))| + \Theta(|E^-|) && (3.19)
\end{aligned}
$$

Equation (3.14) reflects the definition of $\phi(T)$. Equation (3.15) follows form the definition of $\phi^*_{as}(P'(T))$ as the conjunction of $\phi^*_{iff}(P'(T))$ and $\phi^*_{loop}(P'(T))$. Equation (3.16) holds because both $\phi^*_{loop}(P'(T))$ and $\phi_{neg}(E^-)$ do not contain bi-implications, and their NNF transformation is therefore linear in size. Equation (3.17) trivially follows by the definition of $\phi_{neg}(E^-)$. We now consider equation (3.18). Notice that $\phi^*_{iff}(P'(T))$ contains exactly one bi-implication for every atom $a \in atoms(P'(T)) \setminus S$ where $S = \{\texttt{h(i)} \mid h_i \in S_M\}$. The structure of $\phi^*_{iff}(P'(T))$ is as follows:

$$\phi^*_{iff}(P'(T)) = \bigwedge_{a \in atoms(P'(T)) \setminus S} lhs(a) \leftrightarrow rhs(a)$$

Each bi-implication can be rewritten as the conjunction of two implications:

$$\phi^*_{iff}(P'(T)) = \bigwedge_{a \in atoms(P'(T)) \setminus S} (lhs(a) \rightarrow rhs(a)) \wedge (rhs(a) \rightarrow lhs(a))$$

This transformation only doubles the size of the formula, and since NNF of an implication is linear, the transformation is still linear in total size. Finally, equation (3.19) follows from the observation that each rule $R \in B \cup S_M$ contributes a bounded number of atoms to the completion $\phi^*_{iff}(P'(T))$. This analysis shows that the size of $P^-_{dis}(T)$ is in $\Theta(|B| + |S_M| + |\phi^*_{loop}(P'(T))| + |E^-|)$. By summing this with the size of $P^+_{dis}(T)$, we obtain the claimed bound for $|P_{dis}(T)|$.

To prove point 2, observe that $P^-_{dis}(\cdot)$ is a ground program. In $P^+_{dis}(T)$, each rule in $B \cup S_M$ is instantiated once for each interpretation label `interpretation(`i`)`, that is, once per positive example.

For point 3, recall that the formula $\phi^*_{loop}(P'(T))$ is empty if and only if the program $B \cup S_M$ is tight, in which case the loop formulas are not required. □

Lemma 3.17 shows that the encoding $P_{dis}(T)$ has linear size whenever the program $B \cup S_M$ is tight. This fact consistently justifies the use of $P_{dis}(T)$ over its counterpart $P_{exp}(T)$, whose size is always exponential in the worst case. A natural theoretical question arises: *Is it possible to construct a polynomial-size encoding for every ground ILP_{LAS} task?* We conjecture that the answer is negative. The intuition behind this conjecture stems from the limitations of propositional encodings of the answer set semantics, as discussed in section 2.1.4. In particular, it is known that, under widely believed

complexity-theoretic assumptions, there exists no polynomial-size Boolean formula that fully captures the answer set semantics of general $\mathsf{ASP}^{\mathsf{N}}$ programs [39]. If this conjecture is confirmed, then the encoding $P_{dis}(\cdot)$ can be considered optimal (up to polynomial equivalence), as it achieves polynomial size exactly when such compression is theoretically possible.

3.3 Extensions

In sections 3.1 and 3.2, we introduced two encodings, $P_{exp}(T)$ and $P_{dis}(T)$, respectively, such that from any answer set of either program we can derive an inductive solution of T and vice-versa. We have proven that this approach is both correct and complete, provided that T is a ground ILP_{LAS} task.

In this section, we address two natural extensions: (i) computing the set of optimal inductive solutions, denoted by ${}^{*}ILP_{LAS}(T)$, and (ii) lifting the restriction that T must be a ground task.

3.3.1 Optimal inductive solutions

Let $P_{enc}(T)$ be either $P_{exp}(T)$ or $P_{dis}(T)$, the encodings presented in sections 3.1 and 3.2, respectively. We now extend them to compute the set of optimal inductive solutions, i.e., inductive hypotheses of minimal length. We define a new program $P^{\mathsf{opt}}_{enc}(T)$ such that the set ${}^{*}AS(T)$ corresponds one-to-one to the set ${}^{*}ILP_{LAS}(T)$, where, for an ASP program P, the set ${}^{*}AS(P)$ denotes the set of optimal answer sets of P.

In particular, let $T = \langle B, S_M = \{h_1, \ldots, h_s\}, E^+, E^- \rangle$ be a ground ILP_{LAS} task. For each $h_i \in S_M$ let $l(h_i)$ denote the length of the rule h_i.

We define the program $P^{\mathsf{opt}}_{enc}(T)$ as follows:

$$P^{\mathsf{opt}}_{enc}(T) \coloneqq P_{enc}(T) \cup \{\ \texttt{\#minimize}\{l(h_1), 1 : \texttt{h(1)}; \ldots; l(h_s), s : \texttt{h(s)}\}.\ \} \qquad (Opt)$$

The `#minimize` directive is a syntactic sugar provided by Clingo [30] to express optimization: the solver seeks answer sets that minimize the total weight, computed as the sum of weights associated with selected atoms. Based on the correctness and completeness of $P_{enc}(T)$, we can easily prove the correctness and completeness of $P^{\mathsf{opt}}_{enc}(T)$ by considering the semantics of the `#minimize` statement.

Theorem 3.18 (Correctness and completeness) *Let $T = \langle B, S_M = \{h_1, \ldots, h_s\}, E^+, E^- \rangle$ be a ground ILP_{LAS} task. Let $P_{enc}(T)$ a correct and complete encoding of T such that $ILP_{LAS}(T) = \{\{h_i \mid \texttt{h(i)} \in A\} \mid A \in AS(P_{enc}(T))\}$. Then, it holds that*

$${}^{*}ILP_{LAS}(T) = \left\{\{h_i \mid \texttt{h(i)} \in A\} \mid A \in {}^{*}AS(P^{\mathsf{opt}}_{enc}(T))\right\} \qquad (3.20)$$

Proof. First notice that the `#minimize` operator is translated into weak constraints and, by definition, a weak constraint does not alter the definition of an answer set. Namely, if we have a program P without weak constraints and a program P' made up only of weak constraints, it holds that $AS(P) = AS(P \cup P')$. Thus, in this case it holds that $AS(P_{enc}(T)) = AS(P^{\mathsf{opt}}_{enc}(T))$ and in particular, by using the correctness

and completeness property of $P_{enc}(T)$ it holds:

$$ILP_{LAS}(T) = \big\{\{h_i \mid \texttt{h(i)} \in A\} \mid A \in AS(P^{\mathsf{opt}}_{enc}(T))\big\} \tag{3.21}$$

The effect of the rule *Opt* is to associate a weight $w(A)$ to every answer set $A \in AS(P^{\mathsf{opt}}_{enc}(T))$. In this case, the weight is defined as :

$$w(A) \coloneqq \sum_{\texttt{h(i)} \in A} l(h_i) \tag{3.22}$$

The set ${}^*AS(P^{\mathsf{opt}}_{enc}(T))$ is defined as:

$${}^*AS(P^{\mathsf{opt}}_{enc}(T)) \coloneqq \big\{A \mid A \in AS(P^{\mathsf{opt}}_{enc}(T)), w(A) = M\big\} \tag{3.23}$$

where $M = \min_{A \in AS(P^{\mathsf{opt}}_{enc}(T))} w(A)$.

To prove equation (3.20), let $H \in {}^*ILP_{LAS}(T)$. From equation (3.21) we know that there exists $A \in AS(P^{\mathsf{opt}}_{enc}(T))$ such that $H = \{h_i \mid \texttt{h(i)} \in A\}$. Since H is an optimal inductive solution it must be that its weight is minimal and so it must hold $w(A) = M$. This proves that $A \in {}^*AS(P^{\mathsf{opt}}_{enc}(T))$.

Conversely, let $A \in {}^*AS(P^{\mathsf{opt}}_{enc}(T))$ and define $H = \{h_i \mid \texttt{h(i)} \in A\}$. By equation (3.21) we have that $H \in ILP_{LAS}(T)$ and since $w(A) = M$ it must hold $H \in {}^*ILP_{LAS}(T)$. □

Since theorems 3.2 and 3.16 prove the correctness and completeness of $P_{exp}(T)$ and $P_{dis}(T)$ respectively, by using theorem 3.18 we obtain that both programs $P^{\mathsf{opt}}_{exp}(T)$ and $P^{\mathsf{opt}}_{dis}(T)$ are correct and complete. Therefore, both encodings $P_{exp}(T)$ and $P_{dis}(T)$ correctly and completely capture the set of optimal inductive solutions, thus matching the capabilities of state-of-the-art systems like ILASP in this respect.

Before concluding this section, we briefly discuss the structural complexity of the program $P^{\mathsf{opt}}_{exp}(T)$ and the computational complexity of the associated minimization statement.

First, the size of the program $P^{\mathsf{opt}}_{exp}(T)$ is $|P_{exp}(T)| + \mathcal{O}(s)$, where s denotes the number of rules in the hypothesis space S_M. This implies that the overall program size remains manageable and does not introduce any significant overhead.

Second, as noted several times throughout the thesis, the satisfiability problem for ILP_{LAS} is Σ^P_2-complete. When dealing with optimal inductive solutions of ground ILP_{LAS} tasks, the relevant problem becomes the *optimum verification problem*, which is known (according to table 2.2 in section 2.2.2) to be Π^P_2-complete [12]. This aligns with the known complexity of deciding whether a given interpretation A is an optimal stable model of a ground $\mathsf{ASP}^{\mathsf{D}}$ program P which is also Π^P_2-complete [2]. Therefore, the $\mathsf{ASP}^{\mathsf{D}}$ framework, when extended with optimization statements, is well suited for solving the optimum verification problem—exactly as done with the encoding $P^{\mathsf{opt}}_{dis}(T)$. If we restrict P to be a ground $\mathsf{ASP}^{\mathsf{N}}$ program with optimization statements, then the problem of deciding whether A is an optimal stable model of P becomes coNP-complete [2]. Using a ground $\mathsf{ASP}^{\mathsf{N}}$ program of exponential size with optimization (as in the case of $P_{exp}(T)$) suffices to achieve the required complexity.

3.3.2 Non-ground tasks

So far, our encodings $P_{exp}(T)$ and $P_{dis}(T)$ have been designed for ground ILP_{LAS} tasks. However, ground tasks are often too restrictive for real-world applications. For example, none of the ILP_{LAS} tasks presented in [20] are ground. The main difficulty with non-ground tasks is that, given a task

$T = \langle B, S_M, E^+, E^- \rangle$, the grounding of $B \cup H$ may differ for each subset $H \subseteq S_M$. This variability makes it challenging to precompute a single grounding that is valid for all possible hypotheses. Since our approach is *single-shot*—that is, we aim to encode the task T into a single ASP program—the need for a uniform grounding appears to limit its applicability. At first glance, a *multi-shot* approach (like the one used by ILASP) might seem more appropriate. Nevertheless, for the class of *safe* non-ground ILP_{LAS} tasks, the grounding of the full program $B \cup S_M$ is sufficient to compute inductive solutions. In what follows, we explore how to perform a single grounding of $B \cup S_M$, and how this grounding can be incorporated into the encoding $P_{enc}(T)$ (where $P_{enc}(T)$ is either $P_{exp}(T)$ or $P_{dis}(T)$). The main problem of the proposed approach is that we require the full instantiation of the program $B \cup S_M$. Thus, we cannot take advantage of advanced grounding techniques available in tools like `gringo` [30] and `dlv` [26].

