Composition of biases in Inductive Logic Programming

Frédéric Moal and Christel Vrain

LIFO
Université d'Orléans,
rue Léonard de Vinci, BP 6759,
45067 Orléans cedex 02, France
{moal,cv}@lifo.univ-orleans.fr

Abstract. Studying biases is an important topic in Inductive Logic Programming, and many works have dealt with the problem of defining interesting formalisms for expressing biases. The expected properties for such representation languages are usually declarativity, universality, easy shift of biases and learning optimization. In this paper, we propose a new formalism which was designed when attempting to apply learning techniques to Data Mining applications. In that context, an important property is incrementality, i.e., how to modify a set of already coded biases, without re-coding the entire set of biases. The formalism we propose is based on Regular Tree Grammars: they seem interesting in the purpose of incrementality, because they allow the computation of sets operations: intersection, union and complement. Nevertheless, such a formalism is not sufficient for expressing all typical ILP biases. Therefore, we propose to refine this framework with a formalism which is still declarative but more expressive than Tree Grammars: biases are expressed by means of constraints and a solver enables to check the membership of a hypothesis to the search space.

1 Introduction

Learning is often viewed as the problem of a finding "good" hypothesis in a search space, usually ordered by a generality relation. In Inductive Logic Programming (**ILP**), the search space is composed of clauses, and exhaustively exploring such a space is intractable. Therefore, defining biases for reducing such a complexity is a topic which has already long been studied in ILP. In this paper, we focus on language biases which allow to reduce this hypotheses space. They are usually divided into two groups: semantic biases which are mainly restricted to the definitions of types and modes for predicate symbols and syntactic biases. This latter form of biases has attracted much more attention, and many works have tried to define representation languages for expressing biases. Four main approaches have been developed to represent biases [14]:

- formalisms closed to a logic programming representation, as in Clause Sets [1], or $\mathcal{D}$LAB [4];

– second-order (or scheme-) based formalisms, as in MOBAL [9], or in MILES-CTL [18] allowing to express schemata that can be instantiated by predicate symbols;
– grammar-based formalisms, as in Grendel [2];
– parametric approaches [15] based on the definition of parameters, such as the maximum number of variables in a clause, the maximum number of literals, the maximum depth of terms, ...

Each of these formalisms has its own advantages:

– declarativity for the two first formalisms;
– ability to express many classical biases, in the grammar-based formalisms;
– ability to design a way of searching the hypotheses space in relation with a generality order;
– easy shift of biases [15] for the parametric approaches: this enables to learn in a given search space, and iteratively to enlarge it until a solution to the learning task is obtained.

As far as we know, none of these frameworks satisfy all these advantages.

The work we present in this paper can be seen as a new framework for defining biases in ILP. It has been designed in the context of applying ILP techniques to Data Mining applications. In that case, applying ILP (and more generally Machine Learning) techniques is an iterative process, that can be roughly defined as follows:

1. definition of preliminary biases for the learning task,
2. application of Machine Learning techniques,
3. validation of the results,
4. when necessary, modification of the biases and iteration to Step 2.

When dealing with non parametric biases, the last step leads to a new problem: how to modify a set of biases already encoded in a given formalism, without re-coding the entire set of biases? In this paper, we stress on the importance of *incrementality* for biases languages. The ideas underlying our work is that changing biases can be seen as operations (intersection, union, difference) between sets of hypotheses. Therefore, a way to handle that problem of incrementality is to design a new biases language that allows to easily compute such sets operations.

Let us illustrate this principle on the classical Arch example [19], that will be explained in details in Section 3.2. For that learning task, we have initial knowledge $\mathcal{B}_1$ stating that an arch is composed of blocks defined by their shapes (parallelepiped, cube, pyramid, sphere, ...) and that blocks are linked by a relation $stable(X, Y)$ meaning that when put on Y, the object X remains stable. This defines a hypotheses space composed of the clauses defining the predicate *arch* and which bodies are built with the predicates defining the shapes and the predicate *stable*. Now, let us suppose that the user wants to specify that an arch is composed of at most five blocks. Let us call $\mathcal{B}_2$ this new bias. To code it, the user can either directly specify the whole set of biases, or specify only $\mathcal{B}_2$ and let the system automatically compute the language satisfying $\mathcal{B}_1$ and $\mathcal{B}_2$. Let us suppose now that the user wants to specify that the literal $stable(X, X)$ is not valid. He can define the set of hypotheses containing that literal, and then let the system automatically compute the whole set of hypotheses, using a complement operation.

We propose to use a formalism based on Regular Tree Grammars [6, 3], which is well-adapted to model clauses with functional symbols and which allows to compute intersection, union and difference over (infinite) tree languages. Nevertheless, the context-free nature of this formalism is restrictive, and is not expressive enough to model all the classical ILP biases. Therefore, we propose to refine this framework with constraint-based languages that allow by means of constraints to express context dependent restrictions.

This paper is organized as follows. In Section 2, we present Regular Tree Grammars, and our extension to Constraint Tree Grammars. In Section 3, intersection, union and difference of tree languages are defined, and the interest of these operations for biases specification is illustrated by an example. In Section 4, we propose different approaches to search an hypotheses space defined by means of a tree grammar. Section 5 is devoted to comparisons with usual biases formalisms in ILP. In Section 6, we propose some perspectives of that work.

2 A formalism based on Tree Grammars

Before presenting our formalism, let us recall some definitions and notations about terms.

