Representing and Learning Grammars in Answer Set Programming

Mark Law
Imperial College London, UK
mark.law09@imperial.ac.uk

Alessandra Russo
Imperial College London, UK
a.russo@imperial.ac.uk

Elisa Bertino
Purdue University, USA
bertino@purdue.edu

Krysia Broda
Imperial College London, UK
k.broda@imperial.ac.uk

Jorge Lobo
ICREA - Universitat Pompeo Fabra
jorge.lobo@upf.edu

Abstract

In this paper we introduce an extension of context-free grammars called *answer set grammars* (ASGs). These grammars allow annotations on production rules, written in the language of Answer Set Programming (ASP), which can express context-sensitive constraints. We investigate the complexity of various classes of ASG with respect to two decision problems: deciding whether a given string belongs to the language of an ASG and deciding whether the language of an ASG is non-empty. Specifically, we show that the complexity of these decision problems can be lowered by restricting the subset of the ASP language used in the annotations. To aid the applicability of these grammars to computational problems that require context-sensitive parsers for partially known languages, we propose a learning task for inducing the annotations of an ASG. We characterise the complexity of this task and present an algorithm for solving it. An evaluation of a (prototype) implementation is also discussed.

Introduction

All computational problems can be characterised as the task of recognising a language. Many of these languages can be captured by grammars. For example, finite state automata can be characterised by regular grammars, pushdown automata by context-free grammars (CFGs), linear bounded automata by context-sensitive grammars and Turing machines by unrestricted grammars. Grammars are useful in many situations where the problem to solve is that of recognising the sentences of a language. For instance, grammars are useful to automatically generate parsers of programming languages. Typically, the problem of parser generation is, roughly speaking, done in two steps: firstly a CFG is defined and secondly either the grammar, or the parser generated by the grammar, is annotated or modified to capture the language; e.g. a CFG is used, together with annotations, as input to YACC (Johnson 1975) in order to generate parsers.

Parsers are used in many situations. However, in practice there are situations where a parser's grammar may be unknown or a parser's implementation does not match its specification. For instance, in the context of automatic test generation for program debugging, known as *fuzzing* (Sutton, Greene, and Amini 2007), hand written grammars may contain errors, meaning automatically generated test instances may fail in the programs parser (Godefroid, Peleg, and Singh 2017). Another example arises in the domain of signature-based intrusion detection systems. Signatures of attacks are specified as regular expressions. Writing these signatures is difficult, and the definitions are usually incomplete: there are attacks that are not detected by the signature and strings that are classified as attacks when they are not (e.g., (Alnabulsi, Islam, and Mamun 2014)). This is also the case in automatic classification of logs, where signatures are used and specified as regular expressions to parse the logs into classes, and, as in the case of intrusion detection, often there are incorrect classifications, e.g. (Tang, Tao, and Chang-Shing 2011).

The contribution of this paper is twofold. Firstly, a class of grammars, called *Answer Set Grammars* (ASG) is defined. These grammars are CFGs with annotations, which go beyond CFGs and are able to express some context-sensitive languages, including some which are not polynomially decidable. Secondly, a framework that given an ASG and two sets of strings, E^+ and E^-, learns a target ASG that has the same context-free component of the input grammar, and every string in E^+ (resp. E^-) is accepted (resp. rejected) by the target grammar. This framework is intended as a first step towards addressing problems, such as the three described above, by automatically modifying the grammars.

ASG annotations are expressed as Answer Set Programs (ASP). They are inspired by extensions of CFGs, such as attribute grammars (Knuth 1968) and definite clause grammars (Pereira and Warren 1980), which are capable of expressing context-sensitive conditions. The former extend CFGs with attributes and production rules with assignments to these attributes and constraints on the values that these attributes can take. The latter extend CFGs with variables that can be passed up and down the parse tree. ASGs differ from both attribute grammars and DCGs in that the annotations are purely declarative and are not subject to procedural constraints. ASGs have a high degree of expressiveness and by restricting the fragment of ASP used by the ASG, languages of different complexities can be characterised.

The insight of using ASP annotations allows recent advances in learning ASP (Law, Russo, and Broda 2015a) to be used to learn the annotations of ASGs. Our proposed learning framework is able to learn the ASP part of an

ASG that preserves the context-free component of the input grammar. This can be interpreted in many cases as learning semantic constraints of a language when its syntax is already known. This is indeed the case of automatic modification of grammars, common to the three problems described above. Our framework differs from existing approaches of grammar induction (Angluin 1987; Javed et al. 2004; Fredouille et al. 2007), where the task is instead to learn the entire grammar of a language from a set of positive and negative examples of strings in that language. As shown by our evaluation results, learning only the annotation part of an ASG makes the computational task easier than learning the whole grammar. In what follows we present: (1) the formalisation of our notion of ASGs; (2) results on the computational complexity of deciding whether a given string is a member of the language of a given ASG and deciding whether the language of a given ASG is non-empty; (3) the formalisation of the task of learning annotations of ASGs; and (4) a characterisation of the complexity of this learning task, and an algorithm that solves the learning task.
We begin by recalling the relevant notation in the next section, and then present each contribution in turn. We conclude the paper with discussions of related and future work.

Notation and Terminology

In this section we introduce basic notions and terminologies used throughout the paper. Given any atoms $\mathtt{h}$, $\mathtt{b_1, \ldots, b_n}$, $\mathtt{c_1, \ldots, c_m}$, a *normal rule* is $\mathtt{h}$:-$\mathtt{b_1, \ldots, b_n}$, not $\mathtt{c_1}$, ..., not $\mathtt{c_m}$, where $\mathtt{h}$ is the *head* and $\mathtt{b_1, \ldots, b_n}$, not $\mathtt{c_1}$, ..., not $\mathtt{c_m}$ (collectively) is the *body* of the rule, and "not" represents negation as failure. Rules of the form :-$\mathtt{b_1, \ldots, b_n}$, not $\mathtt{c_1}$, ..., not $\mathtt{c_m}$ are called *constraints*. A variable in a rule is said to be *safe* if it occurs in at least one positive literal (i.e. the $\mathtt{b_i}$'s in the above rule) in the body of the rule. In this paper, we assume an Answer Set program to be a set of normal rule and constraints. The Herbrand Base of a program P, denoted HB_P, is the set of variable free (ground) atoms that can be formed from predicates and constants in P. The subsets of HB_P are called the (Herbrand) interpretations of P.

Given a program P and an Herbrand interpretation $I \subseteq HB_P$, the reduct P^I is constructed from the grounding of P in 3 steps: firstly, remove rules whose bodies contain the negation of an atom in I; secondly, remove all negative literals from the remaining rules; and finally, replace the head of any constraint with $\bot$ (where $\bot \notin HB_P$). Any $I \subseteq HB_P$ is an *answer set* of P if it is the minimal model of the *reduct* P^I. Throughout the paper we denote the set of answer sets of a program P with $AS(P)$. Given an answer set $A \in AS(P)$, a ground normal rule of P is satisfied if the head is in A when all positive atoms and none of the negated atoms of the body are in A, that is when the body is satisfied. A ground constraint is satisfied when the body is not satisfied. A constraint has therefore the effect of eliminating all answer sets of P that satisfy the body of the constraint.

Answer Set Grammars

This section formalises Answer Set Grammars (ASGs). We first recall the definition of a CFG (Sipser 1997).

```
1: start -> as bs cs
2: as -> "a" as
3: as ->
4: bs -> "b" bs
5: bs ->
6: cs -> "c" cs
7: cs ->
```

$trace(n)$	$node(n)$	$rule(n)$
$[]$	start	1
$[1]$	as	2
$[1,1]$	a	
$[1,2]$	as	3
$[2]$	bs	5
$[3]$	cs	6
$[3,1]$	c	
$[3,2]$	cs	7

Figure 1: A CFG for $\mathtt{a^i b^j c^k}$ and the parse tree for ac.

Definition 1. *A* context-free-grammar G_{CF} *is a tuple* $\langle G_N, G_T, G_{PR}, G_S\rangle$ *where* G_N *is a (finite) set of non-terminal nodes,* G_T *is a (finite) set, disjoint from* G_N*, of terminal nodes,* G_{PR} *is a set of production rules of the form* $n_0 \rightarrow n_1 \ldots n_k$*, where* $n_0 \in G_N$ *and each* $n_i \in G_N \cup G_T$*.* $G_S \in G_N$ *is the start node of* G_{CF}*.*

Terminal nodes correspond to the characters of the alphabet that appear in the strings generated by the grammar. So, for each node $n \in G_T$, we say that n *yields* the string "n". Production rules are used to generate all possible strings in the formal language formalised as the context-free grammar. So, for any production rule $n_0 \rightarrow n_1 \ldots n_k$ in G_{PR}, if for each $i \in [1, k]$, n_i yields the string s_i then we say that n_0 *yields* the string "$s_1 \ldots s_k$". A string s is said to be in the language of a grammar G, denoted as $\mathcal{L}(G)$, if and only if the start node G_S of the grammar *yields* the string s. We can now define the notion of a parse tree for a CFG.

Definition 2. *Let* $G_{CF} = \langle G_N, G_T, G_{PR}, G_S\rangle$ *be a CFG. A* parse tree PT *of* G_{CF} *consists of a node* $node(PT)$ *in* $G_T \cup G_N$*, a list of parse trees, called* children *and denoted* $children(PT)$*, and if* $node(PT) \in G_N$*, a rule* $rule(PT) \in PR$*, such that*

1. *If* $node(PT) \in G_T$*, then* $children(PT)$ *is empty.*
2. *If* $node(PT) \in G_N$*, then* $rule(PT)$ *is of the form* $node(PT) \rightarrow n_1 \ldots n_k$ *where each* n_i *is equal to* $node(children(PT)[i])$ *and* $|children(PT)| = k$*.*

For any parse tree PT we define str(PT) as: (1) $node(PT)$ *if* $node(PT) \in G_T$*; and (2)* $str(PT_1) \ldots str(PT_n)$ *otherwise (where* $[PT_1, \ldots, PT_n] = children(PT)$*). A parse tree PT of* G_{CF} *is a parse tree for a string s if* $node(PT) = G_S$ *and* $s = str(PT)$*.*[1]

We can represent each node n in a parse tree by its trace, $trace(n)$, through the tree. The trace of the root is the empty list $[]$; the i^{th} child of the root is $[i]$; the j^{th} child of the i^{th} child of the root is $[i, j]$, and so on.

