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 Pompeu 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 as 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 annotations 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 notions and terminologies used throughout the paper. Given atoms $\texttt{h}$, $\texttt{b}_1, \ldots \texttt{b}_n$, $\texttt{c}_1, \ldots, \texttt{c}_m$, a *normal rule* is `h:- b`$_1, \ldots,$`b`$_n$`, not c`$_1, \ldots,$ `not c`$_m$, where `h` is the *head* and `b`$_1, \ldots,$`b`$_n$`, not c`$_1, \ldots,$ `not c`$_m$ (collectively) is the *body* of the rule, and "`not`" represents negation as failure. Rules of the form `:-b`$_1, \ldots,$`b`$_n$, `not c`$_1, \ldots,$ `not 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 $\texttt{b}_i$'s in the above rule) in the body of the rule. In this paper, we assume an ASP 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 (Herbrand) interpretations of P.

Given a program P and an 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. 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 by A if the head is in A or the body is not satisfied by A. A ground constraint is satisfied when the body is not satisfied. A constraint therefore has the effect of eliminating all answer sets of P that satisfy the body of the constraint. A program P is *stratified* if it can be partitioned into (disjoint) *strata* $P_1, \ldots, P_n$ such that for each predicate p, if p occurs positively (resp. negatively) in the body of a rule in P_i then every rule in P with p in the head is in $P_1, \ldots, P_i$ (resp. $P_1, \ldots, P_{i-1}$). The notation ++ denotes list concatenation (e.g. $[1, 2, 3]$++$[4, 5] = [1, 2, 3, 4, 5]$).

Answer Set Grammars

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

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, 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.

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

[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.

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

(a)

$trace$	$node$	$rule$
[]	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

(b)

Figure 1: A CFG for $\mathtt{a^ib^jc^k}$ and the parse tree for `ac`.

where some atoms have been *annotated* with a ground term. For instance, the annotated atom a(1)@2 represents the atom a(1) with the annotation 2. When computing the answer sets of an annotated program, annotated atoms are treated as ordinary atoms, where a@k, a@l and 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.

Note that a CFG can be represented as an ASG where all production rules are unannotated (i.e. the annotation of every production rules is empty). 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 presents an example ASG, and explains the intuition of an ASGs semantics, which is then formalised for general ASGs in Definitions 5 and 6.

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 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.

A parse tree PT of an ASG G for a string s is defined similarly to the context-free case (with the difference that production rules are annotated production rules). 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 5. *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* a@i *with the atom* a@(t++[i]) *and all unannotated atoms* a *with the atom* a@t.

Definition 6. *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 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 size(1)@[1] is guaranteed to be true in all answer sets and 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: {size(0)@[1, 2], size(1)@[1], size(0)@[2, 2], size(1)@[2], size(0)@[3, 2], size(1)@[3]} meaning that 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.

$trace$	$node$	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 CFG), and two sets of strings, E^+ and E^-, and learns an ASG $G^{'}$ such that $G^{'}_{CF}$ is the same as G_{CF}, and every string in E^+ (resp. E^-) is accepted (resp. rejected) by $G^{'}$. 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 a *hypothesis space*, which defines the set of rules that can form possible learned outcomes, referred to as possible *hypotheses*. For convenience, hypothesis spaces are often characterised by a set of *mode declarations* (Muggleton 1995). 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 ILASP mode declarations given in (Law, Russo, and Broda 2014), allowing declarations specifying which atoms can appear in the head and body of rules in the hypothesis space (#modeh's and #modeb's, respectively) whose arguments may be placeholders for variables and constants of given types (denoted `var(t)` and `const(t)`, respectively). For instance the mode declaration `#modeb(p(var(t)), [1, 2, 3])` means that `p(X)`, for any variable `X` of type[3] `t`, may occur in the body of rules which are used to annotate 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 (a subset of the hypothesis space) 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 with 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. For instance, if we were aiming to learn the grammar $a^n b^n c^n$, we might give examples like "`abc`" and "`aabbcc`", as strings that should be accepted, and "`aabcc`" as a string that should not be accepted. Definition 7 formalises the ASG learning task.