Definition 1. *Given a set $\mathcal{F}$ of constructors, and given a set $\mathcal{X}$ of variables, the set of* **terms** *over $\mathcal{F}$ and $\mathcal{X}$, denoted by $T(\mathcal{F}, \mathcal{X})$, is the smallest set defined by:*

- $\mathcal{F}_0 \subseteq T(\mathcal{F}, \mathcal{X})$,
- $\mathcal{X} \subseteq T(\mathcal{F}, \mathcal{X})$,
- *if* $f \in \mathcal{F}_n$ *and* $t_1, \ldots, t_n \in T(\mathcal{F}, \mathcal{X})$, *then* $f(t_1, \ldots, t_n) \in T(\mathcal{F}, \mathcal{X})$,

where $\mathcal{F}_n$ denotes the set of constructors of arity n.

The set of **ground** terms in which no variable occurs is denoted by $T(\mathcal{F})$.
A term of $T(\mathcal{F}, \mathcal{X})$ can be identified to a tree with nodes labelled in $\mathcal{F} \cup \mathcal{X}$. Each node can be uniquely identified by an integer list, called its position and representing the path from the root of the tree to this node. These positions are defined as follows:

- the root node is identified by the empty list, denoted by ε.
- if u identifies a node in t_i, the same node in $f(t_1, ..., t_i, ..., t_n)$ is identified by $i.u$.

Let t be a term of $T(\mathcal{F}, \mathcal{X})$, we write:
$Pos(t)$ the set of all positions (occurrences) in t, $t|_u$ with $u \in Pos(t)$ the subterm of t at position u, and $t[u \leftarrow s]$ denotes the term obtained by replacing in t at position u the subterm $t|_u$ with the term s.

2.1 Regular Tree Grammar

Definition 2. *[6] A* **regular tree grammar** *G is a tuple $(A, N, \mathcal{F}, R)$ where:*

- *A is the axiom,*
- *N is a set of non-terminal symbols, with $A \in N$,*
- *$\mathcal{F}$ is a set of terminal symbols with fixed arities (and $\mathcal{F} \cap N = \emptyset$),*
- *R is a set of production rules: $X \to \alpha$ where $X \in N$ and $\alpha \in T(\mathcal{F} \cup N)$,*

Following standard notations, non-terminals are written in capital letters $(A, X, Y, \ldots)$, terminals are written in lower case letters $(f, c, \ldots)$. The arity of each terminal is declared with Prolog notation, e.g. $f/_2$ means that the arity of terminal f is 2. The symbol $|$ represents an alternative. For sake of clarity, infix notation is used for symbols with arity 2.

Example 1 presents a regular tree grammar, which defines an infinite language (because of the recursive rule $B \to L \wedge B$), corresponding to all clauses $A : -L_1, \ldots, L_n$, written with the predicates $p/2$ and $q/1$, and with two variables x, y.

The names of variables occurring in clauses are written in lower cases to avoid confusion with the non-terminals of the grammar. Let us notice that in this framework, the set of variables occurring in a clause is finite, since the set of terminal symbols are finite.

Example 1. An example of a regular tree grammar.

$$
\begin{array}{l}
G_1 = (\\
\quad C, \\
\quad \{C, H, B, L, A, T\}, \\
\quad \{:-/_2, \wedge/_2, \neg/_1, true/_0, p/_2, q/_1, x/_0, y/_0\}, \\
\quad \{ \\
\quad C \to H \ :- \ B \\
\quad H \to A \\
\quad B \to true \ | \ L \wedge B \\
\quad L \to A \ | \ \neg A \\
\quad A \to p\ (\ T,\ T\ ) \ | \ q\ (\ T\ ) \\
\quad T \to x \mid y \\
\quad \} \\
)
\end{array}
$$

The language defined by a regular tree grammar G and denoted by $L(G)$, is the set of all ground terms of $T(\mathcal{F})$ which can be reached by successive derivations starting from the axiom. Derivations are defined as follows:

Definition 3. *Given a regular tree grammar $G = (A, N, \mathcal{F}, R)$, the* **derivation relation** $\Rightarrow$ *associated to G is a binary relation over $T(\mathcal{F} \cup N)$ such that :*

$t \Rightarrow t'$ if there exists an occurrence $u \in Pos(t)$, a production rule $X \to \alpha \in R$ such that $t|_u = X$ and $t[u \leftarrow \alpha] = t'$.

Definition 4. *The* **language** *generated by a grammar G, and denoted by $L(G)$, is :*

$L(G) = \{t \in T(\mathcal{F}) \mid A \Rightarrow^+ t\}$, where $\Rightarrow^+$ denotes the transitive closure of $\Rightarrow$, and A the axiom of G.

Example 2. For instance, successive derivations for grammar G_1 are:
$C \Rightarrow H :- B \Rightarrow A :- B \Rightarrow p(T, T) :- B \Rightarrow \ldots \Rightarrow p(x, y) :- q(x)$

Using regular tree grammars to express language biases has interesting advantages, but the context-free nature of these grammars limits the expressiveness of this formalism, and its overall interest. Let us consider for instance a clause pattern like $p(X) : -q(X, \alpha), r(\alpha)$ where α can be instantiated by X or Y. The clause $p(X) : -q(X, X), r(Y)$ is in general not interesting, since the last literal is not connected to the head. This means that the choice for the instantiation of the second α depends on the first one, and so is not context free. Another example of a non context-free language is for defining the set of atoms $p(f^n(x), f^n(x))$; here n is clearly a context. This shows that this formalism must be extended to take into account contexts. In the field of compilation where Context Free Grammars have initially been formalized, the notion of *attribute grammars* [10, 7, 21] has been introduced to extend the syntactical definition of a Context Free Grammar with semantical data (for instance, variable typing declaration/use in C). In that case, attributes are associated to each non-terminal symbol, and semantic rules, written in a given imperative language, are associated to each production rule of the grammar. Nevertheless, using attribute grammars in our framework has two main weaknesses:

- Using an imperative language is opposed to the declarative expression of biases [16]. The exclusive nature (inherited/synthesized) of each attribute implies restrictions on dependencies over attributes, and then reinforces its operational semantic.
- The computation of an attribute and the possible detection of errors is often done on *complete* derivation trees. This point implies a two-steps process for generating valid trees, namely generation and test, which may be impossible if the underlying language is infinite. Some techniques have been developed to deal with this problem, as for instance in [17], but they can only be applied in restrictive cases.