Example 1. Consider the CFG given in Figure 1. Note that we give each production rule a unique integer identifier. There is exactly one[2] parse tree PT of G_{CF} for the string ac, which is given by the table in Figure 1.

[1]Note that in the case that a string is empty, all nodes of the tree will correspond to non terminals. The leaf nodes of the tree will be non-terminals with an empty list of children. For "" to be accepted, it is therefore necessary for at least one production rule to have no terminal or non-terminal symbols on the right hand side.

[2]In fact, this grammar is *unambiguous*, meaning that each string that is accepted by the grammar has a unique parse tree.

An *Answer Set Grammar* extends a context-free grammar (CFG) by expressing semantic conditions, written in ASP, on the production rules. To allow the semantic conditions to refer to the structure of the CFG, we use the notion of *annotated ASP programs*. These are programs whose atoms have annotations that refer to nodes of the parse tree of a CFG. Specifically, an annotated ASP program is an ASP program where some atoms have been *annotated* with a ground term. For instance, the annotated atom $\mathtt{a(1)@2}$ represents the atom $\mathtt{a(1)}$ with the annotation 2. When computing the answer sets of an annotated program, annotated atoms are treated as ordinary atoms, where $\mathtt{a@k}$, $\mathtt{a@l}$ and $\mathtt{a}$ are distinct atoms. We can now define the notion of annotated production rules.

Definition 3. *An* annotated production rule *is of the form* $n_0 \rightarrow n_1 \ldots n_k\ P$ *where* $n_0 \rightarrow n_1 \ldots n_k$ *is an ordinary CFG production rule and* P *is an annotated ASP program, where every annotation is an integer between* 1 *and* k.

Definition 4. *An* answer set grammar G *is a tuple* $\langle G_N, G_T, G_{PR}, G_S\rangle$ *where* G_N *is a (finite) set of non-terminal nodes,* G_T *is a (finite) set, disjoint from* G_N *of terminal nodes,* G_{PR} *is a set of annotated production rules and* $G_S \in G_N$ *is the start node of* G.

Given any ASG G, we denote with G_{CF} the context-free part of the grammar; i.e. the CFG constructed from G by removing the ASP programs from each production rule.

Example 2. The following is an example of an ASG G.

```
1: start -> as bs cs
                    {:- size(X)@1, not size(X)@2.
                     :- size(X)@1, not size(X)@3.
                    }
2: as -> "a" as { size(X+1) :- size(X)@2. }
3: as ->        { size(0). }
4: bs -> "b" bs { size(X+1) :- size(X)@2. }
5: bs ->        { size(0). }
6: cs -> "c" cs { size(X+1) :- size(X)@2. }
7: cs ->        { size(0). }
```

The language $\mathcal{L}(G)$ of the above ASG is a subset of the language $\mathcal{L}(G_{CF})$ of the CFG in Example 1, which represents $\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k$. The above ASG G captures the language $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$, where $n \geq 0$. The intuition of the annotations is that in each production rule from 2-7, `size` represents the size of the current string. The atom $\mathtt{size(X)@2}$ in production rule 2 means that the size of the string represented by the second child of the current node (`as`) is `X`. The constraints in production rule 1 enforce that the number of `a`'s, `b`'s and `c`'s are equal.

The notion of parse tree for an ASG is similar to that given in Definition 2 for CFGs, with the difference that rules are annotated production rules.

Definition 5. *Let* $G = \langle G_N, G_T, G_{PR}, G_S\rangle$ *be an ASG. A* parse tree PT *for* G *consists of a node* $node(PT)$ *in* $G_T \cup G_N$*, a list of parse trees called children (denoted* $children(PT)$*), and if* $node(PT) \in G_N$*, a rule* $rule(PT) \in PR$*, such that*

1. *If* $node(PT) \in G_T$*, then* $children(PT)$ *is empty.*
2. *If* $node(PT) \in G_N$*, then* $rule(PT)$ *is of the form* $node(PT) \rightarrow n_1 \ldots n_k\ P$ *where each* n_i *is equal to* $node(children(PT)[i])$ *and* $|children(PT)| = k$.

A parse tree PT of an ASG G for a string s is defined similarly to the context-free case. We can use a parse tree of an ASG to construct an annotated ASP program, which allows us to test whether the parse tree conforms to the semantic conditions of the ASG.

Definition 6. *Let* G *be an ASG and* PT *be a parse tree.* $G[PT]$ *is the program* $\{rule(n)@trace(n) | n \in PT\}$*, where for any production rule* $n_0 \rightarrow n_1 \ldots n_k\ P$*, and any trace* t*,* $PR@t$ *is the program constructed by replacing all annotated atoms* $\mathtt{a@i}$ *with the atom* $\mathtt{a@t}++[\mathtt{i}]$ *and all unannotated atoms* $\mathtt{a}$ *with the atom* $\mathtt{a@t}$.

Definition 7. *Let* G *be an ASG and* str *be a string of terminal nodes.* $str \in \mathcal{L}(G)$ *if and only if there is a parse tree* PT *of* G *for* str *such that* $G[PT]$ *is satisfiable.*

To check, for instance, whether $\mathtt{ac} \in \mathcal{L}(G)$, where G is the ASG given in Example 2, we need to check whether $G[PT]$ is satisfiable. For the string ac, the unique parse tree is that given in Figure 1. Therefore the $G[PT]$ is the program:

```
:- size(X)@[1], not size(X)@[2].
:- size(X)@[1], not size(X)@[3].
size(X+1)@[1] :- size(X)@[1, 2].
size(0)@[1, 2].
size(0)@[2].
size(X+1)@[3] :- size(X)@[3, 2].
size(0)@[3, 2].
```

This program is clearly unsatisfiable, as $\mathtt{size(1)@[1]}$ is guaranteed to be true in all answer sets and $\mathtt{size(1)@[2]}$ is guaranteed to be false in all answer sets, meaning that the first constraint is guaranteed to be violated. If we instead check the string `abc`, this leads to the program:

```
:- size(X)@[1], not size(X)@[2].
:- size(X)@[1], not size(X)@[3].
size(X+1)@[1] :- size(X)@[1, 2].
size(0)@[1, 2].
size(X+1)@[2] :- size(X)@[2, 2].
size(0)@[2, 2].
size(X+1)@[3] :- size(X)@[3, 2].
size(0)@[3, 2].
```

This program has exactly one answer set: $\{\mathtt{size(0)@[1,2]}$, $\mathtt{size(1)@[1]}$, $\mathtt{size(0)@[2,2]}$, $\mathtt{size(1)@[2]}$, $\mathtt{size(0)@[3,2]}$, $\mathtt{size(1)@[3]}\}$ meaning that $\mathtt{abc} \in \mathcal{L}(G)$. This answer set actually reflects the size of the substring in each node of the parse tree (other than the root and the terminal nodes, which do not define size), as shown in the following table.

$trace(n)$	$node(n)$	`size`
$[\,]$	`start`	-
$[1]$	`as`	1
$[1,1]$	`a`	-
$[1,2]$	`as`	0
$[2]$	`bs`	1
$[2,1]$	`b`	-
$[2,2]$	`bs`	0
$[3]$	`cs`	1
$[3,1]$	`c`	-
$[3,2]$	`cs`	0

For any ASG G, we say that G accepts a string str at depth d iff there is a parse tree PT of G for str of depth less than or equal to d such that $G[PT]$ is satisfiable. We denote the set of all strings that are accepted by G at depth d as $\mathcal{L}^d(G)$.

Answer Set Grammar Induction

In this section, we formalise our framework for learning answer set grammars. Informally, our framework takes as input an ASG G (or even simply a G_{CF}), and two sets of strings, E^+ and E^-, and learns a target ASG $G^{'}$ such that the $G^{'}_{CF}$ is the same as the G_{CF} of the input grammar, and every string in E^+ (resp.E^-) is accepted (resp. rejected) by the target grammar $G^{'}$. So the framework enables learning the ASP part of an ASG. We are not learning the "context-free" part of the grammar, but only the semantic conditions, assuming therefore that the syntax of the target language is known, but the semantics is unknown. To perform the learning, our framework uses recent advances in Inductive Learning of ASP (ILASP) from (Law, Russo, and Broda 2015a).

Similarly to most ILP techniques, our framework includes a notion of *hypothesis space*, which defines the set of rules that can form possible learned outcomes, referred also as possible *hypothesis*. For convenience, hypothesis spaces are often characterised by a set of *mode declarations* (Muggleton 1995), often referred to as biases. We use a similar approach, but in order to give the flexibility that some predicates may only be used in some production rules, we add a parameter called the *scope* of a mode declaration. We build upon the notion of Learning from Answer Sets (LAS) mode declarations given in (Law, Russo, and Broda 2014), and define the language bias of our framework by extending this notion with the scope of each mode declaration. For instance the mode declaration $\#\mathtt{modeb}(\mathtt{p}(\mathtt{var}(\mathtt{t})), [1, 2, 3])$ means that this mode declaration only applies to production rules 1–3. An *ASG hypothesis space* is a set of pairs of the form $\langle PR_{id}, R\rangle$, where PR_{id} is an identifier for a production rule and R is an annotated ASP rule. Each pair $\langle PR_{id}, R\rangle$ in a hypothesis space means that R can be added to the annotation of PR. Given any ASG G and any hypothesis[3] H, $G\!:\!H$ is the ASG constructed by adding R to the annotation of PR for each $\langle PR_{id}, R\rangle$ in H.

As any other form of grammar induction, to learn an ASG we also need examples of strings that should, or should not, be accepted by the final grammar. The former are positive examples, whereas the latter are negative examples. For instance, if we were aiming to learn the grammar $a^n b^n c^n$, we might give positive examples like "abc" and "aabbcc", as strings that should be accepted, and negative examples like "aabcc" as a string that should not be accepted. A learning task for ASGs consists then of an existing ASG, an ASG hypothesis space, and two sets of strings called the positive and negative examples.