We begin by introducing the *aggregate LAS* framework, denoted as ILP^{agg}_{LAS}. This framework generalizes ILP_{LAS} by allowing each hypothesis element $h_i \in S_M$ to be a *set* of $\mathsf{ASP}^{\mathsf{N}}$ rules (an aggregation of rules) rather than a single $\mathsf{ASP}^{\mathsf{N}}$ rule, as in the standard setting.

Definition 3.19 (Aggregate Learning From Answer Sets) *An aggregate learning from answer sets (ILP^{agg}_{LAS}) task T is a tuple $\langle B, S_M, E^+, E^- \rangle$ where B is an $\mathsf{ASP}^{\mathsf{N}}$ program, S_M is a set of sets of $\mathsf{ASP}^{\mathsf{N}}$ rules, E^+ and E^- are sets of partial interpretations. A hypothesis $H \subseteq S_M$ is an inductive solution of T if and only if*

1. $\forall e^+ \in E^+ \; \exists A \in AS(B \cup \bigcup_{h_i \in H} h_i)$ *A extends e^+ ;*
2. $\forall e^- \in E^- \; \forall A \in AS(B \cup \bigcup_{h_i \in H} h_i)$ *A does not extend e^- .*

Both encodings $P_{exp}(T)$ and $P_{dis}(T)$ can be adapted to handle ground ILP^{agg}_{LAS} tasks. The only required modification is in how guards are applied: instead of associating each rule with a distinct guard, we assign the same guard atom `h(i)` to all rules in an aggregate hypothesis h_i. That is, for each rule $R \in h_i$, the encoding includes the guard `h(i)` in the body of the rule.

Let T be an ILP_X task. For every subset $S = \{S_{i_1}, \ldots, S_{i_k}\} \subseteq S_M$ of the hypothesis space, we define the set of indices associated with S as:

$$\mathit{indexes}(S) := \{i_1, \ldots, i_k\}.$$

The notion of indexes allows to define *equivalence* between tasks of different frameworks.

Definition 3.20 (Task equivalence) *Let X and Y be two (possibly different) ILP frameworks. Let T be an ILP_X task and let T' be an ILP_Y task. We say that T and T' are equivalent if and only if the following condition holds:*

$$\{\mathit{indexes}(H) \mid H \in ILP_X(T)\} = \{\mathit{indexes}(H) \mid H \in ILP_Y(T')\}$$

The aggregate LAS framework is clearly a generalization of the standard LAS setting. Given a ground ILP_{LAS} task $T = \langle B, S_M, E^+, E^- \rangle$, we define its aggregate version as: $T^{agg} := \langle B, \{\{h_i\} \mid h_i \in S_M\}, E^+, E^- \rangle$. It is straightforward to verify that T and T^{agg} are equivalent.

Our goal is to transform a non-ground ILP_{LAS} task T and into a ground ILP^{agg}_{LAS} task $\mathit{ground}(T)$ such T and $\mathit{ground}(T)$ are equivalent.

Definition 3.21 (Grounding for ILP_{LAS} tasks) *Let $T = \langle B, S_M, E^+, E^- \rangle$ be a (possibly non-ground) ILP_{LAS} task. Let P be the program $B \cup S_M$. Let U be the Herbrand universe of P. We define the grounded version of T as the following ILP^{agg}_{LAS} task:*

$$ground(T) \coloneqq \langle \bigcup_{R_i \in B} ground_U(R_i), \{ground_U(h_i) \mid h_i \in S_M\}, E^+, E^- \rangle$$

where $ground_U(R)$ denotes the set of ground instances of R using all possible terms in U.

We define the notion of *safe* tasks as follows.

Definition 3.22 (Safe ILP_{LAS} tasks) *Let $T = \langle B, S_M, E^+, E^- \rangle$ be a (possibly non-ground) ILP_{LAS} task. We say that T is a safe task if the program $B \cup S_M$ is safe. Namely, for every rule $R \in B \cup S_M$, for every variable X appearing in R, it must hold that X also appears in a positive literal in the body of R.*

The main theorem that we prove is that every safe ILP_{LAS} task T is equivalent to its grounded version.

Theorem 3.23 *Let $T = \{B, S_M = \{h_1, \ldots, h_s\}, E^+, E^-\}$ be a (possibly non-ground) safe ILP_{LAS} task. Let $T' = \{B', S'_M, E^+, E^-\}$ be the ILP^{agg}_{LAS} task ground(T) where:*

- *U is the Herbrand universe of $B \cup S_M$.*
- *$B' \coloneqq \bigcup_{R_i \in B} ground_U(R_i)$*
- *$S'_M \coloneqq \{h'_i \mid h_i \in S_M, h'_i = ground_U(h_i)\}$*

. It holds that T and T' are equivalent.

Proof. Let J be a set of indexes $\{j_1, \ldots, j_k\}$ such that $1 \leq j_i \leq s$ for every $j_i \in J$. Define the corresponding hypothesis for both T and T':

$$H \coloneqq \{h_{j_1}, \ldots, h_{j_k}\} \subseteq S_M \qquad H' \coloneqq \{h'_{j_1}, \ldots, h'_{j_k}\} \subseteq S'_M$$

To prove the theorem we have to prove that H is an inductive solution of T if and only if H' is an inductive solution of T', in symbols:

$$H \in ILP_{LAS}(T) \leftrightarrow H' \in ILP^{agg}_{LAS}(T') \tag{3.24}$$

We begin by studying answer sets of $B \cup H$ and $B' \cup \bigcup_{h'_i \in H'} h'_j$. For simplicity we define the following abbreviations:

$$P = B \cup S_M \qquad P_H = B \cup H \qquad U = HU_P \qquad U_H = HU_{P_H} \qquad P'_H = B' \cup \bigcup_{h'_i \in H'} h'_i$$

Consider the case in which P_H is non-ground. We have that $A \in AS(P_H)$ if and only if $A \in AS(ground_{U_H}(P_H))$. The notation highlights the fact that the ground instance of each rule in P_H is computed considering the Herbrand universe of P_H (i.e. U_H). We now prove the following fact:

$$A \in AS(ground_{U_H}(P_H)) \rightarrow A \in AS(ground_U(P_H)) \tag{3.25}$$

To prove equation (3.25), we proceed by induction. In particular, let $S \subseteq P$ and let $S^+ = S \cup \{h\}$ where $h \in P \setminus S$, we have to prove that if $A \in AS(ground_{HU_S}(P_H))$ then $A \in AS(ground_{HU_{S^+}}(P_H))$. If it holds $HU_S = HU_{S^+}$, i.e. h does not introduce new terms, then trivially we have $A \in AS(ground_{HU_{S^+}}(P_H))$. Consider now the case where $HU_S \subsetneq HU_{S^+}$ and in particular let $V = HU_{S^+} \setminus HU_S$ the set of terms in HU_{S^+} but not in HU_S. Let W denote the set of non-ground atoms appearing in P_H. In program $ground_{HU_{S^+}}(P_H)$, atoms in W can be instantiated also with terms in V. Denote with Z the ground instances of atoms in W such that, each $z_i \in Z$ contains at least one term in V, i.e, z_i is of type $z_i(\dots, v_j, \dots)$ for some $v_j \in V$. Suppose to prove that (i) each atom in Z cannot be supported by $ground_{HU_{S^+}}(P_H)$. Consider a rule $R \in ground_{HU_{S^+}}(P_H) \setminus ground_{HU_S}(P_H)$. By definition R must contain an atom in $z_i(\dots, v_j, \dots) \in Z$ and since the task is safe it must be that (ii) there must exists $z_i'(\dots, v_j, \dots) \in Z \cap body^+(R)$, i.e., there exists a positive literal $z_i' \in Z$ occurring in the body of R and such that z_i' contains the term v_j. By combining (i) and (ii) we obtain that R can never be activated and hence $A \in AS(ground_{HU_{S^+}}(P_H))$. To prove point (i), notice that, trivially, (iii) $ground_{HU_S}(P_H)$ cannot support any atom in Z. Interestingly, also every rule in $K \coloneqq ground_{HU_{S^+}}(P_H) \setminus ground_{HU_S}(P_H)$ cannot support an atom in Z. Consider $G(K, Z)$ as the following graph (which is a slight modification to the positive atom dependency graph of definition 2.9 in section 2.1.4):

$$G(K, Z) = (Z, \big\{(b, h) \mid R \in K, b \in body^+(R), h = head(R), b \in Z, h \in Z\big\})$$

Let $G^c(K, Z)$ be the condensation of $G(K, Z)$, i.e. the graph in which each strongly connected component is contracted to a single vertex. Let $\pi = \langle C_1, \dots, C_k \rangle$ be a topological sorting of $G^c(K, Z)$. Notice that (by using property (iii)) each atom in C_1 cannot be supported by any rule in $ground_{HU_{S^+}}(P_H)$. By a simple inductive argument it holds that each atom in C_i cannot be supported in $ground_{HU_{S^+}}(P_H)$. This proves property (i) and equation (3.25).

The converse of equation (3.25) also holds. In particular, we have:

$$A \in AS(ground_{P_H}(P_H)) \leftarrow A \in AS(ground_P(P_H)) \tag{3.26}$$

Equation (3.26) is proved in a similar manner. We proceed by induction by removing rules from P and gradually arriving to P_H, in particular we prove that if $A \in AS(ground_S(P_H))$ then $A \in AS(ground_{S^-}(P_H))$ where $S \subseteq S_M$ and $S^- = S \setminus \{h\}$ for $h \in S \setminus P_H$. This is trivially verified if $HU_S = HU_{S^-}$. Consider the case $HU_S \supsetneq HU_{S^-}$ and let $V = HU_S \setminus HU_{S^-}$. This case is addressed exactly like in the proof of equation (3.25). The intuition is that rules in $K = ground_S(P_H) \setminus ground_{S^-}(P_H)$ can never be activated since (iv) $ground_{S^-}(P_H)$ cannot support any atom with a term in V and (v) at least one term in V appears in the body of every rule in K (since T is safe). This proves equation (3.26).

The key intuition behind equation (3.25) and equation (3.26) is that, due to the safety of the task, rules involving new terms introduced by a larger Herbrand universe are not activatable and thus do not influence the stable models.

We can now prove that an answer set of P_H is also an answer set of P'_H and vice-versa:

$$
\begin{aligned}
& A \in AS(P_H) \\
\leftrightarrow\ & A \in AS(ground_{U_H}(P_H)) \\
\leftrightarrow\ & A \in AS(ground_U(P_H)) \\
\leftrightarrow\ & A \in AS(ground_U(B \cup H)) \\
\leftrightarrow\ & A \in AS(\bigcup_{R_i \in B} ground_U(R_i) \cup \bigcup_{h_i \in H} ground_U(h_i)) \\
\leftrightarrow\ & A \in AS(B' \cup \bigcup_{h'_i \in H'} h'_i) \\
\leftrightarrow\ & A \in AS(P'_H)
\end{aligned}
$$

We now conclude the proof of the theorem by proving equation (3.24):

$$
\begin{aligned}
& H \in ILP_{LAS}(T) \\
\leftrightarrow\ & \forall e^+ \in E^+\ \exists A \in AS(P_H)\ A \text{ extends } e^+ \text{ and } \forall e^- \in E^-\ \forall A \in AS(P_H)\ A \text{ does not extend } e^- \\
\leftrightarrow\ & \forall e^+ \in E^+\ \exists A \in AS(P'_H)\ A \text{ extends } e^+ \text{ and } \forall e^- \in E^-\ \forall A \in AS(P'_H)\ A \text{ does not extend } e^- \\
\leftrightarrow\ & H' \in ILP^{agg}_{LAS}(T')
\end{aligned}
$$

□

Theorem 3.23 provides a strong result, establishing the equivalence between a (possibly non-ground) ILP_{LAS} task and its grounded counterpart, constructed via single-shot grounding. This result is significant not only for the encoding techniques developed in this work, but also from a theoretical perspective within the broader field of Learning from Answer Sets.