The solution we propose is an intermediate one which enables to express the way of computing an attribute in a declarative formalism, but which also enables to stop the tree generation process as soon as the partial tree built is not valid. To achieve this, we have extended the notion of Regular Tree Grammars to Constraint Tree Grammars.

2.2 Constraint Tree Grammars

Definition 5. *A* **Constraint Tree Grammar** *is a tuple* $G = (A, N, \mathcal{F}, \Sigma, R)$ *where :*

- *A is the axiom,*
- *N is a set of non-terminal symbols* $X(x_1, x_2, \ldots, x_n)$*, where* $(x_1, x_2, \ldots, x_n)$ *represents a set of attributes linked to the symbol* X*,*
- $\mathcal{F}$ *is a set of terminal symbols, with their arities,*
- Σ *is a structure defining the constraint language and an interpretation for this constraint language,*
- *R is a finite set of production rules of the form* $X \to \alpha \Box c$*, where* $X \in N$*,* $\alpha \in T(\mathcal{F} \cup N)$*, and c is a set of constraints on the attributes of the non-terminals occurring in the rule. This set is interpreted as a conjunction of constraints.*

To express the constraints, we use a classical notation of attribute grammars: the value of an attribute named a in the non-terminal symbol N is denoted by $N.a$, and if there are more than one occurrence of the same non-terminal in one rule, these occurrences are numbered from left to right, starting from 0 for the head. For instance, we write $A_0 \to f(B, A_1, C, A_2)$ or $A \to g(B_1, C, B_2)$.

Example 3. A constraint tree grammar which generates the finite set of terms $p(f^n(x))$ for $5 \le n \le 50$.

$$
\begin{array}{ll}
G_c = \{ & \\
 & C, \\
 & \{A, B(count)\}, \\
 & \{p/_1, f/_1, x/_0\}, \\
 & (\,\mathbb{N}, \{=, \le\}, \{+\}), \\
 & \{ \\
 & \begin{array}{lll} A & \to p(B) & \{5 \le B.count, B.count \le 50\} \\ B_0 & \to f(B_1) & \{B_0.count = 1 + B_1.count\} \\ B & \to x & \{B.count = 0\} \end{array} \\
 & \}
\end{array}
$$

The basic structure of a Regular Tree Grammar is unchanged. But each rule is decorated with a set of constraints which expresses the contexts in which this rule can be applied.

In the framework of constraint regular tree grammars, the notion of derivation given in Definition 3 must be extended to deal with constraints: a set of constraints is associated to each derivation, namely the union of the sets of constraints associated to each rule applied in this derivation (with renamed variables). The language generated by a *constraint* tree grammar is the set of all ground terms that can be reached by successive derivations starting from the axiom and *satisfying the associated constraints*. The solver is represented by a *reject criterion*, denoted by RC; a set S of constraints is rejected (unsatisfiable) if $S \in RC$. Let us notice that the completeness of the solver depends on the domain and the constraints.

The introduction of constraints between attributes enables to stop deriving a partial tree, when its associated set of constraints is rejected by the solver. This is possible since new derivations can only add new constraints to this already unsatisfiable set.

3 Biases and Incrementality

The main advantage of using Regular Tree Grammars is that this allows an easy computation of set operations on languages, namely the intersection, the union and the complement of two languages. When used for modeling biases, this means that the conjunction, disjunction and negation of biases, expressing respectively the intersection, union and complement of the corresponding hypotheses spaces, can be easily computed.

When extended to Constraint Regular Tree Grammars, the ability to compute intersections and unions is kept, but the ability to compute differences is in general no longer satisfied.

3.1 Languages Operations

To define these operations, we must first introduce the notion of *normal form* for a grammar:

Definition 6. *A regular tree grammar $G = (A, N, \mathcal{F}, R)$ is in* **normal form** *if each production rule of R has the form: $X \rightarrow f(Y_1, ..., Y_n)$ with $X, Y_1, ..., Y_n \in N$ and $f \in \mathcal{F}_n$.*

Each regular tree grammar G can be automatically transformed into an equivalent regular tree grammar $\widetilde{G}$, written in normal form and leading to the same language ($L(G) = L(\widetilde{G})$) [3].

Definition 7. *Let $G_1 = (A_1, N_1, \mathcal{F}_1, R_1)$ and $G_2 = (A_2, N_2, \mathcal{F}_2, R_2)$ be two Regular Tree Grammars in normal form, which define respectively the tree languages L_1 and L_2, then:*

- *The language $L_\cap = L_1 \cap L_2$ is defined by the Regular Tree Grammar $G_\cap$:* $G_\cap = (A_1A_2, N_1 \times N_2, \mathcal{F}_1 \cap \mathcal{F}_2, R_\cap)$ *with* $R_\cap = \{X_1X_2 \rightarrow f(Y_1Z_1, ..., Y_nZ_n) \mid X_1 \rightarrow f(Y_1, ..., Y_n) \in R_1 \ \wedge \ X_2 \rightarrow f(Z_1, ..., Z_n) \in R_2\}$.
- *The language $L_\cup = L_1 \cup L_2$ is defined by the Regular Tree Grammar $G_\cup$:* $G_\cup = (A_1A_2, N_1 \times N_2, \mathcal{F}_1 \cup \mathcal{F}_2, R_\cup)$ *with* $R_\cup = \{XA_2 \rightarrow f(Y_1A_2, Y_2A_2, ..., Y_nA_2) \mid X \rightarrow f(Y_1, ...Y_n) \in R1\}$ $\cup \ \{A_1X \rightarrow f(A_1Z_1, ..., A_1Z_n) \mid X \rightarrow f(Z_1, ..., Z_n) \in R_2\}$.
- *The language $L_- = L_1 - L_2$ is defined by the Regular Tree Grammar G_-:* $G_- = (A_1A_2, (N_1 \times N_2) \cup N_1, \mathcal{F}_1 \cup \mathcal{F}_2, R_-$ *with* $R_- = R_1 \cup \{X_1X_2 \rightarrow f(Y_1, ..., Y_iZ_i, ...Y_n) \ \mid \ n > 0, i \in [1, n]$ *and*