Definition 8. *An ASG learning task T is of the form $\langle G, S_M, E^+, E^-\rangle$, where G is an ASG called the* existing grammar*, S_M is an ASG hypothesis space and E^+ and E^- are sets of strings called the* positive *and* negative *examples, respectively. An* inductive solution *of T at depth d is a hypothesis $H \subseteq S_M$ such that (1) $\forall s \in E^+, s \in \mathcal{L}^d(G : H)$; and (2) $\forall s \in E^-, s \notin \mathcal{L}^d(G : H)$. $ILP^d_{ASG}(T)$ denotes the set of all inductive solutions of T at depth d.*

Note that the existing grammar G of an ASG learning task may or may not contain ASP rules on some or all of the production rules. The ASP rules that are present constitute what is also known as *background knowledge* in a conventional ILP task, as they are used to encode knowledge that is already known. In the extreme, this knowledge can all be learned[4]. But, as shown in our evaluation, when some semantic conditions are already known, including them in the existing grammar can reduce the number of examples and time needed to learn the correct target grammar.

Computational Complexity

We consider two classes of decision problem. The first addresses the complexity of decision problems for existing ASGs, and the second the complexity of the learning task. Similar to the complexity results of ASP with bounded predicate arities (e.g. (Eiter et al. 2007)), our results are presented in terms of a bound on the maximum depth of parse trees.

Grammar decision problems:

- *Bounded-ASG-membership* is the problem of deciding whether an ASG accepts a string.
- *Bounded-ASG-satisfiability* is the problem of deciding whether an ASG has a non-empty language.

Results are summarised in Table 1 and the proofs can be found in the supplemental material. An ASG G is *unstratified* if there is at least one parse tree PT of G such that $G[PT]$ is unstratified. Results in the second row are for function free first-order ASP annotations. One observation to make is that for all the hardness proofs one can use grammars that generate finite languages. Hence, the complexity of the context-free part of the grammar has no effect on the complexity of the ASG. Nevertheless, the grammar is needed to get the lower bound of the classes of ASGs since Horn ASP alone is in P. The results also show that we would need first order Horn to capture CSGs. However, it is an open question whether there is a class of ASGs exactly capturing CSGs since they are PSPACE-complete.

Learning decision problems:

- *Bounded-verification* is the problem of deciding whether a given hypothesis is an inductive solution of a given learning task.
- *Bounded-satisfiability* is the problem of deciding whether a given learning task has any inductive solutions.

Results for the learning task are limited to propositional ASP[5] and are summarised in Table 2. The proofs can also be found in the supplemental material. The observation to make is that the complexities for bounded-verification and bounded-satisfiability under propositional ASP are identical to the complexities for verification and satisfiability of $ILP^{context}_{LAS}$ tasks (Law, Russo, and Broda 2018). These equivalences provided a strong hint that we could solve any ASG learning task by encoding it as an $ILP^{context}_{LAS}$ task, and

[3]A subset of the hypothesis space.

[4]This is the case when the existing G is a pure CFG.

[5]Our implementation is able to learn first-order ASGs.

	Horn	Stratified	Unstratified
Propositional	NP	NP	NP
First-order	EXP	EXP	NEXP

Table 1: Complexity classes for propositional and first-order ASGs on the decision problems of bounded-ASG-membership and bounded-ASG-satisfiability (each element of the table is the complexity class for which both decision problems are complete).

	Horn	Stratified	Unstratified
Bounded-verification	DP	DP	DP
Bounded-satisfiability	Σ_2^P	Σ_2^P	Σ_2^P

Table 2: Complexity results for learning propositional ASGs. Each entry in the table is the complexity class for which the given decision problem is complete.

use the existing ILASP (Law, Russo, and Broda 2015a) system in order to solve the ASG learning task. We will show in the next section how this can be done. Furthermore, we will also show that, with some restrictions, stratified bounded-satisfiability is in NP. The complexity of the learning task for the function-free first-order case is still an open question.

Learning ASGs with ILASP

This section describes our method for solving ASG learning tasks. We use the ILASP (Inductive Learning of Answer Set Programs) system (Law, Russo, and Broda 2015a), as a black box. We transform our task into a task that can be solved by ILASP. For completeness, we summarise existing notions that are key for the learning tasks solved by ILASP.

Definition 9. *((Law, Russo, and Broda 2016)) A* context-dependent partial interpretation *(CDPI) is a pair $e = \langle\langle e^{inc}, e^{exc}\rangle, e^{ctx}\rangle$, where $\langle e^{inc}, e^{exc}\rangle$ is pair of sets of atoms called a* partial interpretation *and e^{ctx} is an ASP program, called a* context*. A program P is said to* accept *e iff there is an answer set A of $P \cup e^{ctx}$ such that $e^{inc} \subseteq A$ and $e^{exc} \cap A = \emptyset$.*

An $ILP_{LAS}^{context}$ learning task can be defined as follows.

Definition 10. *An $ILP_{LAS}^{context}$ task is a tuple $T = \langle B, S_M, \langle E^+, E^-\rangle\rangle$ where B is an ASP program, S_M is the set of rules allowed in the hypotheses and E^+ and E^- are finite sets of CDPIs called, respectively, positive and negative examples. A hypothesis $H \subseteq S_M$ is an inductive solution of T (written $H \in ILP_{LAS}^{context}(T)$) if and only if: (1) $\forall e^+ \in E^+$, $B \cup H$ accepts e^+; and (2) $\forall e^- \in E^-$, $B \cup H$ does not accept e^-.*

Definition 10 is a simplified version of the full $ILP_{LOAS}^{context}$ framework presented in (Law, Russo, and Broda 2016). To define the transformation of an ASG learning task into an $ILP_{LAS}^{context}$ task, we use the following notation. For any ASP rule R, $R_X(PR_{id})$ denotes the rule constructed from R in two steps: (1) replacing each annotated atom $\mathtt{a@[t_1, \ldots, t_n]}$ with the atom $\mathtt{ann(a, X++[t_1, \ldots, t_n])}$; and (2) adding the atom $\mathtt{pr(PR_{id}, [t_1, \ldots, t_n])}$ to the body of the rule.[6] The mapping in Definition 11 translates each rule R that occurs (resp. could occur) in the annotation of each production rule PR to $R_X(PR_{id})$ putting it in the background knowledge (resp. hypothesis space) of the $ILP_{LAS}^{context}$ task. The intuition is that for any $ILP_{LAS}^{context}$ hypothesis H and any parse tree $G[PT]$ (for some string s), $B \cup H \cup \{\mathtt{pr(rule(n)_{id}, trace(n))}.| n \in PT\}$ is satisfiable if and only if $(G : H')[PT]$ is satisfiable (where H' is the ASG hypothesis represented by H). Contexts of the positive (resp. negative) examples ensure that for each positive (resp. negative) string example there is at least one[7] (resp. no) parse tree of G such that $(G : H')[PT]$ is satisfiable.

Definition 11. *Let d be a positive integer and $T = \langle G, S_M, E^+, E^-\rangle$ be an ASG learning task. $LAS(T, d)$ is the $ILP_{LAS}^{context}$ task $\langle B, S_M^{LAS}, E_{LAS}^+, E_{LAS}^-\rangle$, where the individual components are defined as follows.*

- $B = \left\{ R_X(PR_{id}) \;\middle|\; \begin{array}{c} PR \in G_{PR}, \\ PR = n \rightarrow n_1 \ldots n_k\ P, \\ R \in P \end{array} \right\}$
- $S_M^{LAS} = \{R_X(PR_{id}) | \langle PR_{id}, R\rangle \in S_M\}$
- *E_{LAS}^+ contains one CDPI $\langle\langle\emptyset, \emptyset\rangle, C\rangle$ for each string $s \in E^+$, where given $\{PT_1, \ldots, PT_m\}$, the set of parse trees of s for G_{CF} at depth d, $C = \{\mathtt{1\{pt_1, \ldots, pt_m\}1.}\} \cup \{\mathtt{pr(rule(n)_{id}, trace(n))\,\text{:-}\,pt_i.} | i \in [1, m], n \in PT_i\}$*
- *E_{LAS}^- is the set of CDPIs of the form $\langle\langle\emptyset, \emptyset\rangle, \{\mathtt{pr(rule(n)_{id}, trace(n))}. | n \in PT\}\rangle$, where PT is a parse tree of a string in E^- for G_{CF}.*

Given any hypothesis $H \subseteq S_M^{LAS}$, we write H^{ASG} to denote the hypothesis $\{\langle PR_{id}, R\rangle \in S_M \mid R_X(PR_{id}) \in H\}$.

Theorem 1 shows that we can use the mapping in Definition 11 to translate any ASG learning task T and depth d, and use ILASP to find the solutions in $ILP_{ASG}^d(T)$. The proof of Theorem 1 is given in the supplemental material.

Theorem 1. *Let T be an ASG learning task. $ILP_{ASG}^d(T) = \{H^{ASG} \mid H \in ILP_{LAS}^{context}(LAS(T, d))\}$*

Learning stratified ASGs

Full $ILP_{LAS}^{context}$ tasks are Σ_2^P complete (in the propositional case); however, if there are no negative examples then the complexity is only NP-complete. Tasks with no negative examples tend to run faster in ILASP than equivalent tasks with negative examples (Law, Russo, and Broda 2015a), and so when it is possible to modify the representation to eliminate negative examples, it is often advantageous to do so. When ASGs are stratified, the representation $LAS(T, d)$ used by Theorem 1 can be modified to use only positive examples. This can be achieved by representing each constraint `:- body` as the rule `vio :- body` (where `vio` is a new atom that indicates that at least one constraint has been violated). The positive CDPI examples in $LAS(T, d)$ are then extended to indicate that the unique answer set of the (stratified) program should not prove `vio` (which is equivalent to saying that none of the constraints should be violated). The negative examples in $LAS(T, d)$ are represented similarly

[6]Lists are represented as pairs (for example, $\mathtt{X++[t_1, \ldots, t_3]}$ is represented as $\mathtt{(((X, t_1), t_2), t_3)}$).

[7]The context of each positive example has a choice rule that means the program checks for the existence of such a parse tree.

(again as positive examples), but indicating that the unique answer set of the (stratified) program must contain `vio`. This means that the unique answer set must violate at least one constraint in order to cover the example.

If each example string has only a polynomial number of parse trees then this task is polynomial in size of the ASG learning task[8]. Hence in the propositional case, learning stratified ASGs is in NP, provided that each example string has a polynomial number of parse trees.