As a direct consequence of theorem 3.23, we now show that, given a correct and complete encoding P_{enc} for the grounded framework ILP^{agg}_{LAS}, the composition $P_{enc}(ground(\cdot))$ yields a correct and complete encoding for every safe ILP_{LAS} task.

Theorem 3.24 (Correctness and completeness) *Let $T = \langle B, \{h_1, \ldots, h_s\}, E^+, E^- \rangle$ be a (possibly non-ground) safe ILP_{LAS} task. Let $P_{enc}(\cdot)$ be a correct and complete encoding for ground tasks in the ILP^{agg}_{LAS} framework, i.e., for every ground task $T' = \langle B', \{h'_1, \ldots, h'_s\}, E'^+, E'^- \rangle$, it holds that: $ILP^{agg}_{LAS}(T') = \{\{h'_i \mid \texttt{h'(i)} \in A\} \mid A \in AS(P_{enc}(T'))\}$ Then, the composition $P_{enc}(ground(T))$ is a correct and complete encoding for T, that is:*

$$ILP_{LAS}(T) = \{\{h_i \mid \texttt{h(i)} \in A\} \mid A \in\ AS(P_{enc}(ground(T)))\} \tag{3.27}$$

Proof. Let $T'' = \langle B'', \{h''_1, \ldots, h''_n\}, E^+, E^- \rangle$ be the task ILP^{agg}_{LAS} $T'' = ground(T)$. Let . By theorem

We have that:

$$H = \{h_{j_1}, \ldots, h_{j_1}\} \in ILP_{LAS}(T) \tag{3.28}$$

$$\leftrightarrow H'' = \left\{h''_{j_1}, \ldots, h''_{j_1}\right\} \in ILP^{agg}_{LAS}(\mathit{ground}(T)) \tag{3.29}$$

$$\leftrightarrow \exists A \in AS(P_{enc}(T'')) \text{ such that } \left\{\texttt{h''(i)} \mid h''(i) \in A\right\} = H'' \tag{3.30}$$

Equation 3.29 follows from theorem 3.23. Equation 3.30 follows from the fact that $P_{enc}(\cdot)$ is a correct and complete encoding for ground ILP^{agg}_{LAS} tasks. □

Theorems 3.2 and 3.16 establish that both encodings, $P_{exp}(\cdot)$ and $P_{dis}(\cdot)$, are correct and complete with respect to ground ILP_{LAS} tasks. These results can be naturally extended to the aggregated framework ILP^{agg}_{LAS}: by incorporating the modification described immediately after definition 3.19, both encodings remain correct and complete for ground ILP^{agg}_{LAS} tasks. Combining this observation with theorem 3.24, we obtain correctness and completeness for the composed encodings $P_{exp}(\mathit{ground}(T))$ and $P_{dis}(\mathit{ground}(T))$ with respect to all safe (possibly non-ground) ILP_{LAS} tasks. Finally, as expected, the proposed encodings can also be used to compute optimal solutions. In particular, the composed encodings $P^{\mathsf{opt}}_{exp}(\mathit{ground}(T))$ and $P^{\mathsf{opt}}_{dis}(\mathit{ground}(T))$, presented in section 3.3.1, can be applied to capture optimal inductive solutions for safe ILP_{LAS} tasks. This matches the capabilities of existing ILASP algorithms, which are guaranteed to return optimal inductive solutions for safe tasks.

Before moving on to the next chapter, which introduces the `lasco` tool, we highlight once more a major limitation of the grounding approach presented here: it requires full instantiation of the program $B \cup S_M$. This requirement limits the power of current ASP solvers, which employ various optimizations to avoid full grounding [47], as full enumeration is often computationally prohibitive. The need for complete grounding arises from a fundamental modelling constraint: rules in B must remain agnostic to the content of S_M, and each hypothesis $h_i \in S_M$ must in turn be blind to other hypotheses.

4

Implementation and Experiments

In this chapter, we move from theory to practice. Specifically, we take each idea introduced in chapter 3 and provide a concrete implementation.

First, we present the tool `lasco` (*LAS to ASP Compiler*), a solver for safe ILP^{ctx}_{LAS} tasks that implements the encodings introduced earlier. The code and precompiled binaries for major desktop platforms are publicly available at `https://github.com/robyBorelli/lasco`. `lasco` supports common ASP extensions such as integer arithmetic and choice rules, and its features are therefore comparable to those of ILASP—although ILASP remains a more mature and refined system.

Secondly, we evaluate `lasco` on a set of representative instances and compare its performance to ILASP, the state-of-the-art ILP_{LAS} solver. Although ILASP remains a more refined and generally more efficient system overall, our experiments reveal that `lasco` clearly outperforms all versions of ILASP on several benchmark instances, in particular, on tight tasks. In these cases, `lasco` is also able to solve instances that ILASP fails to handle within reasonable time or resource limits. These results demonstrate that, despite its younger development stage, `lasco` can surpass the state-of-the-art in ILP_{LAS} solving on non-trivial tasks, confirming its practical competitiveness.

4.1 The tool `lasco`

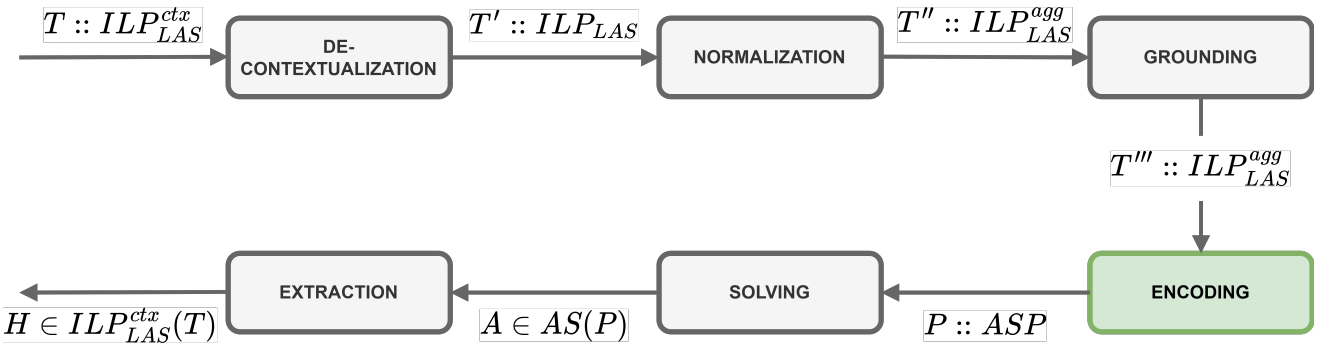


Figure 4.1: Overview of the `lasco` pipeline. Given an input task T in the ILP^{ctx}_{LAS} framework, the tool returns an inductive solution $H \in ILP^{ctx}_{LAS}(T)$.

In this section, we describe in detail the `lasco` tool. We start by outlining the main steps of its

execution pipeline, as illustrated in figure 4.1. The procedure carried out by `lasco` consists of six basic steps. Let $T = \langle B, S_M, E^+, E^- \rangle$ be a safe ILP^{ctx}_{LAS} task, i.e., an ILP_{LAS} task with context-dependent examples (as defined in definition 2.19). The task T may include normal rules as well as standard ASP extensions, such as integer arithmetic and choice rules. A formal specification of the input syntax is given in section 4.1.2.

1. **De-contextualization.** This phase transforms the task T into an equivalent task T' in the ILP_{LAS} framework. The transformation is the one described in the proof of theorem 2.21.

2. **Normalization.** The normalized task T' is converted into an equivalent task T'' in the ILP^{agg}_{LAS} framework, where each rule in the background knowledge and in the hypothesis space is a normal rule. The reason T'' is an ILP^{agg}_{LAS} task and not a standard ILP_{LAS} task is that some ASP constructs (such as choice rules) are translated into multiple normal rules. These transformations are standard in the ASP literature and are discussed in [2].

3. **Grounding.** The task T'' is transformed into a ground ILP^{agg}_{LAS} task T'''. This step follows the notion of grounding introduced in section 3.3.2. Note that the transformation $ground(\cdot)$ from definition 3.21 is originally defined for safe ILP_{LAS} tasks; however, it naturally generalizes to the aggregate framework while preserving correctness and completeness (more details will be given in section 4.1.4). This module is implemented using `gringo`, forcing it to return the full instantiation of the rules, as required by definition 3.21.

4. **Encoding.** The grounded task T''' is translated into an ASP program P, such that the answer sets of P correspond one-to-one with the inductive solutions of T''', and hence of T. This phase can be carried out using either the encoding $P_{exp}(\cdot)$ from section 3.1 or the disjunctive encoding $P_{dis}(\cdot)$ from section 3.2. In the former case, P is an $\mathsf{ASP}^{\mathsf{N}}$ program of exponential size in $|T'''|$; in the latter, P is an $\mathsf{ASP}^{\mathsf{D}}$ program of linear size in $|T'''|$. The `lasco` tool supports both encodings, but the default is $P_{dis}(\cdot)$ for obvious scalability reasons. If the user wishes to compute optimal inductive solutions, the optimized versions $P^{\mathsf{opt}}_{exp}(\cdot)$ and $P^{\mathsf{opt}}_{dis}(\cdot)$ can be used, as discussed in section 3.3.1.

5. **Solving.** This phase computes the answer sets of the program P. The solving is delegated to an external ASP solver such as `clingo` or `dlv`. Note that if P contains the optimization statement `#minimize` of $P^{\mathsf{opt}}_{exp}(\cdot)$ or $P^{\mathsf{opt}}_{dis}(\cdot)$, it must be translated into weak constraints when using `dlv`, as that syntax is specific to `clingo`. The user can specify the desired solver through the command-line interface (described in section 4.1.3). Additionally, the number of answer sets to compute can be specified, depending on how many inductive solutions are desired.

6. **Extraction.** From an answer set A of P, we extract an inductive solution H_A of T, defined as: $H_A = \{h_i \in S_M \mid \texttt{h(i)} \in A\}$.

As described in the six-step pipeline, the *grounding* phase requires the installation of `gringo`, while the *encoding* phase relies on either `clingo` or `dlv` to solve the resulting ASP program. We tested `lasco` using the following toolchain versions:

- `gringo` version 5.7.1, 64-bit (libgringo 5.7.1, configured with Python 3.12.1, without Lua)

- `clingo` version 5.7.1, 64-bit (libclingo 5.7.1, with Python 3.12.1, without Lua; libclasp 3.3.10; libpotassco 1.1.0)
- `dlv` version 2.1.2

It is worth noting, however, that for ground ILP^{ctx}_{LAS} tasks, `lasco` can be used as a stand-alone tool by stopping at phase 4, where it outputs the corresponding ASP encoding P.

After this high-level overview, we now proceed with a more detailed description of the implementation. In particular, we illustrate the structure of the code in section 4.1.1, the input syntax supported by `lasco` in section 4.1.2, the command-line interface in section 4.1.3, and other relevant implementation details in section 4.1.4.

4.1.1 Organization of the code

The code is written in `Haskell` [48]. The main reasons behind this choice are as follows:

- Haskell supports all major desktop platforms, including Windows, Linux, and macOS.
- It is well suited for building compilers, especially when combined with tools such as `BNFC` [49], `Happy` [50], and `Alex` [51].
- It provides a natural way to implement mathematical definitions, and the resulting code is efficient, although generally slower than equivalent C++ implementations.