$X_1 \to f(Y_1, ...Y_n) \ \in R_1, X_2 \to f(Z_1, ...Z_n) \ \in \ R_2 \ \}$
$\cup \{X_1X_2 \ \to \ f(Y_1, ...Y_n) \ \ | \ \ X_1 \ \to \ f(Y_1, ...Y_n) \ \in \ R_1 \ \ and \ \not\exists X_2 \ \to$
$f(Z_1, ...Z_n) \in R_2\}$

In these definitions, A_1A_2, X_1X_2, ... represent new terminal symbols. New terminal symbols of $N_1 \times N_2$ that do not occur in the production rules can be removed.

With Constraint Regular Tree Grammars, a notion of normal form can still be defined, as well as a way of computing the intersection of two constraint tree grammars written in normal form. It is important to notice that in this case, the process must be adapted to handle the attributes associated to non-terminals, as shown in Example 4. For more details, see [13].

Example 4. Given a Constraint Tree Grammar with non-terminals $A(n), B(m)$, and a rule $A_0 \to f(A_1, B)$ $\{A_0.n = A_1.n + 1, B.m = 0\}$, another one with non-terminals $D(k), E(n)$, and a rule $E_0 \to f(E_1, D)$ $\{E_0.n \leq E_1.n, D.k = 2 * E_1.n\}$, the intersection of these grammars leads to the rule:
$A_0E_0 \to f(A_1E_1, BD)$
$\{A_0E_0.n = A_1E_1.n + 1, BD.m = 0, A_0E_0.n \leq A_1E_1.n, BD.k = 2 * A_1E_1.n\}$.
Let us notice that this last set of constraints is unsatisfiable $(A_1E_1.n + 1 \leq A_1E_1.n)$, so this rule is useless and can be deleted.

Let us recall that, on the other hand, the difference is in general no longer computable.

3.2 A complete example

Let us consider a version of the well-known *Arch* problem, extracted from [19]. The problem is, given the set of unary predicates specified in the hierarchy of Figure 1, and a binary predicate *stable* represented by the graph in Figure 2, find the definition of an arch:
$arch(X, Y, Z) \leftarrow flatTop(X), flatTop(Y), volume(Z), stable(Z, X), stable(Z, Y)$.

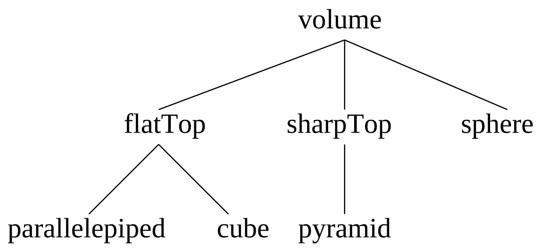

Fig. 1. The predicate hierarchy for the Arch problem.

The basic hypotheses space can be represented by the grammar given in Example 5, and defining an infinite space.

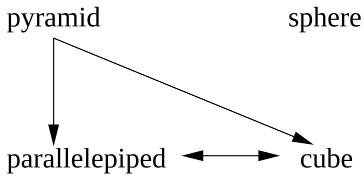

Fig. 2. Graph of the relation *stable*.

Example 5. A Regular Tree Grammar which generates the Arch problem hypotheses space.

$$
\begin{aligned}
G_1 = \{ & \\
& A, \\
& \{A, B, L, V\}, \\
& \{arch/_3, :-/_2, \wedge/_2, flatTop/_1, volume/_1, ..., stable/_2, x/_0, y/_0, z/_0, true/_0\}, \\
& \{ \\
& A \rightarrow arch(x, y, z) :- B \\
& B \rightarrow L \wedge B \mid true \\
& L \rightarrow flatTop(V) | volume(V) | ... | stable(V, V) \\
& V \rightarrow x \mid y \mid z \\
& \}
\end{aligned}
$$

The user can first restrict this language to clauses containing at least 3 and at most 5 literals in the body. This can be defined by a language containing the three patterns: $T :- T \wedge T \wedge T \wedge true$ to get 3 literals, $T :- T \wedge T \wedge T \wedge T \wedge true$ for 4 literals, and so on for 5 literals. The grammar G_2, given in Example 6, defines such a language. The language of clauses defining the arch problem and containing from 3 up to 5 literals is then $L = L(G_1) \cap L(G_2)$.

Example 6.

$$
\begin{aligned}
G_2 = \{ & \\
& A, \\
& \{A, B, T\}, \\
& \{arch/_3, :-/_2, \wedge/_2, flatTop/_1, volume/_1, ..., stable/_2, x/_0, y/_0, z/_0\}, \\
& \{ \\
& A \rightarrow T :- T \wedge T \wedge T \wedge B \\
& B \rightarrow true \mid T \wedge true \mid T \wedge T \wedge true \\
& T \rightarrow arch(T, T, T) \mid flatTop(T) \mid ... \mid stable(T, T) \mid x \mid y \mid z \mid true \\
& \}
\end{aligned}
$$

Let us notice that the grammar G_2 has been roughly defined and contains invalid clauses (for instance the term $x :- y \wedge true \wedge volume(sphere(z)) \wedge true$ belongs to $L(G_2)$). This does not matter, since the intersection with G_1 leads to only valid clauses. The grammar G_2 could also have been expressed by a Constraint Tree Grammar using an attribute *count* linked to the non-terminal T and counting the number of times T was derived.