Evaluation

This section summarises experimental results of using our approach to induce ASGs. The approach was evaluated on several context-sensitive languages, including some languages drawn from a related paper targeting learning mildly context-sensitive (MCS) languages represented as linear indexed grammars (LIGs) (Nakamura and Imada 2011). The languages learned in this section are:

- The `copy` language: `ww`, where `w` is a non-empty string of `a`'s and `b`'s. The input language was a CFG corresponding to the language $\mathtt{w_1w_2}$ where $\mathtt{w_1}$ and $\mathtt{w_2}$ are both non-empty strings of `a`'s and `b`'s.
- The language $\mathtt{a^nb^nc^m}$, $\mathtt{n} \leq \mathtt{m}$: the input language was a CFG corresponding to the language $\mathtt{a^ib^jc^k}$ and the task was to learn annotations expressing that $i = j \leq k$.
- The language $\mathtt{a^nb^nc^n}$. We considered four different learning tasks for this language:
  - **A**: the input language was a CFG corresponding to the language $\mathtt{a^ib^jc^k}$ and the task was to learn annotations expressing that $i = j = k$.
  - **B**: the input language was a CFG corresponding to the language $\mathtt{a^nb^nc^m}$ and the task was to learn annotations expressing that $n = m$.
  - **C**: the input language was an ASG G such that G_{CF} corresponds to the language $\mathtt{a^ib^jc^k}$, but the existing annotations correspond to $i = j$. The task was to extend the annotations to express that $i = k$. There is slightly less to learn in this task than in $\mathtt{a^nb^nc^n}$ **B**, as some of the annotations necessary to express that $i = k$ are already present (i.e. definition of `size` in the `as` production rules); however the hypothesis space is larger in this task, as there are more production rules which could possibly be annotated.
  - **D**: the input language was a CFG representing the language (`a`|`b`|`c`)∗. The task was to learn annotations expressing that all `a`'s occur before all `b`'s, which occur before all `c`'s, and that the number of `a`'s, `b`'s and `c`'s are equal. Essentially this task corresponds to learning both the CFG and the ASG annotations, as the input ASG represents the full language of `a`'s, `b`'s and `c`'s.

[8]The mapping can be converted to a propositional mapping by replacing R_X with R_T for each trace $[t_1, \ldots, t_i]$, where $i \in [0, d]$ and each $t_j \in [1, \mathtt{max\_k}]$, where `max_k` is the number of terminal and non-terminal nodes in the body of the longest production rule. There are a polynomial number of such traces, so this "grounding" of the problem is still polynomial in the size of the input problem.

Language	Final Time	Total Time	$\lvert E^+\rvert$	$\lvert E^-\rvert$
`copy`	501.8s	1194.9s	1	12
$\mathtt{a^nb^nc^m}$, $\mathtt{n} \leq \mathtt{m}$	1.1s	6.6s	2	8
$\mathtt{a^nb^nc^n}$ **A**	1.1s	5.1s	1	7
$\mathtt{a^nb^nc^n}$ **B**	0.4s	1.4s	1	3
$\mathtt{a^nb^nc^n}$ **C**	0.5s	1.5s	1	2
$\mathtt{a^nb^nc^n}$ **D**	1020.6s	13393.9s	1	46
$\mathtt{a^nb^mc^nd^m}$	1.8s	11.7s	2	9
`subset-sum`	16.8s	184.4s	4	10

Table 3: A summary of the results of our evaluation.

- The language $\mathtt{a^nb^mc^nd^m}$: the input language was a CFG corresponding to the language $\mathtt{a^ib^jc^kd^l}$ and the task was to learn annotations expressing that $i = k$ and $j = l$.
- The `subset-sum` language: the input language was a CFG corresponding to the language of sets of integers between -5 and 5 (e.g. $\{-5, 4, 1\}$). The task was to learn annotations expressing that at least one non-empty subset of the integers in the string sums to 0. Note that this language is not MCS – for a language to be MCS, its membership decision problem must be in P, but deciding if at least one non-empty subset of a set of integers sums to 0 is NP-complete.

For each language, we evaluated our framework using an iterative approach. In the first iteration the learner started with the initial language detailed above. In each subsequent iteration the learner's current language was checked against the target language for a counter example (either a positive example of a string not in the learner's language that was in the target language, or a negative example for the reverse situation). If such a counter example existed, the shortest such example was added to the examples of the learning task at the next iteration. The iterations, and therefore the learning, terminated when no such counter examples existed.

Table 3 summarises the results of our evaluation. The *Final Time* and *Total Time* show the learning time [9] taken in the final iteration and the total learning time, respectively. The last two columns show the number of positive and negative examples needed to learn the target language in each case. The four $\mathtt{a^nb^nc^n}$ examples show the effect of narrowing (in the case of **B** and **C**) and widening (in the case of **D**) the initial language. In the case of **D**, essentially the CFG needed to be learned, in addition to the constraints represented by the original ASG annotations (in **A**). In general, the more specific the initial language, the fewer examples are needed to learn the constrained target language. In each case, fewer positive examples were needed than negative examples, which is to be expected as the task is to constrain the initial language (of which the target language is a subset).

(Nakamura and Imada 2011) evaluate their LIG learner on each of the above languages other than `subset-sum` and the variations of $\mathtt{a^nb^nc^n}$. LIGs make no distinction between

[9]The experiments used an Ubuntu 14.04 desktop machine with a 3.4 GHz Intel® Core™ i7-3770 processor and 16GB RAM.

the context-free part of the grammar so in their case the entire LIG was learned from scratch. As expected, our own running times are significantly shorter in most cases. We are slower for the copy language and the **D** variation of $\mathtt{a}^\mathtt{n}\mathtt{b}^\mathtt{n}\mathtt{c}^\mathtt{n}$. The copy language has a much more natural representation in LIG, which allows the use of a stack to "store" the copied string, whereas ASP does not have a natural efficient representation of lists or stacks. The **D** variation of $\mathtt{a}^\mathtt{n}\mathtt{b}^\mathtt{n}\mathtt{c}^\mathtt{n}$ takes significantly longer in our case (1020.6s compared to 130s) as our ASG learner was not designed to learn the full language of an ASG, but to restrict an existing context-free language. As shown in the other three cases, when starting from a more restricted CFG, our approach is able to use the existing grammar to find the complete ASG much faster than either method can learn the entire grammar from scratch. As LIGs are MCS, the approach in (Nakamura and Imada 2011) could not learn the subset-sum language.

Related Work

Context-sensitive grammars are often used in specifying the semantics of programming languages. For instance, attribute grammars (Knuth 1968; 1990) are used in tools for generating parsers and compilers (for example, YACC (Mason and Brown 1990) and ANTLR (Parr and Quong 1995)). Attribute grammars are defined in terms of *synthesised* and *inherited* attributes, which are passed up and down (respectively) the parse tree of a context-free grammar. Each atom in the Herbrand Base of an ASP program in an ASG can be thought of as a boolean attribute. These attributes can be defined in terms of the values of parent nodes (similarly to inherited attributes) by using rules in the production rule of the parent node to pass attributes to the child – this is achieved using rules with annotated atoms in the head. Synthesised attributes, on the other hand, can be simulated by using rules with annotations in the body – this causes attribute values to be "passed" from the child to the parent. Through first-order representations, ASGs can simulate attributes of non-boolean types; e.g. in the $a^nb^nc^n$, the predicate size is used to record the (integer) size of the string.

There are several differences between attribute grammars and ASGs. Firstly, ASGs are not defined in terms of attributes which are passed in a single direction along the parse tree. This is because they are not implemented in a procedural way, but are instead purely declarative. At a conceptual level, the child nodes and parent nodes are all evaluated simultaneously, and attributes can in fact be *both* inherited and synthesised at the same time. There can even be recursion between two nodes.

Example 3. Consider the following ASG, which corresponds to conjunctions in ASP. For simplicity, we assume there are production rules defining literals whose ASP rules define the predicate uses_var(V), which is true if and only if the literal uses the variable V, and the atom positive which is true if and only if the literal is positive. The ASP programs in the production rules check whether the conjunction is *safe*, i.e. whether every variable that occurs in the conjunction occurs in at least one positive literal in the conjunction. The start node of the grammar is conjunction.

```
conjunction -> literal {
  safe(V) :- uses_var(V)@1, positive@1.
  :- uses_var(V)@1, not safe(V).
}
conjunction -> conjunction "," literal {
  safe(V) :- safe(V)@1.
  safe(V) :- safe(V)@3.
  safe(V)@1 :- safe(V).
  safe(V)@3 :- safe(V).
}
```

In the second production rule, safe/1 acts as both a synthesised attribute and an inherited attribute. As ASP uses the closed world assumption, there must be some external support for safe(V) to be satisfied – i.e. at least one of the nodes in the first production rule must prove safe(V). This means that for each node of the parse tree corresponding to a conjunction, safe(V) holds iff there is at least one positive literal in the conjunction that uses the variable V.

An additional advantage of using a logic programming formalism to express the semantic conditions is that there is a large amount of work on Inductive Logic Programming (for example, (Muggleton 1991; 1995; Ray 2009; Sakama and Inoue 2009; Corapi, Russo, and Lupu 2010; Muggleton et al. 2012; Law, Russo, and Broda 2014)). This means that we can delegate the learning of the ASP programs to an existing ILP system for learning ASP programs (Law, Russo, and Broda 2015a).

There has been a lot of work on Grammar Induction (for example, (Angluin 1987; Wyard 1993)). In (Fredouille et al. 2007; Muggleton et al. 2014), various ILP approaches have been used to learn context-free grammars in the form of DCGs. There is also some work on learning context-sensitive grammars (e.g. (Oates et al. 2006; Yoshinaka 2009; Imada and Nakamura 2010; Nakamura and Imada 2011)), but we are not aware of any work on learning attribute grammars, or learning semantic conditions on top of an existing context-free grammar. Such a task is useful to provide practical solutions to the problems of parser corrections described in the introduction, as it is faster to learn to restrict the existing grammar than to learn the whole grammar from scratch.

Conclusion

In this paper we have presented a method for learning context-sensitive constraints on an existing context-free grammar. We have introduced a new context-sensitive grammar based on Answer Set Programming, which allows us to delegate the learning to an existing ASP learner.