Below, we list the main source files that make up the codebase.

```
code/
├── DisjunctiveEncoder.hs
├── Encoder.hs
├── EncoderUtilities.hs
├── ExpEncoder.hs
├── Graph.hs
├── Grounder.hs
├── InputParser.hs
├── Logic.hs
├── Main.hs
├── Normalizer.hs
├── Parser/
│   ├── AbsGrammar.hs
│   ├── ErrM.hs
│   ├── Grammar/
│   │   └── grammar.cf
│   ├── LexGrammar.x
│   ├── LexUtilities.hs
│   ├── ParGrammar.y
│   ├── PrintGrammar.hs
│   ├── ShallowLexGrammar.x
│   └── SkelGrammar.hs
└── Solver.hs
```

We now describe the main functionality of the core modules:

- The `Parser` directory contains the modules responsible for parsing input tasks. These files are generated with the help of `BNFC`, starting from the grammar specification

`code/Parser/Grammar/grammar.cf`. The main lexer is defined in `code/Parser/LexGrammar.x`, while the main parser is in `code/Parser/ParGrammar.y`. An additional lexer, `code/Parser/ShallowLexGrammar.x`, is used specifically during the grounding phase. The module `code/Parser/AbsGrammar.hs` defines the main data types used to represent an ILP task.

- The command-line interface is handled by the module `code/InputParser.hs`, which parses and validates user-provided arguments.

- The normalization phase is implemented in the module `code/Normalizer.hs`.

- Interaction with external tools is managed by `code/Grounder.hs` (for grounding) and `code/Solver.hs` (for solving).

- The encodings $P_{dis}(\cdot)$ and $P_{exp}(\cdot)$ are implemented in `code/DisjunctiveEncoder.hs` and `code/ExpEncoder.hs`, respectively. Common utilities used by both encoders are defined in `code/EncoderUtilities.hs`.

- The disjunctive encoding module `code/DisjunctiveEncoder.hs` manipulates logic formulas and graph structures. These are handled by the modules `code/Logic.hs` and `code/Graph.hs`, respectively. The former implements the procedure for transforming formulas into negation normal form (NNF), while the latter includes an algorithm for computing all loops in a directed graph (details are provided in section 4.1.4).

- Finally, the six-step execution pipeline described in figure 4.1 is orchestrated by the main module `code/Main.hs`.

4.1.2 Input syntax

We now describe in detail how a `Task` is represented in the `lasco` tool. The input syntax used by `lasco` is a simplified subset of the one adopted by ILASP, and is designed so that every `Task` accepted by `lasco` can also be processed by ILASP. The converse, however, does not hold in general: ILASP supports additional constructs that are not handled by `lasco`. The most notable difference is the use of *mode declarations* in ILASP, which provide a compact specification of the hypothesis space [12]. Nevertheless, given a task T that includes mode declarations, one can run the command `ILASP -s task.las` to expand the full hypothesis space into an explicit representation that is compatible with `lasco`.

We now proceed to describe the grammar (whose start symbol is `Task`) that defines the accepted input syntax.

Comments and reserved words. Comments start with the character `%` and extend to the end of the line. Any identifier that contains the substring `lasco` or `LASCO` is considered a reserved word. This restriction is necessary because the `lasco` tool internally introduces auxiliary symbols using such prefixes.

Tokens, terms, atoms, and literals. The two main types of tokens are `BasicSymbol` and `VariableSymbol`. A `BasicSymbol` is an identifier that starts with a lowercase letter, followed by any number of letters, digits, or underscores. A `VariableSymbol` is defined similarly, but starts with an uppercase letter.

A term is either a constant (represented with a basic symbol), an arithmetic expression or a tuple with arity grater than one:

```
Term  := BasicSymbol                                          (ConstantTerm)
Term  := ArithmeticExpr                                       (ArithmeticTerm)
Term  := "(" Term_1 "," ...  "," Term_n ")"                   (TupleTerm)
```

We distinguish between two types of atoms *SimpleAtom* and *CompositeAtom*:

```
Atom  := BasicSymbol                                          (SimpleAtom)
Atom  := CompositeAtom "(" Term_1 "," ...  "," Term_n ")"     (CompositeAtom)
```

We have three types of literals, *PositiveLiteral*, *NegativeLiteral* and *ComparisonLiteral*:

```
Literal  := Atom                                              (PositiveLiteral)
Literal  := "not" Atom                                        (NegativeLiteral)
Literal  := Term CompOp Term                                  (ComparisonLiteral)
```

where `CompOp` is a comparison operator of the form `<`, `<=`, `=`, `==`, `!=`, `=>`, `>`.

An arithmetic expression `ArithmeticExpr` is of the form:

```
ArithmeticExpr  := ArithmeticExpr_1 MathOp ArithmeticExpr_2   (BinaryExpr)
ArithmeticExpr  := "-" ArithmeticExpr_1                       (UnaryMinus)
ArithmeticExpr  := "|" ArithmeticExpr_1 "|"                   (AbsValue)
ArithmeticExpr  := VariableSymbol                             (Variable)
ArithmeticExpr  := Integer                                    (IntConstant)
```

where `MathOp` is a mathematical operator of the form `+`, `+`, `*`, `/`. Operator precedence is standard, and an `ArithmeticExpr` can be enclosed in `"("` and `")"`.

ASP declarations. The `Head` of an asp declaration is either an `Atom` or a *ChoiceHead*:

```
Head  := Atom                                                 (SimpleHead)
Head  := Bound "{" HeadElement_1 ";" ...  ";" HeadElement_n "}" Bound   (ChoiceHead)
```

where `Bound` is either an integer or the empty word:

```
Bound  := Integer                                             (ExplicitBound)
Bound  :=                                                     (ImplicitBound)
```

and an `HeadElement` is either an `Atom` or a guarded atom:

```
HeadElement  := Atom                                          (SimpleHeadElement)
HeadElement  := Atom ":" Literal_1 "," ...  "," Literal_n     (GuardedHeadElement)
```

ASP rules follow standard syntax:

```
AspRule  := Head                                                    (Fact)
AspRule  := Head ":-" Literal_1 "," ...  "," Literal_n              (NormalRule)
AspRule  := ":-" Literal_1 "," ...  "," Literal_n                   (Denial)
AspRule  := BasicSymbol "(" Integer ".." Integer ")"                (Range)
```

these correspond to facts, normal rules, denials, and range declarations respectively.

Hypotheses. A `Hypothesis` declaration associates a weight (an integer) to an `AspRule`:

```
Hypothesis  := Integer ∼ AspRule                                    (Hypothesis)
```

Examples. Examples can be either positive or negative and may include an optional context:

```
Example  := "#pos({" Atoms "},{" Atoms "})"                         (PositiveExample)
Example  := "#neg({" Atoms "},{" Atoms "})"                         (NegativeExample)
Example  := "#pos({" Atoms "},{" Atoms "},{" AspProgram "})"        (CtxPositiveExample)
Example  := "#neg({" Atoms "},{" Atoms "},{" AspProgram "})"        (CtxNegativeExample)
```

Here, `Atoms` is a comma-separated list of atoms (possibly empty), and `AspProgram` is a list of ASP rules:

```
Atoms       := Atom_1 "," ...  "," Atom_n                           (AtomList)
AspProgram  := AspRule_1 "." ...  AspRule_n "."                     (Program)
```

Program structure. An input task is a sequence of declarations, each terminated by a period:

```
Task  := Declaration_1 "." ...  Declaration_n "."                   (Task)
```

where each declaration can be:

```
Declaration  := AspRule                                             (AspDeclaration)
Declaration  := Hypothesis                                          (HypothesisDeclaration)
Declaration  := Example                                             (ExampleDeclaration)
```

Before concluding this section, we show how to represent the task T_e using the input syntax described so far.

Running example: representation of the task T_e

To fully understand the input format accepted by `lasco`, we now provide the complete representation of the running task T_e, introduced in section 2.2.1 (see page 14).

```
example.las:

   %% Background knowledge
   p :- not q.
   %% Hypothesis
   1 ~ q.
   2 ~ q :- not p.
   %% Examples
   #pos({p},{q}).
   #neg({q},{p}).
```

4.1.3 Command-line interface

The `lasco` tool is executed via a command-line interface and provides several options to configure its behavior. The general usage is:

```
Usage: lasco [options]
```

The available command-line options are described below:

- `-i`, `--input <IFILE>`
  Specifies the input file `<IFILE>` containing the learning task to be solved. The input file must follow the syntax described in section 4.1.2. If no input file is provided, then the input is read from `stdin`.
- `-o`, `--output <OFILE>`
  Specifies the output file `<OFILE>` where the resulting ASP encoding will be written. If no output file is provided, then the output is written to `stdout`.
- `-e`, `--encoder <ENCODER>`
  Selects the encoding type to use. Supported values are:
  - `exponential`
  - `disjunctive` (default)
- `--solve <SOLVER>`
  Invokes an external ASP solver to solve the generated encoding. Supported solvers:
  - `clingo`
  - `dlv`
- `--solve-mode <MODE>`
  Specifies the solving mode. Accepted values are:

  - `first` (default) — compute the first solution
  - `all` — compute all solutions
  - `optimum` — compute an optimal solution
  - an integer (e.g., `3`) — compute a specific number of solutions

- `-v`, `--verbose`
  Enables verbose output, including intermediate representations and debugging information.

- `--enable-comments`
  Includes inline comments in the generated ASP encoding to improve readability.

- `--show-hypos [i1,i2,...]`
  Displays the hypotheses with the specified indices and exits, without solving the task.

- `-h`, `--help`
  Prints a help message and exits.

Note that if the `--solve` option is not specified, the pipeline illustrated in figure 4.1 stops after the encoding stage.

These options highlight the modularity of the `lasco` tool. In particular, the user can choose between two types of encoding, two different external solvers, and multiple solving modes, including `first`, `all`, or `optimum`. In this sense, `lasco` offers more flexibility than the ILASP system, which relies exclusively on `clingo` and is primarily designed to compute optimal inductive solutions. In contrast, `lasco` also supports the `dlv` solver and allows for the computation of non-optimal solutions as well. Moreover, the modular structure of the codebase makes it straightforward to extend the system with new encoders or support for additional solvers. Each component of the pipeline is implemented as an independent module, which facilitates experimentation and future development.

Before concluding this section, we illustrate how to run the program on our standard running example.

Running example: running `lasco` on the task T_e

Suppose that the task T_e is saved in a file named `example.las`, written using the syntax described in section 4.1.2.
To print the disjunctive encoding, execute:

```
lasco -i example.las
```

To solve the task using `clingo`, run:

```
lasco -i example.las --solve clingo
```

Since the Herbrand base of T_e is small, it is also feasible to use the exponential encoding:

```
lasco -i example.las --encode exponential
```

We can verify that solving the exponential encoding yields the same (empty) inductive solution:

```
lasco -i example.las --encode exponential --solve clingo
```

Finally, note that since T_e is a ground task, contains only normal rules, and includes no context-dependent examples, `lasco` directly executes the encoding module, skipping the de-contextualization, normalization, and grounding stages.

4.1.4 Other details

In this final subsection dedicated to the presentation of `lasco`, we address several implementation and design aspects that were deliberately omitted earlier to avoid overburdening the reader. Specifically, we cover: (i) the grounding procedure required by the `lasco` pipeline (see figure 4.1) for handling tasks in the ILP^{agg}_{LAS} framework; (ii) a simple algorithm to enumerate all cycles in a graph, which is used in the loop formula of the disjunctive encoding; and (iii) the presence of an additional lexer, `ShallowLexGrammar.x`, in the codebase (see also section 4.1.1).

These topics, while not essential for understanding the overall design and contribution of this thesis, may nonetheless be of interest to some readers. Readers not interested in these technical aspects may safely skip this section without losing the general meaning and structure of the work.