Definition 7. *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.*

[3]The only restriction imposed by types is that no variable can appear twice in the same rule with different types. For example, given mode declarations $\{$`#modeh(p(var(t1)), [1])`, `#modeb(q(var(t2)), [1])`$\}$, we cannot add $p(X)$:- $q(X)$ to the annotation of production rule 1.

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

Table 1: Complexity classes for propositional and first-order ASGs (each element of the table is the class for which BAM and BAS are both complete).

Note that the existing grammar of an ASG learning task may or may not contain annotations on some or all of the production rules. The annotations that are present constitute what is known as *background knowledge* in conventional ILP, as they encode knowledge that is already known. In the extreme, this knowledge can all be learned (this is the case when the existing grammar is a pure CFG), but as shown in our evaluation, when semantic conditions are known, including them in the existing grammar can reduce the number of examples and time needed to learn the correct grammar.

Computational Complexity

We consider two classes of decision problem. The first addresses the complexity of existing ASGs, and the second the complexity of the learning task. Our results are presented in terms of a bound on the depth of parse trees, i.e. there is a fixed bound d on the depth of parse trees in this section.

Grammar Decision Problems

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

The results are summarised in Table 1. 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.

Propositional results. In proving the complexity results we make use of two chains of reductions: (1) Horn BAS to Horn BAM to stratified BAM to unstratified BAM to unstratified BAS; and (2) Horn BAS to stratified BAS to unstratified BAS. Note that all but the first and last reductions of chain (1) hold trivially, so it remains to show these two non-trivial cases, which are special cases of Theorems 1 and 2.

Theorem 1. *For any fragment $\mathcal{F}$ of ASP, $\mathcal{F}$ BAS reduces to $\mathcal{F}$ BAM.*

Proof. 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). "" $\in \mathcal{L}^d(G')$ iff $\mathcal{L}^d(G) \neq \emptyset$. As G' is an $\mathcal{F}$ ASG, $\mathcal{F}$ BAS reduces to $\mathcal{F}$ BAM. □

Theorem 2. *For any fragment $\mathcal{F}$ of ASP that contains constraints and negation as failure, $\mathcal{F}$ BAM reduces to $\mathcal{F}$ BAS.*

The proof of Theorem 2 (omitted[4]) shows that given a string s and an ASG G, we can extend G using stratified normal rules and constraints to yield a grammar G' that is satisfiable at depth d iff s is in $\mathcal{L}^d(G')$. Due to the chains of reductions, to show the results in the first row of Table 1 (for both decision problems), it remains to show that Horn BAS is NP-hard and unstratified BAS is in NP.

Theorem 3. *Propositional Horn BAS is NP-hard.*

Proof. We reduce satisfiability of a set of propositional clauses C to propositional Horn BAS. Let $V = \{v_1, \ldots, v_n\}$ be the set of atoms in C. For any clause $c \in C$, $constraint(c)$ represents an annotated constraint form of c (e.g. $v_1 \vee \neg v_2 \vee \neg v_3$ is represented as $\texttt{:-not\_v@1}, \texttt{v@2}, \texttt{v@3}.$). Consider the ASG G consisting of the production rule $\texttt{start} \rightarrow \texttt{a}_1 \ldots \texttt{a}_n \, \{constraint(c) | c \in C\}$ and for each $i \in [1, n]$ the production rules $\texttt{a}_i \rightarrow \{\texttt{v.}\}$ and $\texttt{a}_i \rightarrow \{\texttt{not\_v.}\}$.