In the general case, the computational complexity of deciding verification and satisfiability is the same as the equivalent complexity results for the ASP learning task, but we have shown that by restricting the language of ASP that is allowed in the grammar, the complexity of these decision problems can be significantly lowered. We have shown that in such special cases, a more efficient representation of the learning task can be used.

Our assumption that the CFG is known at the beginning means that this method is appropriate for tasks where the underlying syntax of the language is known, but (some of) the semantic constraints are unknown.

References

Alnabulsi, H.; Islam, M. R.; and Mamun, Q. 2014. Detecting sql injection attacks using snort ids. In *Asia-Pacific World Congress on Computer Science and Engineering*, 1–7.

Angluin, D. 1987. Learning regular sets from queries and counterexamples. *Inf. Comput.*

Corapi, D.; Russo, A.; and Lupu, E. 2010. Inductive logic programming as abductive search. In Hermenegildo, M. V., and Schaub, T., eds., *Technical Communications of the Twenty-sixth International Conference on Logic Programming, July 16-19, 2010, Edinburgh, Scotland, UK*, volume 7 of *LIPIcs*, 54–63. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Eiter, T.; Faber, W.; Fink, M.; and Woltran, S. 2007. Complexity results for answer set programming with bounded predicate arities and implications. *Annals of Mathematics and Artificial Intelligence* 51(2-4):123.

Fredouille, D. C.; Bryant, C. H.; Jayawickreme, C. K.; Jupe, S.; and Topp, S. 2007. An ilp refinement operator for biological grammar learning. In Muggleton, S.; Otero, R.; and Tamaddoni-Nezhad, A., eds., *Inductive Logic Programming*, 214–228. Berlin, Heidelberg: Springer Berlin Heidelberg.

Godefroid, P.; Peleg, H.; and Singh, R. 2017. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 50–59. IEEE Press.

Imada, K., and Nakamura, K. 2010. Search for minimal and semi-minimal rule sets in incremental learning of context-free and definite clause grammars. *IEICE TRANSACTIONS on Information and Systems* 93(5):1197–1204.

Javed, F.; Bryant, B. R.; Črepinšek, M.; Mernik, M.; and Sprague, A. 2004. Context-free grammar induction using genetic programming. In *Proceedings of the 42nd annual Southeast regional conference*, 404–405. ACM.

Johnson, S. C. 1975. *Yacc: Yet Another Compiler Compiler*. Murray hill, New Jersey, USA: Bell Laboratories.

Knuth, D. E. 1968. Semantics of context-free languages. *Mathematical Systems Theory* 2(2):127–145.

Knuth, D. E. 1990. The genesis of attribute grammars. In *Proceedings of the International Conference on Attribute Grammars and Their Applications*, WAGA, 1–12. New York, NY, USA: Springer-Verlag New York, Inc.

Law, M.; Russo, A.; and Broda, K. 2014. Inductive learning of answer set programs. In Fermé, E., and Leite, J., eds., *Proceedings of the Fourteenth European Conference on Logics in Artificial Intelligence, 2014, Funchal, Madeira, Portugal, September 24-26, 2014.*, volume 8761 of *Lecture Notes in Computer Science*, 311–325. Springer.

Law, M.; Russo, A.; and Broda, K. 2015a. The ILASP system for learning answer set programs. `https://www.doc.ic.ac.uk/~ml1909/ILASP`.

Law, M.; Russo, A.; and Broda, K. 2015b. Learning weak constraints in answer set programming. *Theory and Practice of Logic Programming* 15(4-5):511–525.

Law, M.; Russo, A.; and Broda, K. 2016. Iterative learning of answer set programs from context dependent examples. *Theory and Practice of Logic Programming* 16(5-6):834–848.

Law, M.; Russo, A.; and Broda, K. 2018. The complexity and generality of learning answer set programs. *Artificial Intelligence* 259:110–146.

Mason, T., and Brown, D. 1990. *Lex & Yacc*. Sebastopol, CA, USA: O'Reilly & Associates, Inc.

Muggleton, S.; De Raedt, L.; Poole, D.; Bratko, I.; Flach, P.; Inoue, K.; and Srinivasan, A. 2012. ILP turns 20. *Machine Learning* 86(1):3–23.

Muggleton, S. H.; Lin, D.; Pahlavi, N.; and Tamaddoni-Nezhad, A. 2014. Meta-interpretive learning: Application to grammatical inference. *Mach. Learn.* 94(1):25–49.

Muggleton, S. 1991. Inductive logic programming. *New Generation Computing* 8(4):295–318.

Muggleton, S. 1995. Inverse entailment and Progol. *New Generation Computing* 13(3-4):245–286.

Nakamura, K., and Imada, K. 2011. Towards incremental learning of mildly context-sensitive grammars. In *Machine Learning and Applications and Workshops (ICMLA), 2011 10th International Conference on*, volume 1, 223–228. IEEE.

Oates, T.; Armstrong, T.; Bonache, L. B.; and Atamas, M. 2006. Inferring grammars for mildly context sensitive languages in polynomial-time. In Sakakibara, Y.; Kobayashi, S.; Sato, K.; Nishino, T.; and Tomita, E., eds., *Grammatical Inference: Algorithms and Applications*, 137–147. Berlin, Heidelberg: Springer Berlin Heidelberg.

Parr, T. J., and Quong, R. W. 1995. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience* 25(7):789–810.

Pereira, F. C. N., and Warren, D. H. D. 1980. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence* 13:231–278.

Ray, O. 2009. Nonmonotonic abductive inductive learning. *Journal of Applied Logic* 7(3):329–340.

Sakama, C., and Inoue, K. 2009. Brave induction: A logical framework for learning from incomplete information. *Machine Learning* 76(1):3–35.

Sipser, M. 1997. *Introduction to the Theory of Computation*. PWS Publishing.

Sutton, M.; Greene, A.; and Amini, P. 2007. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional.

Tang, L.; Tao, L.; and Chang-Shing, P. 2011. Logsig: Generating system events from raw textual logs. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, 785–794. ACM.

Wyard, P. 1993. Context free grammar induction using genetic algorithms. In *Grammatical Inference: Theory, Applications and Alternatives, IEE Colloquium on*, P11–1. IET.

Yoshinaka, R. 2009. Learning mildly context-sensitive languages with multidimensional substitutability from positive data. In *International Conference on Algorithmic Learning Theory*, 278–292. Springer.

Proofs

Theorem 1. *Let T be an ASG learning task. $ILP^d_{ASG}(T) = \{H^{ASG} \big| H \in ILP^{context}_{LAS}(LAS(T,d))\}$*

Proof. Let T be the ASG learning task $\langle G, S_M, E^+, E^- \rangle$.

Let $LAS(T,d) = \langle B_{LAS}, S^{LAS}_M, E^+_{LAS}, E^-_{LAS} \rangle$. Given any hypothesis $H \subseteq S_M$, we write H^{LAS} to denote the hypothesis $\{\langle R_X(PR_{id}) \in H | PR_{id}, R \rangle \in S_M\}$.

$(*)$ First note that for any parse tree PT of $(G:H)_{CF}$ of depth d, (for any $H \subseteq S_M$) $(G:H)[PT]$ is satisfiable if and only if $\left\{ R_X(PR_{id}) \middle| \begin{array}{c} PR \in (G:H)_{PR}, \\ PR = n \rightarrow n_1 \dots n_k\ P, \\ R \in P \end{array} \right\} \cup$ $\{\mathtt{pr(rule(n)_{id}, trace(n)).} | n \in PT\}$ is satisfiable, which is the case if and only if $B \cup H^{LAS}$ accepts $\langle\langle \emptyset, \emptyset \rangle, \{\mathtt{pr(rule(n)_{id}, trace(n)).} | n \in PT\}\rangle$.

Assume that $H \in ILP^d_{ASG}(T)$

$\Leftrightarrow$ $H \subseteq S_M, \forall s \in E^+, s \in \mathcal{L}^d(G:H)$ and $\forall s \in E^-, s \notin \mathcal{L}^d(G:H)$.

$\Leftrightarrow$ $H \subseteq S_M, \forall s \in E^+, \exists PT$ st PT is a parse tree of s for $(G:H)_{CF}$ at depth d and $(G:H)[PT]$ is satisfiable and $\forall s \in E^-, \forall PT$ st PT is a parse tree of s for $(G:H)_{CF}$ at depth d, $(G:H)[PT]$ is unsatisfiable.

$\Leftrightarrow$ $H \subseteq S_M$, $\forall s \in E^+$ st $\{PT_1, \dots, PT_m\}$ is the set of all parse trees of s for $(G:H)_{CF}$ at depth d, $\exists i \in [1,m]$ st $B \cup H^{LAS}$ accepts $\langle\langle \emptyset, \emptyset \rangle, \{\mathtt{pr(rule(n)_{id}, trace(n)).} | n \in PT_i\}\rangle$ and $\forall s \in E^-, \forall PT$ st PT is a parse tree of s for $(G:H)_{CF}$ at depth d, $(G:H)[PT]$ is unsatisfiable. (by $(*)$).

$\Leftrightarrow$ $H \subseteq S_M$, $\forall s \in E^+$ st $\{PT_1, \dots, PT_m\}$ is the set of all parse trees of s for $(G:H)_{CF}$ at depth d, $B \cup H^{LAS}$ accepts $\langle\langle \emptyset, \emptyset \rangle, \{\mathtt{1\{pt_1, \dots, pt_m\}1.}\} \cup \{\mathtt{pr(rule(n)_{id}, trace(n))\,{:}\text{-}\,pt_i.} | i \in [1,m], n \in PT_i\}\rangle$ and $\forall s \in E^-, \forall PT$ st PT is a parse tree of s for $(G:H)_{CF}$ at depth d, $(G:H)[PT]$ is unsatisfiable.

$\Leftrightarrow$ $H \subseteq S_M$, $\forall e^+ \in E^+_{LAS}$, $B \cup H^{LAS}$ accepts e^+ and $\forall s \in E^-, \forall PT$ st PT is a parse tree of s for $(G:H)_{CF}$ at depth d, $(G:H)[PT]$ is unsatisfiable.