Grounding of ILP^{agg}_{LAS} tasks As shown in figure 4.1, the grounding step of the `lasco` pipeline takes as input a safe task T'' belonging to the ILP^{agg}_{LAS} framework and produces as output an equivalent ground ILP^{agg}_{LAS} task T'''. However, the grounding theory introduced in section 3.3.2 addresses only the case of safe ILP_{LAS} tasks, where each hypothesis is a single rule. In this section, we extend that theory to handle the more general ILP^{agg}_{LAS} setting, in which each hypothesis may consist of a set of rules. This extension is necessary to ensure the correctness of the grounding procedure in the broader context adopted by `lasco`.

We generalize definition 3.21 as follows.

Definition 4.1 (Grounding for ILP^{agg}_{LAS} tasks) *Let $T = \langle B, S_M, E^+, E^- \rangle$ be a (possibly non-ground) ILP^{agg}_{LAS} task. Let P be the program $B \cup \bigcup_{h_i \in S_M} h_i$. Let U be the Herbrand universe of P. We define the grounded version of T as the following ILP^{agg}_{LAS} task:*

$$ground(T) \coloneqq \left\langle \bigcup_{R_i \in B} ground_U(R_i), \left\{ \bigcup_{R_j \in h_i} ground_U(R_j) \mid h_i \in S_M \right\}, E^+, E^- \right\rangle$$

where $ground_U(R)$ denotes the set of ground instances of R using all possible terms in U.

In other words, grounding an ILP^{agg}_{LAS} task requires that the grounding of each set of rules in the hypothesis space be treated as an atomic unit: all ground instances of a rule set must remain grouped together within the same element of the hypothesis space.

With this perspective, theorem 3.23 can also be naturally extended to cover the ILP^{agg}_{LAS} setting.

Theorem 4.2 *Let $T = \langle B, S_M, E^+, E^- \rangle$ be a (possibly non-ground) safe ILP^{agg}_{LAS} task. Let $T' = \langle B', S'_M, E^+, E^- \rangle$ be the ILP^{agg}_{LAS} task $ground(T)$. It holds that T and T' are equivalent.*

Proof. The proof is similar to the one of theorem 3.23.

Let $P = B \cup \bigcup_{h_i \in S_M} h_i$ and let $U = HU_P$. Following definition 4.1, for $h_i \in H$ we define $h'_i \in H'$ as $h'_i = \bigcup_{R_j \in h_i} ground_U(R_j)$. Let J be a set of indexes $\{j_1, \ldots, j_k\}$ such that $1 \leq j_i \leq |S_M|$ for every $j_i \in J$. Define the corresponding hypothesis for both T and T':

$$H \coloneqq \{h_{j_1}, \ldots, h_{j_k}\} \subseteq S_M \qquad H' \coloneqq \{h'_{j_1}, \ldots, h'_{j_k}\} \subseteq S'_M$$

To prove the theorem we have to prove that H is an inductive solution of T if and only if H' is an inductive solution of T'. We define the following abbreviations:

$$P_H = B \cup \bigcup_{h_i \in H} h_i \qquad U_H = HU_{P_H} \qquad P'_H = B' \cup \bigcup_{h'_i \in H'} h'_i$$

We can now prove that an answer set of P_H is also an answer set of P'_H and vice-versa:

$$\begin{aligned}
& A \in AS(P_H) \\
\leftrightarrow^{(1)}\ & A \in AS(ground_{U_H}(P_H)) \\
\leftrightarrow^{(2)}\ & A \in AS(ground_U(P_H)) \\
\leftrightarrow^{(3)}\ & A \in AS(ground_U(B \cup \bigcup_{h_i \in H} h_i)) \\
\leftrightarrow^{(4)}\ & A \in AS(\bigcup_{R_i \in B} ground_U(R_i) \cup \bigcup_{h_i \in H} \bigcup_{R_j \in h_i} ground_U(R_j)) \\
\leftrightarrow^{(5)}\ & A \in AS(B' \cup \bigcup_{h'_i \in H'} h'_i) \\
\leftrightarrow^{(6)}\ & A \in AS(P'_H)
\end{aligned}$$

The equivalence $\leftrightarrow^{(2)}$ follows from equations (3.25) and (3.26) of the proof of theorem 3.23. Having established that $A \in AS(P_H)$ if and only if $A \in AS(P'_H)$, it follows that $H \in ILP^{agg}_{LAS}(T)$ if and only if $H' \in ILP^{agg}_{LAS}(T')$, which completes the proof. □

Enumerating all loops in a graph As discussed in section 3.2, the disjunctive encoder requires computing the set of all loops defined by the positive atom dependency graph. Recall that, given a directed graph $G = (V, E)$, a subset of nodes $V' \subseteq V$ defines a loop in G if it induces a non-trivial strongly connected subgraph, where a subgraph is trivial if it consists of a single node v and no edges, i.e., $(\{v\}, \emptyset)$.

The most straightforward approach to identify all loops is to enumerate every subset $V' \subseteq V$ and check whether the induced subgraph is strongly connected and non-trivial. However, this brute-force strategy is impractical in real-world scenarios, particularly when G is acyclic or contains only a few loops.

To address this issue, we introduce algorithm 5, which still guarantees completeness (i.e., all loops in G are found) but is significantly more efficient in sparse or nearly acyclic graphs. This design choice ensures that the module `DisjunctiveEncoder.hs` (see section 4.1.1) can compute the loop formula $\phi^*_{loop}(P)$ of a program P efficiently, even when P is tight or only weakly non-tight.

The algorithm proceeds as follows. It begins by computing all strongly connected components

Algorithm 5 Enumerating all cycles of a graph

```
Input: A direct graph G = (V, E)
Output: The se of all loops of G
L ← ∅
SCCs ← find all strongly connected components of G                ▷ Tarjan's algorithm
for S ∈ SCCs do
    V' ← the set of nodes of S
    if |V'| = 1 then
        v ← the only node in V'
        if (v, v) ∈ E then                                        ▷ Node v has a self loop
            L ← L ∪ {v}
        else                                                      ▷ {v} is a trivial strongly connected component
            continue
        end if
    else
        for V_S ⊆ V' : V_S ≠ ∅ do
            G_S ← (V_S, E ∪ (V_S × V_S))
            if G_S is strongly connected then                     ▷ Tarjan's algorithm
                L ← L ∪ V_S
            end if
        end for
    end if
end for
return L
```

(SCCs) of the graph G in time $\mathcal{O}(|V| + |E|)$ using the standard Tarjan's algorithm [52]. Then, for each component S, different cases are considered. If S is trivial - that is, it consists of a single node with no self-loop - it is discarded. If S is a singleton node that has a self-loop, it is considered a legitimate loop. In the case where S contains more than one node, the algorithm examines every subset V_S of the nodes in S and checks whether the subgraph induced by V_S is strongly connected; this check is again performed using Tarjan's algorithm. If the induced subgraph is strongly connected, then V_S is added to the set of loops; otherwise, it is discarded.

Unfortunately, we could not adopt the algorithm by Johnson [53], which runs in time $\mathcal{O}((|V| + |E|)(c + 1))$, where c denotes the number of elementary circuits in the graph. The reason is that Johnson's algorithm enumerates only elementary circuits, whereas the loop formula (see again section 2.1.4) requires all circuits, including non-elementary ones. At first glance, computing all circuits might seem unnecessarily redundant. Indeed, this is the case. Gebser et al. [54] introduced the notion of *elementary loops in logic programs* and refined the Lin and Zhao theorem (theorem 2.13 in this thesis), showing that the answer set semantics can be correctly captured using a loop formula that considers only elementary loops. This refinement leads to a more efficient computation of the loop formula. Nonetheless, for the sake of simplicity, we adopted algorithm 5 in the implementation of `lasco`, which enumerates all loops. We leave the integration of the more efficient approach based on elementary loops as a direction for future work.

The `ShallowLexGrammar.x` file We conclude this section by explaining the presence of the file `ShallowLexGrammar.x` in the codebase. This component does not involve any theoretical insight or

formal result, but rather describes a technical workaround adopted in the implementation of `lasco`. We include it here for completeness and to aid future developers or maintainers of the tool.

As discussed in section 4.1.1, `lasco` employs two lexers: `LexGrammar.x` and `ShallowLexGrammar.x`. The former is the main lexer used during standard parsing. As described in section 4.1.2, any identifier containing the substring `lasco` or `LASCO` is treated as a reserved word, and its occurrence triggers a lexical error. This convention helps avoid conflicts with internally generated symbols used by the tool.

However, during the third phase of the `lasco` pipeline (see figure 4.1), the tool performs a grounding step by invoking `gringo` on a normalized task T'', producing its grounded version T'''. Internally, `lasco` prepares a modified version T''_g to force `gringo` to return a fully instantiated program. This input (and consequently the output) may contain automatically generated symbols that include the reserved substrings `lasco` or `LASCO`.

To handle this situation, the output of `gringo` is parsed not by the main lexer, but by the secondary lexer defined in `ShallowLexGrammar.x`. This lexer is nearly identical to `LexGrammar.x`, except that it relaxes the constraint on reserved words, allowing identifiers containing the forbidden substrings. This avoids spurious lexical errors during the post-processing of grounded programs.

4.2 Benchmarks

In this section, we present preliminary results on the performance of the `lasco` tool, introduced in section 4.1. Specifically, we evaluate the tool using the disjunctive encoder, as the exponential encoder is feasible only for very small instances. Both `clingo` and `dlv` are used as backend solvers. The benchmarks considered are the following: (i) a set of tasks for learning a comparison predicate over integers; (ii) a set of logic puzzle learning tasks described in [20]; and (iii) a set of tasks for learning automata.

Results are compared to those obtained by all available versions of ILASP. This not only allows for a comparison in terms of computation time, but also enables an empirical validation of the correctness and completeness of the `lasco` tool.

All benchmarks were executed on a machine equipped with an Intel Core i7-1165G7 (11th Gen) CPU @2.80 GHz and 16GB of RAM.

4.2.1 Learning comparison predicates

In this section, we consider four learning tasks aimed at inferring the relations $<$, $=$, and $>$ between pairs of integer numbers. The background knowledge encodes that, for any two integers X and Y, exactly one of the three relations $X < Y$, $X > Y$, or $X = Y$ holds. Additionally, two rules expressing transitivity for $<$ and $>$ are included. We restrict the domain to the first three positive integers.

All four tasks are derived from a common base task, denoted as T_{comp}:

```
num(1..3).
1{lt(X,Y); gt(X,Y); eq(X,Y)}1 :- num(X), num(Y).

#pos({eq(1,1)}, {}).
#pos({lt(1,2)}, {}).
```

```
#pos({gt(2,1)}, {}).
#neg({eq(1,2)}, {}).
#neg({eq(2,1)}, {}).
#neg({lt(2,2)}, {}).
#neg({lt(2,1)}, {}).
#neg({gt(2,2)}, {}).

1 ~ eq(1,1).    1 ~ eq(2,1).    1 ~ eq(3,1).
1 ~ eq(1,2).    1 ~ eq(2,2).    1 ~ eq(3,2).
1 ~ eq(1,3).    1 ~ eq(2,3).    1 ~ eq(3,3).
1 ~ lt(1,1).    1 ~ lt(2,1).    1 ~ lt(3,1).
1 ~ lt(1,2).    1 ~ lt(2,2).    1 ~ lt(3,2).
1 ~ lt(1,3).    1 ~ lt(2,3).    1 ~ lt(3,3).
1 ~ gt(1,1).    1 ~ gt(2,1).    1 ~ gt(3,1).
1 ~ gt(1,2).    1 ~ gt(2,2).    1 ~ gt(3,2).
1 ~ gt(1,3).    1 ~ gt(2,3).    1 ~ gt(3,3).
```

Let e_{sat} denote the example `#neg({gt(1,2)}, {})`, and e_{unsat} denote `#neg({gt(2,1)}, {})`. We also define two variants of transitivity rules.