Note that C is satisfiable iff there is an interpretation $I \subseteq V$ such that $\{\texttt{v@i.} | v_i \in I\} \cup \{\texttt{not\_v@i.} | v_i \notin I\} \cup \{constraint(c) | c \in C\}$ is satisfiable. There is one parse tree of G corresponding to each $I \subseteq V$; hence there is a parse tree PT of G st $G[PT]$ is satisfiable iff C is satisfiable. $\square$

Theorem 4. *Propositional unstratified BAS is in NP.*

The proof of Theorem 4 (omitted), reduces propositional unstratified BAS to the satisfiability of an ASP program consisting of propositional normal rules and constraints, which is in NP (Leone et al. 2006).

First order results. Theorems 1 and 2 show two chains of reductions in the first-order case: (1) Horn BAS to Horn BAM to stratified BAM to stratified BAS; and (2) unstratified BAS to unstratified BAM to unstratified BAS. Thus, to prove the remaining results in Table 1, it suffices to show that Horn BAS is EXP-hard, stratified BAS is in EXP and unstratified BAS is both in NEXP and NEXP-hard. We first show the two hardness results.

Theorem 5. *First order Horn BAS is EXP-hard.*

Proof. We reduce an arbitrary EXP-hard problem D to first order Horn BAS. As EXP is closed under complement, $\bar{D}$ (the complement of D) must also be in EXP. Hence, as cautious entailment of (function-free) first order Horn programs is EXP-complete (Leone et al. 2006), there is a first order Horn program P and a ground atom $\texttt{a}$ such that $P \models_c \texttt{a}$ iff $\bar{D}$ returns yes. Hence, $P \cup \{\texttt{:-a.}\}$ is satisfiable iff $\bar{D}$ returns no, which is the case iff D returns yes. Consider the ASG G with a single production rule $start \rightarrow P \cup \{\texttt{:-a.}\}$. Note that $\mathcal{L}^d(G)$ is non-empty iff $P \cup \{\texttt{:-a.}\}$ is satisfiable, which is the case iff D returns yes. $\square$

Theorem 6. *First order unstratified BAS is NEXP-hard.*

[4]All proofs omitted from the paper can be found at https://www.doc.ic.ac.uk/~ml1909/AAAI19_proofs.pdf.

	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.

Proof. Let P be an arbitrary (function-free) first order unstratified ASP program and $\texttt{a}$ be an atom. Deciding whether $P \models_b \texttt{a}$ is NEXP-hard (Leone et al. 2006). Hence, it suffices to reduce deciding whether $P \models_b \texttt{a}$ to deciding the satisfiability of a first order unstratified-ASG G. Consider the ASG G that contains a single production rule $start \rightarrow P \cup \{\texttt{:- not a.}\}$. $\mathcal{L}^d(G)$ is non-empty iff $P \cup \{\texttt{:- not a.}\}$ is satisfiable, which is the case iff $P \models_b \texttt{a}$. $\square$

It remains to show that first order stratified BAS is in EXP and first order unstratified BAS is in NEXP.

Theorem 7. *First order stratified BAS is in EXP.*

The proof of Theorem 7 (omitted) reduces the decision problem to deciding the satisfiability of a stratified first-order (function free) ASP program, which is in EXP (Leone et al. 2006).

Theorem 8. *First order unstratified BAS in NEXP.*

The proof of Theorem 8 (omitted) reduces the decision problem to deciding the satisfiability of an unstratified first-order (function free) ASP program, which is in NEXP (Leone et al. 2006).

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

The results in this section are all for propositional learning tasks[5]. We consider the following decision problems:

- *Bounded-verification* (BV) is the problem of deciding if a given hypothesis is a solution of a given learning task.
- *Bounded-task-satisfiability* (BTS) is the problem of deciding whether a given learning task has any solutions.

Complexity results for learning are summarised in Table 2. As there are trivial reductions from Horn to stratified to unstratified BV, the following two theorems suffice to show the completeness results for BV.