$\Leftrightarrow$ $H \subseteq S_M$, $\forall e^+ \in E^+_{LAS}$, $B \cup H^{LAS}$ accepts e^+ and $\forall s \in E^-, \forall PT$ st PT is a parse tree of s for $(G:H)_{CF}$ at depth d, $B \cup H^{LAS}$ does not accept $\langle\langle \emptyset, \emptyset \rangle, \{\mathtt{pr(rule(n)_{id}, trace(n)).} | n \in PT_i\}\rangle$. (by $(*)$).

$\Leftrightarrow$ $H \subseteq S_M$, $\forall e^+ \in E^+_{LAS}$, $B \cup H^{LAS}$ accepts e^+ and $\forall e^- \in E^-_{LAS}$, $B \cup H^{LAS}$ does not accept e^-.

$\Leftrightarrow$ $H^{LAS} \in ILP^{context}_{LAS}(LAS(T))$.

$\Leftrightarrow$ $H \in \{H^{ASG} \big| H \in ILP^{context}_{LAS}(LAS(T,d))\}$

□

Theorem 2. *For any fragment $\mathcal{F}$ of ASP, deciding $\mathcal{F}$ bounded-ASG-satisfiability reduces to deciding $\mathcal{F}$ bounded-ASG-membership.*

Proof. Assume that the depth of parse trees is bounded by some constant d.

Let G be an ASG using some fragment $\mathcal{F}$ of ASP. Let G' be the ASG constructed by removing all terminal symbols from G (and accordingly adjusting all annotations in the ASP of G). "" $\in \mathcal{L}^d(G')$ if and only if $\mathcal{L}^d(G) \neq \emptyset$. G' is also in $ASG^{\mathcal{F}}$, hence deciding $\mathcal{F}$ bounded-ASG-satisfiability reduces to deciding $\mathcal{F}$ bounded-ASG-membership. □

Theorem 3. *For any fragment $\mathcal{F}$ of ASP that contains constraints, deciding $\mathcal{F}$ bounded-ASG-membership reduces to deciding $\mathcal{F}$ bounded-ASG-satisfiability.*

Proof. Assume that the depth of parse trees is bounded by some constant d.

Let G be an ASG in $ASG^{\mathcal{F}}$ and $\mathtt{s}$ be the string $\mathtt{s_1 \dots s_{|s|}}$. To prove the theorem, we show that deciding whether $\mathtt{s} \in \mathcal{L}^d(G)$ reduces to $\mathcal{F}$ bounded-ASG-satisfiability. Let G' be the grammar constructed by extending G in the following way:

- Replace G_S with a new start terminal $\mathtt{start}$, and adding a single production rule $\mathtt{start} \rightarrow \mathtt{start'}\,\{\mathtt{{:}\text{-}\ not\ yields(0, |s| - 1).}\}$ to G_{PR} (where $\mathtt{start'}$ is the original start node of the grammar).
- For each production rule $n \rightarrow n_1 \dots n_k P$ in G_{PR}, add the following rules to P:
  - For each $X \in [0, |s|]$, the fact $\mathtt{yields(X, X, 0)}$.
  - For each $i \in [1,k]$ such that $n_i \in G_T$, for each $X, Y \in [0, |s|]$ such that $s_Y = n_i$, the rule $\mathtt{yields(X, Y+1, i)\,{:}\text{-}\,yields(X, Y, i-1)}$.
  - For each $i \in [1,k]$ such that $n_i \in G_N$, for each $X, Y, Z \in [0, |s|]$, the rule $\mathtt{yields(X, Y, i)\,{:}\text{-}\,yields(X, Y, i-1), yields(Y, Z)@i}$.
  - The rule $\mathtt{yields(X, Y)\,{:}\text{-}\,yields(X, Y, k)}$.

The extra ASP rules in the grammar restrict G so that the only possible string in $\mathcal{L}^d(G')$ is $\mathtt{s}$. This means that $\mathcal{L}(G') \neq \emptyset$ if and only if $\mathtt{s} \in \mathcal{L}^d(G)$. Hence $\mathcal{F}$ bounded-ASG-membership reduces to $\mathcal{F}$ bounded-ASG-satisfiability. □

Theorem 4. *Deciding propositional Horn bounded-ASG-satisfiability is NP-hard.*

Proof. Assume that the depth of parse trees is bounded by some constant d.

We show this by reducing deciding satisfiability of a set of propositional clauses C to deciding whether a propositional Horn ASG has a non-empty language at depth d.

Let $V = \{v_1, \dots, v_n\}$ be the set of atoms in C. For any clause $c \in C$, $constraint(c)$ represents an annotated constraint form of c. For example, $v_1 \vee \neg v_2 \vee \neg v_3$ is represented as $\mathtt{{:}\text{-}\,not\_v@1, v@2, v@3}$.

Consider the following ASG G:

$\mathtt{start} \rightarrow \mathtt{a_1 \dots a_n}\,\{constraint(c) | c \in C\}$
% for each $\mathtt{i} \in [\mathtt{1}, \mathtt{n}]$
$\mathtt{a_i} \rightarrow \{\mathtt{v.}\}$
$\mathtt{a_i} \rightarrow \{\mathtt{not\_v.}\}$

Note that C is satisfiable if and only if there is an interpretation I of the atoms in V such that $\{\mathtt{v@i.} | v_i \in I\} \cup \{\mathtt{not\_v@i.} | v_i \notin I\}\,\{constraint(c) | c \in C\}$ is satisfiable. This is the case if and only if there is an interpretation I of the atoms in V such that I is a model of C. The parse trees of G generate the full set of interpretations of the atoms in

V, and hence there is a parse tree PT of G such that $G[PT]$ is satisfiable if and only if C is satisfiable. □

Corollary 1.

- *Deciding propositional stratified bounded-ASG-satisfiability is NP-hard.*
- *Deciding propositional unstratified bounded-ASG-satisfiability is NP-hard.*
- *Deciding propositional Horn bounded-ASG-membership is NP-hard.*
- *Deciding propositional stratified bounded-ASG-membership is NP-hard.*
- *Deciding propositional unstratified bounded-ASG-membership is NP-hard.*

We say that an annotated ASP program is *groundly annotated* if all its annotations are ground.

Lemma 1. *Deciding the satisfiability of a groundly annotated ASP program reduces to deciding satisfiability of an unannotated ASP program using the same fragment of ASP.*

Proof. Let P be an annotated program. Let P' be the program constructed by replacing each annotated atom $\mathtt{p(t_1, \ldots, t_n)@[a_1, \ldots, a_m]}$ with the atom $\mathtt{p(t_1, \ldots, t_n, annotations, a_1, \ldots, a_m)}$, where `annotations` is a new constant symbol (required to differentiate $\mathtt{p(1,1)@[1]}$ from $\mathtt{p(1)@[1,1]}$, which will be replaced by $\mathtt{p(1,1,annotations,1)}$ and $\mathtt{p(1,annotations,1,1)}$, respectively).

P' is isomorphic to P, and is therefore satisfiable if and only if P is satisfiable. P' also uses the same fragment of ASP. Hence, deciding the satisfiability of an groundly annotated ASP program reduces to deciding satisfiability of an unannotated ASP program using the same fragment of ASP. □

Theorem 5. *Deciding propositional unstratified bounded ASG-satisfiability is a member of NP.*

Proof. Assume that the depth of parse trees is bounded by some constant d.

Let G be a propositional unstratified ASG. Let Π be the following ASP program:

- $\mathtt{node(G_S)@[]}$.
- For each $D \in [1,..,d]$, for each $X_1, X_2, \ldots, X_D \in [1, \mathtt{max\_k}]$:
  - For each $i \in [1, k]$, for each production rule $PR = \mathtt{n \rightarrow n_1 \ldots n_k}\ P$:
    - The rule:
      $\mathtt{node(n_i)@[X_1, X_2, \ldots, X_{D-1}, i]}$:-
      $\quad\mathtt{pr(PR_{ID})@[X_1, X_2, \ldots, X_{D-1}]}$.
    - For each rule $\mathtt{h@a_0}$:- $b_1@a_1, \ldots, b_i@a_1$, not $b_{i+1}@a_{i+1}, \ldots,$ not $b_j@a_j$ in P:
      $\mathtt{h@[X_1, X_2, \ldots, X_D, a_0]}$:-
      $\quad\mathtt{b_1@[X_1, X_2, \ldots, X_D, a_1]}$,
      $\quad\ldots$,
      $\quad\mathtt{b_i@[X_1, X_2, \ldots, X_D, a_i]}$,
      $\quad$not $\mathtt{b_{i+1}@[X_1, X_2, \ldots, X_D, a_{i+1}]}$,
      $\quad\ldots$,
      $\quad$not $\mathtt{b_j@[X_1, X_2, \ldots, X_D, a_j]}$.
  - Let $PR^1, \ldots, PR^m$ be the set of production rules in G_{PR} for n.
    For each node $n \in G_N$, for each $i \in [1, m]$, the rule:
    $\mathtt{pr(PR^i_{ID})@[X_1, X_2, \ldots, X_D]}$:-
    $\quad\mathtt{node(n)@[X_1, X_2, \ldots, X_D]}$
    $\quad$not $\mathtt{pr(PR^1_{ID})@[X_1, X_2, \ldots, X_D]}$,
    $\quad\ldots$
    $\quad$not $\mathtt{pr(PR^{i-1}_{ID})@[X_1, X_2, \ldots, X_D]}$,
    $\quad$not $\mathtt{pr(PR^{i+1}_{ID})@[X_1, X_2, \ldots, X_D]}$,
    $\quad\ldots$
    $\quad$not $\mathtt{pr(PR^m_{ID})@[X_1, X_2, \ldots, X_D]}$.

Π is satisfiable if and only if $\mathcal{L}^d(G) \neq \emptyset$. Hence, deciding whether $\mathcal{L}^d(G) \neq \emptyset$ reduces to the NP problem of deciding the satisfiability of a propositional ASP program consisting of normal rules and constraints. Hence, propositional unstratified bounded ASG-satisfiability is a member of NP. □

Corollary 2.

- *Deciding propositional stratified bounded-ASG-satisfiability is a member of NP.*
- *Deciding propositional Horn bounded-ASG-satisfiability is a member of NP.*
- *Deciding propositional Horn bounded-ASG-membership is a member of NP.*
- *Deciding propositional stratified bounded-ASG-membership is a member of NP.*
- *Deciding propositional unstratified bounded-ASG-membership is a member of NP.*

Corollary 3.