Tight version (denoted as $P_{tight}^{transitivity}$):

```
lt(X,Y) :- X<Z, lt(Z,Y), num(X), num(Y), num(Z).
gt(X,Y) :- X>Z, gt(Z,Y), num(X), num(Y), num(Z).
```

Looping version (denoted as $P_{loop}^{transitivity}$):

```
lt(X,Y) :- lt(X,Z), lt(Z,Y), num(X), num(Y), num(Z).
gt(X,Y) :- gt(X,Z), gt(Z,Y), num(X), num(Y), num(Z).
```

We now have all the elements to define our four learning tasks:

$$
\begin{aligned}
T_{comp}^{sat,tight} &:= T_{comp} \cup \{e_{sat}\} \cup P_{tight}^{transitivity} \\
T_{comp}^{unsat,tight} &:= T_{comp} \cup \{e_{unsat}\} \cup P_{tigth}^{transitivity} \\
T_{comp}^{sat,loop} &:= T_{comp} \cup \{e_{sat}\} \cup P_{loop}^{transitivity} \\
T_{comp}^{unsat,loop} &:= T_{comp} \cup \{e_{unssat}\} \cup P_{loop}^{transitivity}
\end{aligned}
$$

The tasks of type $T_{comp}^{sat,\cdot}$ are satisfiable. Their optimal inductive solution is the set {`lt(1,2), eq(2,2), gt(2,1)`}, although many other non-minimal inductive solutions exist. Conversely, tasks of type $T_{comp}^{unsat,\cdot}$ are unsatisfiable. Tasks with the *tight* transitivity rules are tight (i.e., the combined background and hypothesis space $B \cup S_M$ is tight), while the *loop* versions contain exactly 794 loops. Finally, notice that if we remove all negative examples from the task then the optimal inductive solution is the empty set.

Table 4.1 reports the computation times of `lasco` - tested with both `dlv` and `clingo` backends, and across all three predefined solving modes: `first`, `opt`, and `all` - as well as those of ILASP (versions 2, 2i, 3, and 4). All times are given in seconds. The symbol `e` (for exceeded) indicates that the computation time surpassed the timeout threshold of 180 seconds. For `lasco`, we report not only the total computation time, but also a breakdown of the time spent in the grounding, encoding and solving modules. The times for the de-contextualization, normalization, and extraction phases of the `lasco` pipeline (see figure 4.1) are omitted, as they are negligible in practice. Moreover, table 4.2 reports the first solutions found by `lasco` on the satisfiable tasks.

From those tables, several interesting observations can be made:

- Encoding and grounding times are consistently a small fraction of the total solving time. This aligns with theoretical expectations, as the encoding procedure is linear in the size of the learning task.
- Encoding times for *non-tight* tasks are approximately one order of magnitude higher than those for *tight* tasks. This is again expected, since computing loop formulas can be computationally expensive.
- For satisfiable tasks, finding an inductive solution using the `first` configuration is consistently faster than computing an optimal inductive solution. For instance, in table 4.1 (c), using `clingo` to find the optimum is 2.5 times slower than finding the first solution.
- On *tight* tasks, `lasco` with `clingo` outperforms ILASP in both the `first` and `opt` solving configurations.
- Configurations using `clingo` are generally faster than those using `dlv`.
- Finding a non-minimal inductive solution (i.e., one that is not optimal) may lead to incorrect generalizations. For example, as shown in table 4.2, the first inductive solution returned by the `clingo` configuration on $T_{comp}^{sat,tight}$ includes the atom `eq(1,3)`, which incorrectly asserts equality between 1 and 3. This behavior is likely due to the small number of training examples.
- Although `dlv` configurations are generally slower than `clingo`, the first inductive solutions returned by `dlv` tend to be shorter and of higher quality.
- Compared to ILASP, `lasco` shows superior performance in detecting unsatisfiability, even in the presence of *non-tight* tasks. Specifically, the best time achieved by ILASP on $T_{comp}^{unsat,tight}$ is over 20 times slower than the best `lasco` configuration. Similarly, for $T_{comp}^{unsat,loop}$, the best ILASP time is over 6 times slower.
- Among all ILASP versions, ILASP4 is architecturally the closest to `lasco`, as it introduces conflict analysis. Similarly, `lasco` makes intensive use of conflict analysis by fully delegating the hypothesis search to the underlying ASP solver (e.g., `clingo`), which performs conflict-driven learning and propagation natively. Despite this similarity, in all tested scenarios, the `clingo opt` configuration of `lasco` consistently outperforms ILASP4 in terms of solving time. This empirical evidence supports the idea that generating a single disjunctive encoding upfront and delegating optimization

	grounding	encoding		solving	total
lasco	0.01	0.03	clingo first	0.12	**0.19**
			clingo opt	0.15	0.20
			clingo all	e	e
			dlv first	0.39	0.44
			dlv opt	e	e
			dlv all	e	e
ILASP2	/	/		/	0.43
ILASP2i	/	/		/	0.63
ILASP3	/	/		/	46.17
ILASP4	/	/		/	2.38

(a) $T_{comp}^{sat,tight}$

	grounding	encoding		solving	total
lasco	0.00	0.04	clingo first	0.17	**0.22**
			clingo opt	0.25	0.31
			clingo all	0.21	0.28
			dlv first	0.54	0.61
			dlv opt	0.44	0.49
			dlv all	0.48	0.58
ILASP2	/	/		/	4.72
ILASP2i	/	/		/	5.06
ILASP3	/	/		/	45.11
ILASP4	/	/		/	7.67

(b) $T_{comp}^{unsat,tight}$

	grounding	encoding		solving	total
lasco	0.01	0.37	clingo first	1.60	1.99
			clingo opt	4.49	4.97
			clingo all	e	e
			dlv first	42.30	42.62
			dlv opt	e	e
			dlv all	e	e
ILASP2	/	/		/	**1.29**
ILASP2i	/	/		/	1.98
ILASP3	/	/		/	e
ILASP4	/	/		/	38.27

(c) $T_{comp}^{sat,loop}$

	grounding	encoding		solving	total
lasco	0.01	0.35	clingo first	19.36	19.73
			clingo opt	13.54	**13.96**
			clingo all	16.98	17.34
			dlv first	46.43	46.77
			dlv opt	57.88	58.26
			dlv all	51.35	51.74
ILASP2	/	/		/	85.63
ILASP2i	/	/		/	97.49
ILASP3	/	/		/	e
ILASP4	/	/		/	e

(d) $T_{comp}^{unsat,loop}$

Table 4.1: Computation times (in seconds) for the four tasks $T_{comp}^{\cdot\cdot}$ using `lasco` (with all solver configurations) and ILASP (versions 2, 2i, 3, and 4).

task	configuration	first inductive solution
$T_{comp}^{sat,tight}$	`clingo first`	`eq(1,1), eq(2,2), eq(2,3), gt(2,1), eq(3,2), eq(3,3), eq(1,3), lt(1,2)`
$T_{comp}^{sat,tight}$	`dlv first`	`gt(3,3), gt(2,1), eq(2,2), lt(1,3), lt(1,2), eq(1,1)`
$T_{comp}^{sat,loop}$	`clingo first`	`eq(2,2), lt(2,3), gt(2,1), gt(3,2), gt(3,3), lt(1,2), lt(1,3)`
$T_{comp}^{sat,loop}$	`dlv first`	`gt(3,3), gt(3,2), gt(2,1), eq(2,2), lt(1,3), lt(1,2)`

Table 4.2: First inductive solutions found by `lasco` (under all solver configurations) for the tasks $T_{comp}^{sat,tight}$ and $T_{comp}^{sat,loop}$.

entirely to the ASP solver is not only viable, but a promising approach to solving ILP_{LAS} tasks efficiently.

4.2.2 Learning logic puzzles

In this section, we evaluate the performance of `lasco` on the set of logic puzzle tasks introduced by Dreossi [20]. The selected benchmarks include: (i) *nQueens*, (ii) *FlowLines*, (iii) *StarBattle*, and (iv) *15Puzzle*.

Table 4.3 reports the computation times of `lasco` and ILASP on the *nQueens* task. This task features a hypothesis space consisting of six candidate rules, along with three positive and three negative examples. The optimal inductive solution for this task is:

```
:- same_col(X1,Y1,X2,Y2).
:- same_diagonal(X1,Y1,X2,Y2).
1{queen(X,Y):coord(Y)}1 :- coord(X).
```

The results show that all ILASP versions outperform `lasco` by approximately an order of magnitude. Nevertheless, `lasco`'s computation times remain competitive: under one second for all `clingo`-based configurations, and under ten seconds for all `dlv`-based configurations. Moreover, the hypothesis learned by `lasco` in the `clingo first` and `dlv first` configurations is:

```
:- same_row(X1,Y1,X2,Y2).
:- same_col(X1,Y1,X2,Y2).
:- same_diagonal(X1,Y1,X2,Y2).
1{queen(X,Y):coord(Y)}1 :- coord(X).
```

This solution, although not optimal due to the redundant inclusion of the `same_row` constraint, still consists of valid rules w.r.t. the *nQueens* puzzle. In conclusion, although ILASP exhibits superior raw performance, `lasco` provides feasible computation times and remains a practical alternative for solving this class of tasks.

We were unable to apply `lasco` effectively to the remaining tasks due to scalability issues. The *FlowLines* task, which features a hypothesis space of 12 rules, causes `lasco` to run out of memory - even when restricting the hypothesis space to only the rules that appear in the inductive solution. A similar situation occurs with the *StarBattle* task, which has over 500 rules in the hypothesis space. In this case, `lasco` also runs out of memory under the full hypothesis set. However, when restricting the hypothesis

space to just the two rules forming the correct optimal inductive solution, `lasco` successfully completes the computation in under 4 seconds. Despite this, ILASP solves the task with the full hypothesis space in less than one second, highlighting a significant performance gap. Finally, for the *15Puzzle* task, `lasco` takes over an hour to complete the encoding phase, producing an extremely large exponential-size program. These negative results stem from the fact that all three tasks are non-tight. The presence of numerous loops leads to the generation of an exponentially large program - both costly to construct and even more impractical to submit to an ASP solver.

	grounding	encoding		solving	**total**
`lasco`	0.01	0.20	`clingo first`	0.41	0.67
			`clingo opt`	0.44	0.68
			`clingo all`	0.50	0.75
			`dlv first`	5.87	6.08
			`dlv opt`	6.29	6.56
			`dlv all`	7.24	7.48
ILASP2	/	/		/	**0.07**
ILASP2i	/	/		/	0.08
ILASP3	/	/		/	**0.07**
ILASP4	/	/		/	**0.07**

Table 4.3: Computation times (in seconds) for the *nQueens* task using `lasco` (with all solver configurations) and ILASP (versions 2, 2i, 3, and 4).

4.2.3 Learning automata

In [19], Ielo et al. presented a learning task for passively inferring Linear Temporal Logic (LTL) formulas. Inspired by this, we introduce a new learning task, denoted as $T^n_{automata}$, aimed at passively learning finite automata. The task is defined as follows:

```
%% alphabet and initial state
states(0..n).
char(a). char(b).
initial(0).

%% run definition
run(0,S) :- initial(S), state(S).
run(T,S1) :- word(T,C), run(T-1,S), delta(S,C,S1),
             state(S), state(S1), time(T), char(C).

%% run is a function
:- run(T,S), run(T,S2), S!=S2, state(S), state(S2), time(T).

%% delta is a function
:- delta(S,C,S1), delta(S,C,S2), S1!=S2, char(C), state(S), state(S1), state(S2).

%% reachability predicate
```

```
reachable(0).
reachable(S1) :- reachable(S), delta(S,C,S1), state(S), state(S1), char(C).

%% the automaton is complete on reachable states
complete(S,C) :- reachable(S), char(C), delta(S,C,S1), state(S1).
:- not complete(S,C), state(S), reachable(S), char(C).

%% accepting conditions
accepted :- run(L,S), state(S), final(S), length(L).
rejected :- not accepted.

%% search pruning
:- state(S), not state(S1), states(S), states(S1), S1 < S.
```

Here, `states(0..n)` indicates that the goal is to learn an automaton with at most $n+1$ states, while the facts `char(a).` `char(b).` define the input alphabet as $\Sigma = \{a, b\}$. Given a value of n, the hypothesis space of $T^n_{automata}$ is defined as follows:

$$1 \sim \texttt{delta(i,c,j).} \qquad \forall i = 0, \ldots, n-1 \; \forall j = 0, \ldots, n-1 \forall c \in \Sigma$$
$$1 \sim \texttt{state(i).} \qquad \forall i = 0, \ldots, n-1$$
$$1 \sim \texttt{final(i).} \qquad \forall i = 0, \ldots, n-1$$

For $n = 9$, the hypothesis space contains 220 candidate rules.