Theorem 9. *Horn BV is DP-hard.*

Theorem 10. *Unstratified BV is in DP.*

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

Problems in DP can be reduced to a pair of decision problems, one in NP and one in co-NP, such that the original decision problem returns yes iff both of the new decision problems return yes. The (omitted) proof of Theorem 9 reduces the DP-complete problem of deciding whether one set of propositional clauses is satisfiable and another is unsatisfiable to Horn BV. Similarly, we prove the unstratified case is in DP by reducing it to this DP-complete problem.

Again due to the trivial reductions from Horn to stratified to unstratified BTS, the next two theorems suffice to show the remaining completeness results in Table 2.

Theorem 11. *Horn BTS is Σ_2^P-hard.*

The (omitted) proof reduces deciding the validity of a restricted form of quantified boolean formula (a known Σ_2^P-complete problem) to Horn BTS. It remains to show that unstratified BTS is in Σ_2^P.

Theorem 12. *Unstratified BTS is in Σ_2^P.*

Proof. We show that a non-deterministic Turing Machine (NDTM) 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. An NDTM can have $|S_M|$ choices to make (corresponding to selecting each pair in the hypothesis space as part of the hypothesis). As unstratified BV is in DP (by Theorem 10), this hypothesis can then be verified in polynomial time using an NP oracle, with two queries, answering yes iff the first query returns yes and the second query returns no. Such an NDTM terminates answering yes iff the task is satisfiable (as there is a path through the Turing Machine which answers yes iff there is a hypothesis in S_M which is an inductive solution of the task). $\square$

A key observation to make is that the complexities for BV and BTS under propositional ASP are identical to the complexities for verification and satisfiability of $ILP_{LAS}^{context}$ 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 $ILP_{LAS}^{context}$ task, and use the existing ILASP (Law, Russo, and Broda 2015a) system to solve the ASG learning task. We show in the next section how this can be done. Furthermore, we show that, with some restrictions, stratified BTS is in NP. The complexity of the learning task for the function-free first order case is 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 8. *((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 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 9. *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 9 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 $\mathtt{pr(PR_{id}, X)}$ to the body of the rule.[6] The mapping in Definition 10 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 10. *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 13 shows that we can use the mapping in Definition 10 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 13 is omitted.

Theorem 13. *Let T be an ASG learning task. $ILP_{ASG}^d(T) = \{H^{ASG} \mid H \in ILP_{LAS}^{context}(LAS(T, d))\}$*

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

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

Learning Stratified ASGs

Deciding satisfiability for a general $ILP_{LAS}^{context}$ task is Σ_2^P complete (in the propositional case); however, if there are no negative examples then the complexity is only NP-complete. ILASP tasks with no negative examples tend to run faster in ILASP than equivalent tasks with negative examples, 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 in Definition 10 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 (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 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 propositional stratified BTS 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 $\mathtt{w_1 w_2}$ where $\mathtt{w_1}$ and $\mathtt{w_2}$ are both non-empty strings of `a`'s and `b`'s.
- The language $\mathtt{a^n b^n c^m}$, $\mathtt{n} \leq \mathtt{m}$: the input language was a CFG corresponding to the language $\mathtt{a^i b^j c^k}$ and the task was to learn annotations expressing that $i = j \leq k$.
- The language $\mathtt{a^n b^n c^n}$. We considered four different learning tasks for this language:
  - **A**: the input language was a CFG representing $\mathtt{a^i b^j c^k}$. The task was to learn in the annotations that $i = j = k$.
  - **B**: the input language was a CFG representing $\mathtt{a^n b^n c^m}$. The task was to learn in the annotations that $n = m$.
  - **C**: the input language was an ASG G such that G_{CF} represents $\mathtt{a^i b^j c^k}$, 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 $\mathtt{a^n b^n c^n}$ **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 be annotated.
  - **D**: the input language was a CFG representing the language $(\mathtt{a}|\mathtt{b}|\mathtt{c})*$. 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.
- The language $\mathtt{a^n b^m c^n d^m}$: the input language was a CFG corresponding to the language $\mathtt{a^i b^j c^k d^l}$ 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. $\{\mathtt{-5, 4, 1}\}$). 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 (Garey and Johnson 1979).
- The `graph-col` language: the input language was an ASG representing the language $\mathtt{a^n b^m}$ where each `a` is an integer in $[0, 3]$, representing the nodes of a graph, and each `b` is a pair of integers representing a directed edge (only those integers used as nodes could be used in edges). To avoid equivalent strings we enforced that the nodes and edges were ordered lexicographically. The task was to learn to restrict the language to 3-colourable graphs. We tested this with two ASGs representing the same input language:
  - **A**: the input ASG had no "duplicate" production rules, meaning that the only way to express 3-colourability was to learn an unstratified ASG using a loop to simulate the choice of colour for each node.
  - **B**: the input ASG had 3 "duplicate" production rules for node, meaning that the learned ASG could express the choice of colour by using one production rule for each choice of colour. In this case, we restricted the hypothesis space to only Horn clauses.