- *Deciding propositional Horn bounded-ASG-satisfiability is NP-complete.*
- *Deciding propositional stratified bounded-ASG-satisfiability is NP-complete.*
- *Deciding propositional unstratified bounded-ASG-satisfiability is NP-complete.*
- *Deciding propositional Horn bounded-ASG-membership is NP-complete.*
- *Deciding propositional stratified bounded-ASG-membership is NP-complete.*
- *Deciding propositional unstratified bounded-ASG-membership is NP-complete.*

Theorem 6. *For any fragment of ASP $\mathcal{F}$ which contains constraints, $\mathcal{F}$ bounded-ASG-satisfiability is $\mathcal{O}$-hard where $\mathcal{O}$ is a complexity class such that deciding brave entailment of $ASP^{\mathcal{F}}$ programs is $\mathcal{O}$-hard.*

Proof. Let P be a program in a fragment $\mathcal{F}$ of ASP that allows constraints and $\mathtt{a}$ be an atom. Let $\mathcal{O}$ be a complexity class such that deciding brave entailment of programs in $\mathcal{F}$ is $\mathcal{O}$-hard. To prove the theorem it suffices to show a polynomial reduction from deciding whether $P \models_b \mathtt{a}$ to deciding whether an $\mathcal{F}$-ASG G is satisfiable.

Consider the following grammar G such that $G_T = \emptyset$, $G_N = \{\mathtt{start}\}$, $G_S = \mathtt{start}$ and G_{PR} contains the single production rule $start \rightarrow P \cup \{\mathtt{:- not\ a.}\}$.

Note that for any positive integer d, $\mathcal{L}^d(G)$ is non-empty iff $P \cup \{\mathtt{:- not\ a.}\}$ is satisfiable, which is the case if and only if P bravely entails $\mathtt{a}$. Hence $\mathcal{F}$ bounded-ASG-satisfiability is $\mathcal{O}$-hard.

□

Corollary 4.

- *First-order Horn bounded-ASG-satisfiability is EXP-hard*
- *First-order stratified bounded-ASG-satisfiability is EXP-hard*
- *First-order unstratified bounded-ASG-satisfiability is $NEXP$-hard*
- *First-order Horn bounded-ASG-membership is EXP-hard*
- *First-order stratified bounded-ASG-membership is EXP-hard*
- *First-order unstratified bounded-ASG-membership is $NEXP$-hard*

Theorem 7. *Deciding first order stratified bounded-ASG-satisfiability is a member of EXP.*

Proof. Assume that the depth of parse trees is bounded by some constant d.

Let G be a first order stratified ASG.

Let $\mathtt{max\_k}$ be the number of nodes in the body of the longest production rule in G_{PR}. Let $pt\_size = \Sigma_{i=0}^{d} max\_k^i$. Note that this is polynomial in the size of G (the value of d is constant).

A parse tree PT is represented as an atom $pt(pr_{id}^1, \ldots, pr_{id}^{pt\_size})$, where for each node $n \in PT$, $pr_{id}^{f(trace(n))} = rule(n)_{id}$, where $f([t_1, \ldots, t_m]) = 1 + \Sigma_{i=1}^{m} max\_k^{m-i} t_i$. For any trace t not present in the parse tree $pr_{id}^{f(t)} = 0$.

Let for each $D \in [0, d]$, let $traces(D)$ be the set of lists $\{[t_1, \ldots, t_D] | \forall i \in [1, D], t_i \in [1, \mathtt{max\_k}]\}$

Let C be the following set of rules:

- For each $m \in [0, d-1]$, each trace $T = [t_1, \ldots, t_m]$ such that $\forall i \in [1, m] : t_i \in [1, \mathtt{max\_k}]$, and each $j \in [1, \mathtt{max\_k}]$, the rule:
  $\mathtt{vio}(\mathtt{X}_1, \ldots, \mathtt{X}_{\mathtt{pt\_size}})\mathtt{:-}$
  $\quad \mathtt{p}(\mathtt{X}_1, \ldots, \mathtt{X}_{\mathtt{pt\_size}}),$
  $\quad \mathtt{X}_{\mathtt{f(T)}} = \mathtt{0},$
  $\quad \mathtt{X}_{\mathtt{f(T++[j])}} \neq \mathtt{0}.$
- For each $m \in [0, d-1]$, each production rule $n \rightarrow n_1 \ldots n_k\ P \in G_{PR}$ with id pr_{id}, and each $T = [t_1, \ldots, t_m]$ such that $\forall i \in [1, m] : t_i \in [1, \mathtt{max\_k}]$:
  - For each $j \in [k+1, \mathtt{max\_k}]$, the rule:
    $\mathtt{vio}(\mathtt{X}_1, \ldots, \mathtt{X}_{\mathtt{pt\_size}})\mathtt{:-}$
    $\quad \mathtt{p}(\mathtt{X}_1, \ldots, \mathtt{X}_{\mathtt{pt\_size}}),$
    $\quad \mathtt{X}_{\mathtt{f(T)}} = \mathtt{pr}_{\mathtt{id}},$
    $\quad \mathtt{X}_{\mathtt{f(T++[j])}} \neq \mathtt{0}.$
  - For each $i \in [1, k]$, and production rule $n' \rightarrow n'_1 \ldots n'_{k'}\ P' \in G_{PR}$ with id pr'_{id} such that $n' \neq n_i$, the rule:
    $\mathtt{vio}(\mathtt{X}_1, \ldots, \mathtt{X}_{\mathtt{pt\_size}})\mathtt{:-}$
    $\quad \mathtt{p}(\mathtt{X}_1, \ldots, \mathtt{X}_{\mathtt{pt\_size}}),$
    $\quad \mathtt{X}_{\mathtt{f(T)}} = \mathtt{pr}_{\mathtt{id}},$
    $\quad \mathtt{X}_{\mathtt{f(T++[i])}} = \mathtt{pr}'_{\mathtt{id}}.$

Consider the program $Pi \cup C \cup \{\mathtt{p}(\mathtt{X}_1, \ldots, \mathtt{X}_{\mathtt{pt\_size}})\mathtt{:-index}(\mathtt{X}_1), \ldots, \mathtt{index}(\mathtt{X}_{\mathtt{pt\_size}}).\} \cup \{\mathtt{index(i).} \mid i \in [0, \mathtt{max\_k}]\}$. The program has a single answer set which contains $\mathtt{vio}(\mathtt{X}_1, \ldots, \mathtt{X}_{\mathtt{pt\_size}})$ for each $\mathtt{X}_1, \ldots, \mathtt{X}_{\mathtt{pt\_size}}$ if and only if the corresponding parse tree is not a valid parse tree for G_{CFG}.

Let Π^2 be the program consisting of Π and the following extra rules:

- For each production rule $n \rightarrow n_1 \ldots n_k\ P \in G_{PR}$ with id pr_{id}, each $m \in [1, d]$, each $T = [t_1, \ldots, t_m]$ such that $\forall i \in [1, m] : t_i \in [1, \mathtt{max\_k}]$, and each rule $R \in P$: the rule constructed by appending $p(X_1, \ldots, X_{pt\_size})$ to the body of $R@[X_1, \ldots, X_{pt\_size}] + +T$, and replacing the head of any constraints with $\mathtt{vio}(\mathtt{X}_1, \ldots, \mathtt{X}_{\mathtt{pt\_size}})$.
- The rule:
  $\mathtt{non\_empty:-}$
  $\quad \mathtt{p}(\mathtt{X}_1, \ldots, \mathtt{X}_{\mathtt{pt\_size}}),$
  $\quad \mathtt{not\ vio}(\mathtt{X}_1, \ldots, \mathtt{X}_{\mathtt{pt\_size}}).$

The resulting program is stratified, and bravely entails $\mathtt{non\_empty}$ if and only if $\mathcal{L}^d(G)$ is non-empty – there must be at least one parse tree that is both valid and whose resulting ASP program is satisfiable. Thus, as the program is polynomial in the size of G, we have shown a polynomial reduction from first order bounded-ASG-satisfiability to an EXP-complete problem. Hence, stratified first order bounded-ASG-satisfiability is a member of EXP.

□

Corollary 5.

- *First-order Horn bounded-ASG-satisfiability is a member of EXP*
- *First-order Horn bounded-ASG-membership is a member of EXP*
- *First-order stratified bounded-ASG-membership is a member of EXP*

Corollary 6.

- *First-order Horn bounded-ASG-satisfiability is EXP-complete*
- *First-order stratified bounded-ASG-satisfiability is EXP-complete*
- *First-order Horn bounded-ASG-membership is EXP-complete*
- *First-order stratified bounded-ASG-membership is EXP-complete*

Theorem 8. *Deciding first order unstratified bounded-ASG-satisfiability is a member of $NEXP$.*

Proof. Assume that the depth of parse trees is bounded by some constant d.

We prove the theorem by showing that an ASG G can be mapped to an ASP program P which is satisfiable if and only if $\mathcal{L}^d(G) \neq \emptyset$.

Let G be a first order unstratified ASG. Let Π be the following ASP program:

- $\mathtt{node(G_S)@[]}$.
- For each $D \in [1,..,d]$, for each $X_1, X_2, \ldots, X_D \in [1, \mathtt{max\_k}]$:
  - For each $i \in [1, k]$, for each production rule $PR = \mathtt{n} \rightarrow \mathtt{n_1} \ldots \mathtt{n_k}\ P$:
    - The rule:
      $\mathtt{node(n_i)@[X_1, X_2, \ldots X_{D-1}, i]}$:-
      $\mathtt{pr(PR_{ID})@[X_1, X_2, \ldots X_{D-1}]}$.
    - For each rule $\mathtt{h@a_0}$:- $b_1@a_1, \ldots, b_i@a_1$, not $b_{i+1}@a_{i+1}, \ldots,$ not $b_j@a_j$ in P:
      $\mathtt{h@[X_1, X_2, \ldots, X_D, a_0]}$:-
      $\mathtt{b_1@[X_1, X_2, \ldots, X_D, a_1]}$,
      $\ldots$,
      $\mathtt{b_i@[X_1, X_2, \ldots, X_D, a_i]}$,
      not $\mathtt{b_{i+1}@[X_1, X_2, \ldots, X_D, a_{i+1}]}$,
      $\ldots$,
      not $\mathtt{b_j@[X_1, X_2, \ldots, X_D, a_j]}$.
  - Let $PR^1, \ldots, PR^m$ be the set of production rules in G_{PR} for n.
    For each node $n \in G_N$, for each $i \in [1, m]$, the rule:
    $\mathtt{pr(PR^i_{ID})@[X_1, X_2, \ldots, X_D]}$:-
    $\mathtt{node(n)@[X_1, X_2, \ldots, X_D]}$
    not $\mathtt{pr(PR^1_{ID})@[X_1, X_2, \ldots, X_D]}$,
    $\ldots$
    not $\mathtt{pr(PR^{i-1}_{ID})@[X_1, X_2, \ldots, X_D]}$,
    not $\mathtt{pr(PR^{i+1}_{ID})@[X_1, X_2, \ldots, X_D]}$,
    $\ldots$
    not $\mathtt{pr(PR^m_{ID})@[X_1, X_2, \ldots, X_D]}$.