We can now define a set of examples that allows for the effective learning of a finite automaton. For instance, the following example set $E_{(ab)^*}$ is sufficient to induce the minimal automaton for the regular language $(ab)^*$:

```
#pos({},{},{:- accepted. length(1). time(1). word(1,a).}).  % "a"
#pos({},{},{:- accepted. length(1). time(1). word(1,b).}).  % "b"
#pos({},{},{:- accepted. length(2). time(1..2). word(1,b). word(2,a).}).  % "ba"
#pos({},{},{:- accepted. length(2). time(1..2). word(1,a). word(2,a).}).  % "aa"
#pos({},{},{:- accepted. length(2). time(1..2). word(1,b). word(2,b).}).  % "bb"
#pos({},{},{:- accepted. length(3). time(1..3). word(1,a). word(2,a). word(3,b).}).  % "aab"
#pos({},{},{:- accepted. length(3). time(1..3). word(1,a). word(2,b). word(3,a).}).  % "aba"
#pos({},{},{:- accepted. length(3). time(1..3). word(1,a). word(2,b). word(3,b).}).  % "abb"
#pos({},{},{:- accepted. length(3). time(1..3). word(1,b). word(2,a). word(3,b).}).  % "bab"
#pos({},{},{:- accepted. length(4). time(1..4). word(1,a).
            word(2,b). word(3,a). word(4,a).}).  % "abaa"
#pos({},{},{:- accepted. length(4). time(1..4).
            word(1,a). word(2,a). word(3,b). word(4,b).}).  % "aabb"
#pos({},{},{:- accepted. length(4). time(1..4). word(1,b).
            word(2,a). word(3,a). word(4,b).}).  % "baab"
#pos({},{},{:- accepted. length(4). time(1..4).
            word(1,a). word(2,b). word(3,b). word(4,a).}).  % "abbab"
#pos({},{},{:- accepted. length(5). time(1..5).
```

```
            word(1,a). word(2,b). word(3,a). word(4,b). word(5,b).}).  % "ababb"
#pos({},{},{:- rejected. length(2). time(1..2). word(1,a). word(2,b).}).  % "ab"
#pos({},{},{:- rejected. length(4). time(1..4).
            word(1,a). word(2,b). word(3,a). word(4,b).}).  % "abab"
#pos({},{},{:- rejected. length(6). time(1..6).
            word(1,a). word(2,b). word(3,a). word(4,b). word(5,a). word(6,b).}).  % "ababab"
```

Positive examples, such as those included in the set $E_{(ab)^*}$, are sufficient to specify that certain words must be accepted or rejected by the learned automaton. However, more complex conditions can be enforced through negative examples. For instance, we can express that *every* word of length seven with the symbol `a` in the fifth position must be rejected.

We define the set E_{neg} as follows:

```
#pos({},{},{:- rejected. length(2). time(1..2). word(1,a). word(2,a). }).
#pos({},{},{:- accepted. length(2). time(1..2). word(1,a). word(2,b). }).
#neg({},{},{:- accepted.
            length(7).
            time(1..7).
            word(4,a).
            1{word(1,a); word(1,b)}1.
            1{word(2,a); word(2,b)}1.
            1{word(3,a); word(3,b)}1.
            1{word(4,a); word(4,b)}1.
            1{word(5,a); word(5,b)}1.
            1{word(6,a); word(6,b)}1.
            1{word(7,a); word(7,b)}1. }).
#neg({},{},{:- accepted.
            length(7).
            time(1..7).
            word(5,a).
            1{word(1,a); word(1,b)}1.
            1{word(2,a); word(2,b)}1.
            1{word(3,a); word(3,b)}1.
            1{word(4,a); word(4,b)}1.
            1{word(5,a); word(5,b)}1.
            1{word(6,a); word(6,b)}1.
            1{word(7,a); word(7,b)}1. }).
```

Based on the definitions above, we introduce two learning tasks:

- $T_{(ab)^*} := T^9_{automata} \cup E_{(ab)^*}$ - a task aimed at learning the minimal automaton for the regular language $(ab)^*$ from positive examples only. An optimal inductive solution of this task is `delta(0,b,2), delta(2,b,2), final(0), state(0), state(1), state(2), delta(0,a,1), delta(1,a,2), delta(2,a,2), delta(1,b,0)`.

- $T_{pattern} := T^3_{automata} \cup E_{neg}$ - a task designed to enforce structural constraints via negative examples, e.g., accepting all words of length seven with an `a` in the fifth position. An optimal

inductive solution of this task is `final(1), state(0), state(1), state(2), delta(1,b,1), delta(1,a,1), delta(2,a,1), delta(0,a,2), delta(0,b,2), delta(2,b,2)`.

	grounding	encoding		solving	**total**
`lasco`	0.15	0.13	`clingo first`	1.00	1.26
			`clingo opt`	1.10	1.43
			`dlv first`	29.75	30.12
			`dlv opt`	33.54	33.79
ILASP2	/	/		/	**0.15**
ILASP2i	/	/		/	0.55
ILASP3	/	/		/	e
ILASP4	/	/		/	e

(a) $T_{(ab)^*}$

	grounding	encoding		solving	**total**
`lasco`	0.04	0.20	`clingo first`	0.63	**0.88**
			`clingo opt`	27.87	28.17
			`dlv first`	e	e
			`dlv opt`	83.19	83.44
ILASP2	/	/		/	e
ILASP2i	/	/		/	e
ILASP3	/	/		/	e
ILASP4	/	/		/	e

(b) $T_{pattern}$

Table 4.4: Computation times (in seconds) for the two tasks $T_{(ab)^*}$ and $T_{pattern}$ using `lasco` (with all solver configurations) and ILASP (versions 2, 2i, 3, and 4).

Table 4.1 reports the computation times of `lasco` and ILASP on the two tasks defined above. We omit the `lasco all` configurations, as the number of isomorphic inductive solutions is prohibitively large and not informative for performance comparison. Table 4.5 further presents the first solutions returned by `lasco`. These results, in line with those discussed in section 4.2.1, confirm that `lasco` achieves strong performance. In particular, for the task $T_{(ab)^*}$ - which consists solely of positive examples - only ILASP2 and ILASP2i outperform `lasco`. For the task $T_{pattern}$, which includes negative examples, `lasco` outperforms all versions of ILASP.

These findings suggest that `lasco` is especially well-suited for tasks characterized by: (i) the presence of negative examples, (ii) tight programs, and (iii) small grounding size.

task	configuration	first inductive solution
$T_{(ab)^*}$	`clingo first`	`delta(0,b,1) delta(2,b,0) delta(0,a,2) delta(2,a,1) delta(1,a,1) delta(1,b,1) state (2) state (1) state (0) final (0)`
$T_{(ab)^*}$	`dlv first`	`delta(0,a,8), delta(7,a,7), delta(3,a,7), delta(2,a,7), delta(7,b,0), delta(6,b,0), delta(5,b,0), delta(1,b,0), delta(6,a,9), delta(0,a,2), delta(9,a,4), delta(8,a,4), delta(5,a,4), delta(1,a,4), delta(6,a,3), delta(4,a,3), delta(2,b,9), delta(0,b,7), delta(9,b,6), state (7), state (6), state (5), state (4), state (3), state (2), state (1), state (0), final (6), final (5), final (1), delta(8,b,9), delta(7,b,9), delta(6,b,9), delta(8,b,2), delta(3,b,1), delta(4,b,6), delta(2,b,5)`
$T_{pattern}$	`clingo first`	`delta(0,a,3) delta(2,a,3) delta(3,a,2) delta(3,b,0) final (2) delta(0,b,3) delta(1,b,3) delta(2,b,3) final (3) state (0) state (1) state (2) state (3) delta(1,a,0)`

Table 4.5: First inductive solutions found by `lasco` (under all solver configurations) for the tasks$T_{(ab)^*}$ and $T_{pattern}$.

5

Conclusions

Summary of the results In this thesis, we investigated the problem of Inductive Logic Programming within the Learning from Answer Sets (LAS) framework, focusing on the computation of inductive solutions. This problem is known to be Σ_2^P-hard, placing it at the second level of the polynomial hierarchy.

Previous approaches, such as ILASP, address this challenge by relying on multiple encodings and several iterative calls to an ASP solver, applying domain-specific optimizations at each step. We refer to this as a *multi-shot* approach. In contrast, our work proposes a fundamentally different *single-shot* strategy: the entire LAS task is encoded as a single ASP program, and its inductive solutions (or optimal ones) are directly extracted from the program's answer sets.

The motivation behind this design choice lies in the idea that, by exposing the full problem structure to the ASP solver, we can leverage decades of advancements in ASP solving - such as conflict-driven learning and optimization - without hardcoding such mechanisms externally. This raises the possibility of outperforming ILASP, the current state-of-the-art LAS solver.

We first considered ground LAS tasks and introduced two encodings. The first is based solely on normal rules and has exponential size; the second is disjunctive and has linear size under the assumption that the union of background knowledge and hypothesis space is tight. The disjunctive encoding is derived by reducing the LAS task to a QBF formula of the form $\exists X \forall Y\ \varphi(X, Y)$, and then translating it into an $\mathsf{ASP}^{\mathsf{D}}$ program via the well-known saturation technique by Eiter and Gottlob.

For both encodings, we formally proved correctness and completeness: the answer sets of the resulting ASP program correspond one-to-one with the inductive solutions of the original LAS task. We also provided structural complexity bounds on the size of the generated encoding.

We then extended our attention to non-ground tasks. To this end, we introduced a formal notion of grounding for ILP_{LAS} tasks. The grounding process yields a corresponding ILP_{LAS}^{agg} task, where hypothesis elements are aggregations of rules rather than individual rules. A key theoretical result is that any ILP_{LAS} task is equivalent to its grounded version - an insight that is not only foundational for our encodings, but also of independent interest for the LAS theoretical landscape.

Building on these foundations, we implemented a new LAS solver named `lasco`, which is publicly available on GitHub. The tool follows a modular pipeline consisting of six stages: (i) de-contextualization, (ii) normalization, (iii) grounding, (iv) encoding, (v) solving, and (vi) extraction. The encoding phase

is based on our correct and complete disjunctive encoding. The tool supports configurable solving backends, encoding strategies, and search modalities, while maintaining compatibility with the ILASP input format.

We evaluated `lasco` on a variety of relevant benchmarks. Despite ILASP being a mature tool backed by over a decade of development, our prototype `lasco` demonstrated competitive performance. Notably, there are instances where ILASP is over 20 times slower than `lasco`. These results point to the effectiveness of the single-shot disjunctive encoding approach, and open up promising directions for future improvements, with the potential to establish `lasco` as a new state-of-the-art solver in the ILP_{LAS} landscape.

Open problems Among the open directions, one key theoretical question remains unresolved:

Can we design a polynomial-size disjunctive encoding for every ground ILP_{LAS} task?