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 counterexample (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 counterexample existed, the shortest such example was added to the learning task at the next iteration. The iterations, and therefore the learning, terminated when no such counterexamples existed. Note that as checking whether two CFGs are equivalent is undecidable, this is

[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	E^+	E^-	S_M
copy	174.6s	404.8s	3	10	55
$\mathtt{a^n b^n c^m}$, $\mathtt{n} \leq \mathtt{m}$	1.0s	6.4s	2	8	45
$\mathtt{a^n b^n c^n}$ **A**	1.1s	5.1s	1	7	45
$\mathtt{a^n b^n c^n}$ **B**	0.3s	1.3s	1	3	28
$\mathtt{a^n b^n c^n}$ **C**	0.5s	1.4s	1	2	45
$\mathtt{a^n b^n c^n}$ **D**	1004.0s	13314.9s	1	45	66
$\mathtt{a^n b^m c^n d^m}$	1.8s	12.1s	2	10	64
subset-sum	336.9s	657.5s	3	9	479
graph-col **A**	90.1s	256.9s	4	4	297
graph-col **B**	11.5s	23.8s	2	2	117

Table 3: A summary of the results of our evaluation. *Final Time* and *Total Time* show the learning time (on an Ubuntu 14.04 desktop machine with a 3.4 GHz Intel® Core™ i7-3770 processor and 16GB RAM) taken in the final iteration and the total learning time, respectively. $|E^+|$ and $|E^-|$ show the number of positive and negative examples needed to learn the target language in each case. $|S_M|$ is the number of rules in the hypothesis space. Note that a hypothesis space of n rules leads to 2^n potential hypotheses.

also the case for ASGs. Therefore, when computing counterexamples we put an upper bound on the depth of parse trees to make the computation feasible. In each case the bound was high enough that the final grammar was equivalent to the target grammar (this was manually checked).

Table 3 summarises the results of our evaluation. The four $\mathtt{a^n b^n c^n}$ 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). The graph colouring experiment shows that by restricting a learning task to the Horn case, fewer examples (and significantly less time) may be needed. However, it should be noted that constructing the input language for the graph-col **B** task required prior knowledge of how many duplicate production rules were needed and that the level of choice is therefore not learned, but is given as background knowledge. On the other hand, the graph-col **A** task may have taken longer and required more examples, but it did not require such prior knowledge. This is an indication that in real settings if the required level of choice is known a priori then inputting it in the initial language will save time, but in cases where it is unknown then unstratified ASGs are necessary.

(Nakamura and Imada 2011) evaluate their LIG learner on each of the above languages other than subset-sum, graph-col and the variations of $\mathtt{a^n b^n c^n}$. LIGs make no distinction between 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 $\mathtt{a^n b^n c^n}$. 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 $\mathtt{a^n b^n c^n}$ takes significantly longer in our case (1004s compared to 18s) 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 or graph-col languages.