Another usual and interesting bias in ILP is forbidden literals. For instance, with the Arch problem, there is no need to test *stable* on the same object ($stable(X,X)$, $stable(Y,Y)$ or $stable(Z,Z)$). We can express this bias by removing from L all clauses that contain such a literal. So we define the grammar G_3, of all terms with one of these three literals:

$$
\begin{array}{l}
G_3 = \{ \\
\quad A, \\
\quad \{A, B, T, S\}, \\
\quad \{arch/_3, :-/_2, \wedge/_2, flatTop/_1, volume/_1, ..., stable/_2, x/_0, y/_0, z/_0\}, \\
\quad \{ \\
\quad A \rightarrow T :- B \\
\quad B \rightarrow S | S \wedge T \; | T \wedge S \mid T \wedge S \wedge T \\
\quad S \rightarrow stable(x,x) \mid stable(y,y) \mid stable(z,z) \\
\quad T \rightarrow arch(T,T,T) \mid \; \wedge (T,T) \mid flatTop(T) \mid volume(T) \mid \\
\qquad\quad ... \mid stable(T,T) \mid x \mid y \mid z \mid true \\
\quad \}
\end{array}
$$

Then the new hypotheses space is $L' = L \setminus L(G_3) = (L(G_1) \cap L(G_2)) \setminus L(G_3)$.

The grammar obtained by computing the intersection of two grammars can be complicated. The main point is that the user can define simple biases by means of grammars, and then specify more complicated biases as the intersection of simpler biases; their grammars can then be automatically computed.

4 Some Algorithms to use this formalism

In this section, we propose different algorithms to search through a hypotheses space expressed by means of a Constraint Tree Grammar.

4.1 Generate and Test

The definition of derivations for Constraint Tree Grammars is closed to the definition of CSLD (Constrained SLD), the operational semantics for Constraint Logic Programming [8]. The trees generated by successive derivations of a Constraint Tree Grammar correspond to proof trees of a constraint logic program.

Using this similarity with Constraint Logic Programming, a Generate-and-Test algorithm can then be implemented: a constraint tree grammar is directly translated into a CLP program, which is executed to get all the valid terms of the language. For instance, the rule: $A_0 \rightarrow p(B, A_1)\{A_0.n = A_1.n + 1, B.u > B.v + A_0.n\}$ with $A(n)$ and $B(u,v)$ is translated into the constraint clause $a(N_0) :- b(U,V), a(N_1)\ \{N_0 = N_1 + 1, U > V + N_0\}$.

It should be possible to establish equivalences between the semantic of Constraints Grammars and the semantic of Constraint Logic Programs, in terms of *skeleton* of proof trees [5]. The study of these equivalences is beyond the framework of this paper.

4.2 Refinement Approach

A usual search strategy in ILP relies on a *refinement operator*. The idea is to partially order the hypotheses space, by means of a binary relation over clauses expressing a generalization partial order. This allows, for instance, to prune hypotheses that do not cover an example in a top-down (general to specific) strategy. The major difficulty is then to exploit at the same time, this generality order AND the language biases. A typical problem occurs when a current hypothesis does not satisfy the language biases, but some of its descendants through the refinement operator satisfy it. In that case, no pruning is possible.

Torre and Rouveirol [20] introduce an important property for biases in relation to refinement operators, namely private property.

Definition 8. *A property P is private with respect to the relation $\mathcal{R}$ iff* $\forall H, H' \in S : (H\ \mathcal{R}\ H' \wedge \overline{P(H)} \Rightarrow \overline{P(H')})$

Then, when a bias (property P) is *private* with respect to the refinement operator, hypotheses can be pruned as soon as they do not satisfy P, because none of their refinements would satisfy P.

A Simple Example Let us illustrate on a simple example the differences between private and not private properties through the refinement process. If we consider two simple biases over Horn clauses: the number of literals in the clause is greater or equal to 2 (property P_1), and less or equal to 4 (property P_2). Let us consider the usual top-down operator "add literal", denoted ρ in the following, and examine a refinement chain through the hypotheses space of Horn clauses[1]:

$$
\begin{aligned}
(1)\ p &\rightarrow_\rho p :- q_1 && (2)\\
&\rightarrow_\rho p :- q_1, q_2 && (3)\\
&\rightarrow_\rho p :- q_1, q_2, q_3 && (4)\\
&\rightarrow_\rho p :- q_1, q_2, q_3, q_4 && (5)\\
&\rightarrow_\rho p :- q_1, q_2, q_3, q_4, q_5 && (6)\\
&\rightarrow_\rho \cdots
\end{aligned}
$$

The hypotheses from (1) to (3) do not satisfy P_1, and after (4), they *always* satisfy it. So after (4), P_1 does not need to be tested anymore. On the other hand, the hypotheses from (1) to (5) satisfy P_2, and no hypothesis after (5) satisfy P_2 (the property P_2 is private v.s. the operator ρ). So we can safely prune the search at step (6).