We conjecture that this is not the case:

Conjecture 5.1 *Assume that $\mathsf{P} \not\subseteq \mathsf{NC}^1/poly$, that is, not all languages in P can be decided by families of propositional formulas of polynomial size. Then, there does not exist a polynomial-size ASP^D encoding $P(T)$ that, for every ground ILP_{LAS} task T, precisely captures the set of its inductive solutions.*

Following our intuition, we think that if we refuse this conjecture, i.e. we found a polynomial size ASP^D encoding for every ground task T, we would be able to capture the full answer set semantics by means of a propositional formula of polynomial size for every ASP^N program. But this last statement is false and so assuming the contrary of our conjecture would lead to a contradiction.

Future directions Several directions remain open for future research and development.

Regarding encodings and the `lasco` tool:

- Generalize the current approach to support tasks within the ILP^{noise}_{LAS} framework [12], thereby allowing for learning from noisy or inconsistent data.
- Extend the experimental evaluation to include additional benchmarks, and perform a comparative analysis between `lasco` and FastLAS [13].
- Incorporate preprocessing stages into the `lasco` pipeline to optimize the input task before encoding, potentially improving performance and scalability.
- Extend the encoding language to natively support *mode declarations*, eliminating the need for external preprocessing tools.
- Investigate alternative encoding strategies based on other formalisms, such as Integer Linear Programming (ILP), Quadratic Programming (QP), or Constraint Programming (CP), to assess their potential for LAS solving.

Regarding LAS-solving theory:

- Explore parallelization techniques for LAS solving, especially in the context of ILASP-style algorithms. One long-term goal could be a GPU-based implementation (e.g., via CUDA), which may significantly accelerate inductive learning. This requires a careful analysis of where parallelism arises in LAS: for example, across examples or across candidate rules in the hypothesis space.

- Develop an *approximation theory* for ILP_{LAS}. Similar to classic NP-hard problems, one could aim to design approximation algorithms that compute inductive solutions in linear time with guaranteed bounds on optimality. This would be especially relevant given the optimization nature of LAS tasks.

Regarding LAS and ILP more broadly:

- Investigate alternative frameworks such as *Probabilistic Learning from Answer Sets* (see section 2.2), and compare their expressiveness and complexity to the LAS framework.

- Identify real-world domains where LAS-based learning is particularly suitable or advantageous, possibly involving structured symbolic knowledge or constraints.

- Explore hybrid approaches that integrate ILP techniques into sub-symbolic systems (e.g., neural networks), potentially enabling neuro-symbolic learning within the LAS paradigm.

Final comments In recent years, sub-symbolic approaches have become the dominant paradigm in many real-world applications. Symbolic systems are often overlooked due to their inherent complexity and less immediate scalability. However, modern SAT and ASP solvers have reached a high level of maturity and performance, making them powerful tools for tackling NP-hard problems.

Although symbolic AI is currently less prominent than its sub-symbolic counterpart, recent developments - such as the introduction of the LAS framework and the ILASP system - mark significant progress toward the integration of symbolic methods into practical domains. A key advantage of symbolic systems lies in their inherent explainability, which remains a major limitation in most sub-symbolic models.

This thesis contributes to this line of work by proposing an alternative approach to the current state-of-the-art LAS solver, demonstrating superior performance on several benchmarks. While modest in scope, this contribution represents another step forward in making symbolic AI more practical, efficient, and accessible.

Bibliography

[1] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski, Bowen, and Kenneth, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080. MIT Press, 1988.

[2] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer set solving in practice*. Springer Nature, 2022.

[3] Martin Gebser, Roland Kaminski, Murat Knecht, and Torsten Schaub. plasp: A prototype for pddl-based planning in asp. In *Logic Programming and Nonmonotonic Reasoning: 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings 11*, pages 358–363. Springer, 2011.

[4] Son Cao Tran, Enrico Pontelli, Marcello Balduccini, and Torsten Schaub. Answer set planning: a survey. *Theory and Practice of Logic Programming*, 23(1):226–298, 2023.

[5] Paola Cappanera, Simone Caruso, Carmine Dodaro, Giuseppe Galatà, Marco Gavanelli, Marco Maratea, Cinzia Marte, Marco Mochi, Maddalena Nonato, Marco Roma, et al. Recent answer set programming applications to scheduling problems in digital health. In *CEUR WORKSHOP PROCEEDINGS*, volume 3883, pages 129–141. CEUR-WS, 2024.

[6] Andreas Falkner, Gerhard Friedrich, Konstantin Schekotihin, Richard Taupe, and Erich C Teppan. Industrial applications of answer set programming. *KI-Künstliche Intelligenz*, 32(2):165–176, 2018.

[7] Finale Doshi-Velez and Been Kim. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*, 2017.

[8] Ly Ly Trieu, Tran Cao Son, and Marcello Balduccini. xasp: An explanation generation system for answer set programming. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 363–369. Springer, 2022.

[9] Mario Alviano, Ly Ly T Trieu, Tran Cao Son, Marcello Balduccini, et al. Advancements in xasp, an xai system for answer set programming. In *CILC*, 2023.

[10] Mario Alviano, Ly Ly Trieu, Tran Cao Son, and Marcello Balduccini. The xai system for answer set programming xasp2. *Journal of Logic and Computation*, 34(8):1500–1525, 2024.

[11] Stephen Muggleton. Inductive logic programming. *New generation computing*, 8:295–318, 1991.

[12] Mark Law. *Inductive learning of answer set programs*. PhD thesis, Imperial College London, UK, 2018.

[13] Mark Law, Alessandra Russo, and Krysia Broda. The ilasp system for inductive learning of answer set programs. *arXiv preprint arXiv:2005.00904*, 2020.

[14] Mark Law. Conflict-driven inductive logic programming. *Theory and Practice of Logic Programming*, 23(2):387–414, 2023.

[15] Cynthia Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature machine intelligence*, 1(5):206–215, 2019.

[16] Hleg Ai. High-level expert group on artificial intelligence. *Ethics guidelines for trustworthy AI*, 6, 2019.

[17] Fredrik Heintz, Michela Milano, and Barry O'Sullivan. *Trustworthy AI-integrating learning, optimization and reasoning*. Springer, 2021.

[18] Daniele Fossemò, Filippo Mignosi, Luca Raggioli, Matteo Spezialetti, Fabio Aurelio D'Asaro, et al. Using inductive logic programming to globally approximate neural networks for preference learning: challenges and preliminary results. In *CEUR WORKSHOP PROCEEDINGS*, pages 67–83, 2022.

[19] Antonio Ielo, Mark Law, Valeria Fionda, Francesco Ricca, Giuseppe De Giacomo, and Alessandra Russo. Towards ilp-based ltl f passive learning. In *International Conference on Inductive Logic Programming*, pages 30–45. Springer, 2023.

[20] Talissa Dreossi et al. Exploring ilasp through logic puzzles modelling. In *CEUR Workshop Proceedings*, volume 3428. CEUR-WS, 2023.

[21] Francesco Chiariello, Antonio Ielo, and Alice Tarzariol. An ilasp-based approach to repair petri nets. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 85–97. Springer, 2024.

[22] Agostino Dovier, Talissa Dreossi, Andrea Formisano, et al. Xai-law towards a logic programming tool for taking and explaining legal decisions. In *CEUR WORKSHOP PROCEEDINGS*, volume 3733. CEUR-WS, 2024.

[23] Talissa Dreossi, Agostino Dovier, Andrea Formisano, Mark Law, Agostino Manzato, Alessandra Russo, and Matthew Tait. Towards explainable weather forecasting through fastlas. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 262–275. Springer, 2024.

[24] Mark Law, Alessandra Russo, Elisa Bertino, Krysia Broda, and Jorge Lobo. Fastlas: Scalable inductive logic programming incorporating domain-specific optimisation criteria. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 2877–2885, 2020.

[25] Thomas Eiter and Georg Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15:289–323, 1995.

[26] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, July 2006.

[27] Martin Gebser, Roland Kaminski, and Torsten Schaub. Complex optimization in answer set programming, 2011.

[28] Francesco Ricca. Compute paracoherent answer sets via saturation. In Marco Maratea and Mauro Vallati, editors, *Proceedings of the RiCeRcA Workshop co-located with the 17th International Conference of the Italian Association for Artificial Intelligence, RiCeRcA@Ai*iA 2018, Trento, Italy, November 22, 2018*, volume 2272 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018.

[29] Fangzhen Lin and Yuting Zhao. Assat: Computing answer sets of a logic program by sat solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[30] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The potsdam answer set solving collection. *Ai Communications*, 24(2):107–124, 2011.

[31] Keith L Clark. Negation as failure. In *Logic and data bases*, pages 293–322. Springer, 1977.

[32] Victor W Marek and Miroslaw Truszczyński. Stable models and an alternative logic programming paradigm. In *The logic programming paradigm: A 25-year perspective*, pages 375–398. Springer, 1999.

[33] Maarten H Van Emden and Robert A Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM (JACM)*, 23(4):733–742, 1976.

[34] Mario Alviano, Carmine Dodaro, Matti Järvisalo, Marco Maratea, and Alessandro Previti. Cautious reasoning in asp via minimal models and unsatisfiable cores. *Theory and Practice of Logic Programming*, 18(3-4):319–336, 2018.

[35] Giovanni Amendola, Carmine Dodaro, and Marco Maratea. Abstract solvers for computing cautious consequences of asp programs. *Theory and Practice of Logic Programming*, 19(5-6):740–756, 2019.

[36] Giovanni Amendola, Carmine Dodaro, and Marco Maratea. A formal approach for cautious reasoning in answer set programming. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 4716–4720, 2021.

[37] Fran cois Fages. Consistency of clark's completion and existence of stable models. *Journal of Methods of logic in computer science*, 1(1):51–60, 1994.

[38] Yuliya Babovich, Esra Erdem, and Vladimir Lifschitz. Fages' theorem and answer set programming. *arXiv preprint cs/0003042*, 2000.

[39] Vladimir Lifschitz and Alexander Razborov. Why are there so many loop formulas? *ACM Transactions on Computational Logic (TOCL)*, 7(2):261–268, 2006.

[40] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187:52–89, 2012.

[41] Stephen Muggleton. Inverse entailment and progol. *New generation computing*, 13:245–286, 1995.

[42] Luc De Raedt and Wim Van Laer. Inductive constraint logic. In *International Workshop on Algorithmic Learning Theory*, pages 80–94. Springer, 1995.

[43] Luc De Raedt and Luc Dehaspe. Learning from satisfiability. In *Ninth Dutch Conference on Artificial Intelligence (NAIC'97)*, pages 303–312, 1997.

[44] Chiaki Sakama and Katsumi Inoue. Brave induction: a logical framework for learning from incomplete information. *Machine Learning*, 76:3–35, 2009.

[45] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 1020 states and beyond. *Information and computation*, 98(2):142–170, 1992.

[46] George S Boolos, John P Burgess, and Richard C Jeffrey. *Computability and logic*. Cambridge university press, 2002.

[47] Benjamin Kaufmann, Nicola Leone, Simona Perri, and Torsten Schaub. Grounding and solving in answer set programming. *AI magazine*, 37(3):25–32, 2016.

[48] Simon Marlow et al. Haskell 2010 language report. 2010.

[49] Markus Forsberg and Aarne Ranta. Bnf converter. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 94–95, 2004.

[50] Simon Marlow and Andy Gill. Happy user guide. *Glasgow University, December*, 12, 1997.

[51] Chris Dornan and Isaac Jones. Alex user guide, 2003.

[52] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[53] Donald B Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[54] Martin Gebser, Joohyung Lee, and Yuliya Lierler. On elementary loops of logic programs. *Theory and Practice of Logic Programming*, 11(6):953–988, 2011.