Π is satisfiable if and only if $\mathcal{L}^d(G) \neq \emptyset$. Hence, deciding whether $\mathcal{L}^d(G) \neq \emptyset$ reduces to the NEXP problem of deciding the satisfiability of a first order ASP program consisting of normal rules and constraints. Hence, first order unstratified bounded-ASG-satisfiability is a member of NEXP.

□

Corollary 7.

- *First-order unstratified bounded-ASG-membership is a member of $NEXP$*

Corollary 8.

- *First-order unstratified bounded-ASG-satisfiability is $NEXP$-complete*
- *First-order unstratified bounded-ASG-membership is $NEXP$-complete*

Proofs for the complexity of learning ASGs

Theorem 9. *Propositional Horn $\mathcal{F}$ bounded-verification is DP-hard.*

Proof. Let D be a decision problem in DP. There is a pair of decision problems D_1 and D_2 such that D_1 is in NP and D_2 is in $coNP$. There is a mapping from D_1 to deciding whether a set of propositional clauses C_1 is satisfiable and from D_2 to deciding whether a set of propositional clauses C_2 is unsatisfiable.

Let $V_1 = \{v^1_1, \ldots, v^1_n\}$ be the set of atoms in C_1 and $V_2 = \{v^2_2, \ldots, v^2_m\}$ be the set of atoms in C_2. For any clause $c \in C$, $constraint(c)$ represents an annotated constraint form of c. For example, $v^1_1 \vee \neg v^1_2 \vee \neg v^1_3$ is represented as $\mathtt{:\text{-}\ not\_v^1@1, v^1@2, v^1@3}$.

Consider the ASG learning task $T = \langle G, \emptyset, \{\text{“}\mathtt{pos}''\}, \{\text{“}\mathtt{neg}''\}\rangle$, where G is the following propositional Horn ASG:

```
start → pos a_1 ... a_n {constraint(c)|c ∈ C_1}
start → neg b_1 ... b_m {constraint(c)|c ∈ C_2}
% for each i ∈ [1,n]
a_i → {v^1.}
a_i → {not_v^1.}
% for each i ∈ [1,m]
b_i → {v^2.}
b_i → {not_v^2.}
```

Note that C_1 is satisfiable if and only if there is an interpretation I of the atoms in V_1 such that $\{\mathtt{v@i.}|v_i \in I\} \cup \{\mathtt{not\_v@i.}|v_i \notin I\}\ \{constraint(c)|c \in C_1\}$ is satisfiable. This is the case if and only if there is an interpretation I of the atoms in V_1 such that I is a model of C_1. The parse trees of G for the string “pos” generate the full set of interpretations of the atoms in V_1, and hence there is a parse tree PT of G for “pos” such that $G[PT]$ is satisfiable if and only if C_1 is satisfiable. Similarly, there is a parse tree PT of G for “neg” such that $G[PT]$ is satisfiable if and only if C_2 is satisfiable.

Hence, the hypothesis $H = \emptyset$ is a solution of the learning task T if and only if the decision problem D returns true. Hence, any decision problem in DP can be polynomially reduced to bounded-verification. Hence, bounded-verification is DP-hard.

□

Theorem 10. *Propositional unstratified bounded-verification is a member of DP.*

Proof. Assume that the depth of parse trees is bounded by some constant d.

Checking whether H is a solution of a given learning task $T = \langle G, S_M, E^+, E^-\rangle$ at depth d corresponds to checking that for each positive example $s \in E^+$, $s \in \mathcal{L}^d(G)$ and for each negative example $s \in E^-$, $s \notin \mathcal{L}^d(G)$. As propositional unstratified bounded-ASG-membership is in NP, this means that there is a pair of sets of decision problems (each in NP) $D^+ = \{D^+_1, \ldots D^+_{|E^+|}\}$ and $D^- = \{D^-_1, \ldots D^-_{|E^-|}\}$ such that $H \in ILP^d_{ASG}(T)$ iff each problem in D^+ returns yes and each problem in D^- returns no.

Each decision problem D^i_j can be mapped to a set of propositional clauses C^i_j such that C^i_j is satisfiable iff D^i_j returns yes. Without loss of generality, we can assume that the atoms used in each set of clauses are disjoint. Hence, $H \in ILP^d_{ASG}$ iff $C^+_1 \cup \ldots \cup C^+_{|E^+|}$ is satisfiable and for each $i \in [1, |E^-|]$, C^-_i is unsatisfiable. This is the case if and only if $C^+_1 \cup \ldots \cup C^+_{|E^+|}$ is satisfiable and $\{v_1 \vee \ldots \vee v_{|E^-|}\} \cup \{c \vee \neg v_i | i \in [1, |E^-|], c \in C^-_i\}$ is unsatisfiable (where the v_i's are new atoms). Hence, deciding bounded-verification can be reduced to deciding one problem in NP and one problem in $coNP$. Hence, bounded-verification is a member of DP.

□

Corollary 9.

- *Propositional Horn bounded-verification is DP-complete.*
- *Propositional stratified bounded-verification is DP-complete.*
- *Propositional unstratified bounded-verification is DP-complete.*

Theorem 11. *Propositional Horn bounded-satisfiability is Σ^P_2-hard.*

Proof. Assume that the depth of parse trees is bounded by some constant d.

We prove this by reducing the Σ^P_2-complete problem of deciding whether $\Phi \in QBF_{2,\exists}$, where $\Phi = \exists x_1, \ldots, \exists x_m, \forall x_{m+1}, \ldots, \forall x_n E$, where E is a disjunction $C_1 \vee \ldots \vee C_k$ of conjunctions of length 3 over the atoms (or negations of atoms) in $\{x_1, \ldots, x_m, x_{m+1}, \ldots, x_n\}$.

Let $constraint(C_j)$ be a denial representation of C_j using annotations. So, for example, $x_1 \wedge \neg x_3 \wedge x_5$ is represented as `:- v@1, not_v@3, v@5.`

Consider the ASG learning task $T = \langle G, S_M, \{\text{“pos”}\}, \{\text{“neg”}\}\rangle$, where G is the following propositional Horn ASG:

```
1 : start → x_1 ... x_m pos {}
2 : start → x_1 ... x_n neg {constraint(C_j) | j ∈ [1, k]}

% for each i ∈ [1, m]
a_i : start → x_i neg {
  :- v@1.
  :- not_v@1.
}
b_i : x_i → {
  :- v, not_v.
}

% for each i ∈ [m + 1, n]
c_i : x_i → {v.}
d_i : x_i → {not_v.}
```

The hypothesis space of the task, $S_M = \{\langle \texttt{v.}, i\rangle, \langle \texttt{not\_v.}, b_i\rangle | b_i \in [1, m]\}$, means that each hypothesis represents an assignment to the existential variables in Φ. Note that the a_i rules mean that in order to cover the negative example, every inductive solution must contain at least one of the facts `v` or `not_v` in each production rule b_i. The constraint in each production rule b_i means that none of the b_i production rules can contain both `v` and `not_v`. Hence the only possible inductive solutions contain exactly one of $\langle \texttt{v.}, i\rangle$ or $\langle \texttt{not\_v.}, b_i\rangle$ for each $i \in [1, m]$.

Let θ be an assignment to $\{x_1, \ldots, x_m\}$ such that $\forall x_{m+1}, \ldots, \forall x_n E$. Then the hypothesis corresponding to θ cannot cover the negative example with the second production rule (as every assignment to $\{x_{m+1}, \ldots, x_n\}$ must violate at least one of the constraints in the ASP of the second production rule). Conversely let θ be an assignment to $\{x_1, \ldots, x_m\}$ such that $\neg\forall x_{m+1}, \ldots, \forall x_n E$. Then the hypothesis corresponding to θ covers the negative example with the second production rule (as there is at least one assignment to $\{x_{m+1}, \ldots, x_n\}$ which does not satisfy any of the conjunctions in E, and thus does not violate any of the constraints in the ASP of the second production rule).

Hence, T is satisfiable at depth d (for any $d \geq 1$) if and only if Φ is valid (i.e. iff $\Phi \in QBF_{2,\exists}$). Hence, as deciding whether $\Phi \in QBF_{2,\exists}$ is Σ^P_2-complete, deciding bounded-satisfiability is Σ^P_2-hard.

□

Theorem 12. *Propositional unstratified bounded-satisfiability is a member of Σ^P_2.*

Proof. To prove the theorem, we show that a non-deterministic Turing Machine with access to an NP oracle could check satisfiability of any ASG learning task $T = \langle G, S_M, E^+, E^-\rangle$ in polynomial time.

A non-deterministic Turing Machine can have $|S_M|$ choices to make (corresponding to selecting each pair in the hypothesis space as part of the hypothesis). As propositional unstratified bounded-verification is in DP (by Theorem 10), this hypothesis can then be verified in polynomial time using an NP oracle, with two queries, answering yes if and only if the first query returns yes and the second query returns no.

Such a Turing Machine would terminate answering yes if and only if the task is satisfiable (as there is a path through the Turing Machine which answers yes if and only if there is an hypothesis in S_M which is an inductive solution of the task).

Hence, deciding propositional unstratified bounded-satisfiability is in Σ^P_2. □

Corollary 10.

- *Propositional Horn bounded-satisfiability is Σ^P_2-complete.*
- *Propositional stratified bounded-satisfiability is Σ^P_2-complete.*
- *Propositional unstratified bounded-satisfiability is Σ^P_2-complete.*