The idea of the system $\mathcal{HAIKU}$ [20] is to ask the user to declare private properties to allow such a pruning optimization. We argue that in our framework, the solver can automatically detect some private properties. Let us show it on the previous example. The constraint rule used to derive the non-terminal B is $B_0 \rightarrow L \wedge B_1\ \{B_0.n = 1 + B_1.n\}$, where n is the context (on the domain $\mathbb{N}$)

[1] For sake of simplicity, we represent atom without their variables.

that represents the number of literals in the generated clause. P_1 and P_2 are expressed by means of constraints over this context: $B.n \geq 2$ and $B.n \leq 4$.

$$
\begin{array}{ll}
& H : -B \; \{B.n \geq 2, B.n \leq 5\} \quad (1) \\
\rightarrow_\rho & H : -L \wedge B_1 \\
& \{B.n \geq 2, B.n \leq 5, B.n = 1 + B_1.n\} \quad (2) \\
\rightarrow_\rho & H : -L \wedge L \wedge B_2 \\
& \{B.n \geq 2, B.n \leq 5, B.n = 1 + B_1.n, B_1.n = 1 + B_2.n\} \quad (3) \\
\rightarrow_\rho & H : -L \wedge L \wedge L \wedge B_3 \\
& \{B.n \geq 2, B.n \leq 5, B.n = 1 + B_1.n, B_1.n = 1 + B_2.n, B_2.n = 1 + B_3.n\} \quad (4) \\
\rightarrow_\rho & H : -L \wedge L \wedge L \wedge L \wedge B_4 \\
& \{B.n \geq 2, B.n \leq 5, B.n = 1 + B_1.n, B_1.n = 1 + B_2.n, B_2.n = 1 + B_3.n, \\
& B_3.n = 1 + B_4.n\} \quad (5) \\
\not\rightarrow_\rho & H : -L \wedge L \wedge L \wedge L \wedge L \wedge B_5 \\
& \{B.n \geq 2, B.n \leq 5, B.n = 1 + B_1.n, B_1.n = 1 + B_2.n, B_2.n = 1 + B_3.n, \\
& B_3.n = 1 + B_4.n, B_4.n = 1 + B_5.n\} \quad (6)
\end{array}
$$

In (3), a propagation algorithm (for instance, based on local consistency) can deduce from the two constraints $B.n = 1 + B_1.n, B_1.n = 1 + B_2.n$, and because the domains (over $\mathbb{N}$) of $B_2.n$ is $[0; +\infty[$, that $B_1.n$ is in $[1; +\infty[$, and then that $B.n$ is in $[2; +\infty[$. So $B.n \geq 2$ is redundant and can be removed from the store (i.e., the set of working constraints). The same propagation algorithm leads in (6) to $B.n$ is in $[5; +\infty[$, which is a contradiction with the constraint $B.n \leq 5$. So this set of constraints is unsatisfiable, and the derivation is pruned.

However, this interesting result does not hold for any property of any domain. One basic limitation is that the solver is not always complete.

5 Related works

Let us show by examples how the Tree Grammars formalism can be used to model some of the biases languages mentioned in the introduction.

5.1 Formalisms based on sets

Let us consider first the Clause Set formalism [1]. It extends standard Prolog notations with the introduction of clause sets, literal sets and term sets. A program, or a clause, or a literal belongs to the search space if it can be obtained by deleting one or more elements from the sets.

Example 7. A literal set example and all its corresponding clauses.

$$p(X) \; :- \; \{ \; a(X), b(f(X), Y) \; \}$$

$$
\begin{array}{l}
p(X). \\
p(X) \; :- \; a(X). \\
p(X) \; :- \; b(f(X), Y). \\
p(X) \; :- \; a(X), b(f(X), Y).
\end{array}
$$

Example 7 presents a Clause Set expression and the finite set of clauses defined by this expression. Since this hypotheses space is finite, it is always possible to define it with a regular tree grammar. Example 8 presents a set of production rules, that defines a language equivalent to the previous set of clauses.

Example 8. The production rules of the tree grammar G_2

$$\begin{array}{ll} C & \rightarrow p(x) \ :- \ B \\ B & \rightarrow B' \ \mid \ a(x) \wedge B' \\ B' & \rightarrow B'' \ \mid \ b(f(x), y) \wedge B'' \\ B'' & \rightarrow true \end{array}$$

Let us notice that a Clause Set specification, which generates a finite set of clauses can be automatically translated into an equivalent regular tree grammar: a non-terminal is introduced for each element of the sets and for each non-terminal, an alternative expresses whether it is present or not.

Another formalism to express structural biases is $\mathcal{D}$LAB [4]. It is a grammar based formalism which uses $\mathcal{D}$LAB terms to define sets of clauses. A $\mathcal{D}$LAB term is either an atomic formula, or a formula $Min - Max : T$ where T is a $\mathcal{D}$LAB term, $0 \leq Min \leq Max \leq length(T)$ are integers. This last case represents all subsets of T with length between Min and Max.

Example 9 shows a $\mathcal{D}$LAB definition and its expansion.

Example 9. A $\mathcal{D}$LAB term and the set of clauses it represents.

$$\begin{array}{l} DG = \{ \\ \qquad 1 - 1 : [p(X)] \\ \qquad\qquad \leftarrow \\ \qquad 0 - 2 : [a(X), b(X,Y), c(Y,X)] \\ \qquad \} \end{array}$$

$$L = \begin{cases} p(X). \\ p(X) \leftarrow a(X). \\ p(X) \leftarrow b(X,Y). \\ p(X) \leftarrow c(Y,X). \\ p(X) \leftarrow a(X), b(X,Y). \\ p(X) \leftarrow a(X), c(Y,X). \\ p(X) \leftarrow b(X,Y), c(Y,X). \end{cases}$$

As for the Clause Sets formalism, any $\mathcal{D}$LAB specification can be translated into an equivalent tree grammar.