Our evaluation assumed an oracle giving counterexamples. This is not uncommon in grammar induction (e.g. (Angluin 1987; Sakakibara 1990)). Induction from randomly observed examples is more challenging and in general we cannot be sure that the learned hypothesis is correct (it may only be correct on the seen examples). In practice, this means that the learner should be deployed incrementally, with the option to relearn on seeing a new counterexample. The hypothesis spaces in our evaluation were constructed from the target hypotheses, allowing predicates to be used in the head/body if they occurred there in the target hypothesis. In practice, when the target hypotheses are unknown, larger hypothesis spaces must be used, which may mean that more examples are required to learn the correct grammar.

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 CFG. 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 $a^n b^n c^n$, 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 procedural, 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)@1:- safe(V).
  safe(V):- safe(V)@3.  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.

Note that although annotated head atoms allow more declarative representations, they do not affect the complexity results presented earlier in the paper – if we were to consider a new class of "strictly stratified" ASGs, where annotations could only appear in the body, the hardness proofs would still apply as they do not make use of annotated head atoms.

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 2015b)). 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 CFGs in the form of DCGs. There is also some work on learning CSGs (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 CFG. 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 CFG, expressed in ASP. Grammars have been previously used to encode more typical AI problems such as scheduling constraints in (Kadioglu and Sellmann 2008; Drescher and Walsh 2011). By extending CFGs with ASP we cover a larger class of problems through the added context-sensitive constraints. Our second contribution is more foundational, expanding the results on automatic generation of finite state controllers (Bonet, Palacios, and Geffner 2010; Hu and De Giacomo 2013), and the more recent results on automatic generation of hierarchical controllers and CFGs (Segovia-Aguas, Jiménez, and Jonsson 2016), by providing a better understanding of the complexity of learning more complex automata, e.g. the ones captured by some CSGs.

In the general case, the complexity of 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.

Just as it is interesting to consider the implications of using different classes of ASP program to annotate production rules, it is also possible to consider using other formalisms in the production rule annotations, such as Constraint Satisfaction Problems (CSP) or Satisfiability Modulo Theories (SMT). It may be that other formalisms allow more expressive languages to be captured or that the complexity of decision problems for grammars annotated using other formalisms is lower than for ASGs. Our choice of ASP in this paper allowed us to delegate the learning of the annotations to an existing ASP learner.

The ASG learning task presented in this paper takes an initial grammar as input, which allows known concepts to be given as background knowledge, aiding the learning process. 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.

Acknowledgements

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-16-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon. It was also partially supported by the Spanish Ministry of Economy and Competitiveness under Grant Numbers TIN-2016-81032-P & MDM-2015-0502.

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. *Information and computation* 75(2):87–106.

Bonet, B.; Palacios, H.; and Geffner, H. 2010. Automatic derivation of finite-state machines for behavior control. In *AAAI*.

Corapi, D.; Russo, A.; and Lupu, E. 2010. Inductive logic programming as abductive search. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 7. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Drescher, C., and Walsh, T. 2011. Modelling grammar constraints with answer set programming. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 11. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

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.

Garey, M., and Johnson, D. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, CA.

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.

Hu, Y., and De Giacomo, G. 2013. A generic technique for synthesizing bounded finite-state controllers. In *ICAPS*.

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.

Kadioglu, S., and Sellmann, M. 2008. Efficient context-free grammar constraints. In *AAAI*, 310–316.

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.

Leone, N.; Pfeifer, G.; Faber, W.; Eiter, T.; Gottlob, G.; Perri, S.; and Scarcello, F. 2006. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL)* 7(3):499–562.

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. *Machine Learning* 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, 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.

Sakakibara, Y. 1990. Learning context-free grammars from structural data in polynomial time. *Theoretical Computer Science* 76(2-3):223–242.

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

Segovia-Aguas, J.; Jiménez, S.; and Jonsson, A. 2016. Hierarchical finite state controllers for generalized planning. In *International Joint Conference on Artificial Intelligence*.

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.