Example 10. A tree grammar G_3 which generates a language $L(G_3)$ equivalent to the language L given in 9.

$$
\begin{array}{l}
G_3 = (\\
\quad C, \\
\quad \{C, B, B', B''\}, \\
\quad \{:-/_2, \wedge/_2, a/_1, b/_2, c/_2, x/_0, y/_0\}, \\
\quad \{ \\
\qquad C \rightarrow p(x) :- B \mid B' \mid B'' \\
\qquad B \rightarrow true \\
\qquad B' \rightarrow a(x) \mid b(x,y) \mid c(y,x) \\
\qquad B'' \rightarrow a(x) \wedge b(x,y) \\
\qquad B'' \rightarrow a(x) \wedge c(y,x) \\
\qquad B'' \rightarrow b(x,y) \wedge c(y,x) \\
\quad \} \\
)
\end{array}
$$

This translation is always possible, because each $\mathcal{D}$LAB language is finite, but not very intuitive and leads to enumerate all the possibilities. A more natural way to express it would be to allow to derive $c(y,x)$ only if the maximum number of previous derivations (with $a(x)$ and $b(x,y)$) is not reached. This requires first, to *count* derivations and then, to allow some derivations, depending on this count.

The Constraint Tree Grammar formalism allows to express the $\mathcal{D}$LAB example given in 9, by counting the derived literals, as shown in Example 11.

Example 11. A Constraint Tree Grammar equivalent to the $\mathcal{D}$LAB Example 9.

$$
\begin{array}{llll}
G_3 = \{ & & & \\
\quad C, & & & \\
\quad \multicolumn{3}{l}{\{C, B(count), B'(count), B''(count)\},} \\
\quad \multicolumn{3}{l}{\{:-/_2, \wedge/2, true/0, p/_1, a/_1, b/_2, c/_2, x/_0, y/_0\},} \\
\quad \multicolumn{3}{l}{(\mathbb{N}, \{=, \leq\}, \{+\}),} \\
\quad \{ & & & \\
\quad C & \rightarrow p(x) :- B & \{0 \leq B.count, B.count \leq 2\} \\
\quad B & \rightarrow B' & \{B.count = B'.count\} \\
\quad B & \rightarrow a(x) \wedge\ B' & \{B.count = 1 + B'.count\} \\
\quad B' & \rightarrow B'' & \{B'.count = B''.count\} \\
\quad B' & \rightarrow b(x,y) \wedge\ B'' & \{B'.count = 1 + B''.count\} \\
\quad B'' & \rightarrow true & \{B''.count = 0\} \\
\quad B'' & \rightarrow c(x,y) & \{B''.count = 1\} \\
\quad \} & & &
\end{array}
$$

In this example, the integer attribute *count* is a context which represents the number of derived literals (a, b, c). The constraints associated to the B rules (B, B', B'') express how to compute this context by means of dependencies between "*count*" occurrences. The constraints on the first rule restrict the valid trees to those for which the context is between 0 and 2.

5.2 MILES-CTL

For sake of simplicity, we only present here an example of MILES-CTL and a grammar which generates the same language. Let us notice that the slot *arity* has been transformed in our formalism into an attribute and that the restrictions on that slot have been transformed into constraints.

Example 12.

$$T1 : \begin{bmatrix} predicate : P11, \\ arguments : A11 \end{bmatrix} \leftarrow \begin{bmatrix} predicate : P12, \\ arguments : A12 \\ arity : Ar12 \| (Ar12 \leq 3) \end{bmatrix}.$$

$$T2 : \begin{bmatrix} predicate : P21, \\ arguments : A21 \end{bmatrix} \leftarrow \begin{bmatrix} predicate : P22, \\ arguments : A22 \\ arity : Ar22 \| (Ar22 \leq 4) \end{bmatrix}, \begin{bmatrix} predicate : P23, \\ arguments : A23, \\ predicate\_type : comp \end{bmatrix}.$$

"Equivalent" grammar rules:

$$\begin{array}{l} \{ \\ A \ \rightarrow P_1 : -P_2 \qquad \{P_2.arity \leq 3\} \\ A \ \rightarrow P_1 : -P_2 \wedge P_3 \ \{P_2.arity \leq 4, P_3.type =' comp'\} \\ \ldots \\ \} \end{array}$$

5.3 Grendel

The biases representation language of Grendel uses grammars that generate strings of literals, instead of symbols. It uses a macro facility, called *lazy macro*, that allows to concisely express sets of grammar rules and that offers second order possibilities [2]. Rules of the form "$A \rightarrow B\ where\ P$", where P is a Prolog goal, are macro-expanded to $A\theta_i \rightarrow B\theta_i$ where θ_i are substitutions associated to the proofs of P.

The derivation relation is closed to the usual string grammar derivation, but it relies on the unification: Let "$B \rightarrow C\ where\ P$"be a rule and σ be the m.g.u. of the literals A and B, then $\alpha A \beta$ is derived into $(\alpha C \beta)\sigma\theta$ where θ is a substitution associated to the proofs of $P\sigma$.

Let us express with this formalism the biases P_1 and P_2 presented in Section 4.2.

Example 13. The number of literals in the clause must be between 2 and 4.

$Body(p(X, Y)) \rightarrow rels(X, Y, 0)\ where\ true.$

$rels(X, Y, N) \rightarrow []\ where\ N >= 2.$
$rels(X, Y, N) \rightarrow rel(X, Y), rels(X, Y, M)\ where\ N =< 4, M\ is\ N + 1.$

$rel(X, Y) \rightarrow q(A, B)\ where\ member(A, [X, Y]), member(B, [X, Y]).$

Here, the variable N represents the context (number of literals), and the expressions $N =< 4$ and $N >= 2$ state that it must be between 2 and 4.

This example shows the similarities between Grendel's formalism and our Constraint Tree Grammar (CTG) formalism:

- The contexts are defined by arguments of literals.
- The contextual restrictions for rules are expressed by means of *where* goals, so that rules can be applied only when the goals succeed.

But this implies that variables of the clause and contexts (for instance variables X, Y, and context N) appear in the literals whereas they are clearly separated in our formalism. On the other hand, this allows to easily manipulate variables, by means of unification, whereas no other formalism allows it.

The main difference appears in the *rels* definitions, where the constraint $N >= 2$ *must* be placed in the first rule, whereas the constraint $N =< 4$ *must* be placed in the recursive rule. This means that a user must carefully modify the overall grammar to add such biases to a given specification in Grendel, whereas he can simply add $A.n >= 2, A.n =< 4$ on the axiom with the CTG formalism.

6 Conclusion

The formalism we propose is more expressive than many classical languages for expressing biases, since it allows to define infinite and contextual search spaces. The syntax may seem complicated, because it relies on grammars and constraints. We see several ways of overcoming that problem:

- because of incrementality, each bias could be defined separately, and conjunction of biases could then be automatically computed;
- library of biases can be pre-defined, so that the user has only to choose the biases that are relevant to his learning task;
- some constraints, for instance those relying on the counting of elements, could be automatically generated;
- finally, as shown in Section 5, some formalisms that may seem easier to use could be automatically translated into our formalism. It should be possible to let the user choose the formalism which seems the most adapted to his task, model the biases in that formalism, and then to translate his specification into our formalism.

Our goal is to use this formalism for Data Mining tasks in a relational framework, using a RDBMS (Relational DataBases Management System). Despite the links between the logical framework and the relational one, characteristics of Data Mining tasks differ from usual ILP ones, from the point of view of the hypotheses space:

- The search space is more restricted: hypotheses are usually not recursive and possible joins between literals are very limited (often to an ID attribute).

– The database provides meta data, namely the database scheme, with types, constraints, and join restrictions.
– Potentially, the number of functions that can be applied to each variable is important (for instance, all integrated date functions in the RDBMS).

So, an idea is first to automatically extract information from the database schemata to define the hypotheses space of syntactically and semantically (types) well-formed clauses. Let us notice that this space is usually infinite. Then, the user can express one or more biases over this hypotheses space, for instance to get a finite search space. There are two solutions : the user modifies by hand the automatically generated specification, but this seems difficult with all the integrated functions, relations, types ... The other solution is for the user to provide a restriction of this initial search space, without modifying it. With the CTG formalism, this can be achieved by:

– Adding constraints over the attributes of the axiom, so that the user does not need to look at the rules of the grammar.
– Providing another CTG specification of interesting hypotheses, and compute the intersection of this specification with the initial one.

Another interesting point that we would like to study is the shift of bias. It is a change of the hypotheses space, which could in that framework either be expressed in terms of sets operations, or in terms of relaxation of constraints.

Such a framework can be easily used in some inductive learner, for instance using a generate and test approach. The use of Tree Grammars to model biases goes beyond the domain of ILP, since it has also been applied in a Genetic Programming framework [12], and has been used in an application [11] for learning relational algebra expressions.

References

[1] F. Bergadano and D. Gunetti. *Learning clauses by tracing derivations.* MIT Press, 1995.
[2] W. W. Cohen. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68:303–366, 1994.
[3] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 1997.
[4] L. Dehaspe and L. De Raedt. Dlab: A declarative language bias formalism. In *Proceedings of the International Symposiumon Methodologies for Intelligent Systems (ISMIS96)*, pages 613–622. Springer Verlag, 1996.
[5] P. Deransart and J. Maluszyński. *Grammatical view of Logic Programming.* MIT Press, 1993.
[6] F. Gécseg and M. Steinby. *Tree Automata.* Akadémiai Kidoó, Budapest, 1984.
[7] J. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Language, and Computation.* Addison–Wesley, Reading, MA, 1979.
[8] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In 14^{th} *ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987.

[9] J-U. Kietz and S. Wrobel. Controlling the complexity of learning in logic through syntactic and task-oriented models. In S. Muggleton, editor, *Proceedings of the 1st International Workshop on Inductive Logic Programming*, pages 107–126, 1991.

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

[11] L. Martin, F. Moal, and C. Vrain. A relational data mining tool based on genetic programming. In *Proceedings of the 2nd European Symposium on Principles of Data Mining and Knowledge Discovery*, volume 1510 of *LNAI*, pages 130–138. Springer, 1998.

[12] L. Martin, F. Moal, and C. Vrain. Declarative expression of biases in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 401–408. Morgan Kaufmann, 1999.

[13] F. Moal. Biases and constraint tree grammars. Technical Report RR-2000-07, LIFO, Université d'Orléans, France, To Appear in 2000.

[14] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19 & 20:629–680, May 1994.

[15] L. De Raedt. *Interactive Theory Revision: an Inductive Logic Programming Approach*. Academic Press, 1992.

[16] S. Russell and B. Grosof. A sketch of autonomous learning using declarative bias. In *Machine Learning, Meta-Reasoning and Logics*, pages 19–54. Kluwer Academic Publishers, 1990.

[17] U. Sarkans and J. Bārzdiņš. Using attribute grammars for description of inductive inference search space. In *Proceedings of the 9th International Conference on Algorithmic Learning Theory (ALT-98)*, volume 1501 of *LNAI*, pages 400–408, Berlin, October 8–10 1998. Springer.

[18] B. Tausend. Representing biases for inductive logic programming. In *Proceedings of the European Conference on Machine Learning*, volume 784 of *LNAI*, pages 427–430, Berlin, April 1994. Springer.

[19] F. Torre. Exploitation de biais de langage en apprentissage symbolique automatique dans un cadre logique. DEA d'Informatique, L.R.I., Université Paris-Sud France, June 1995.

[20] F. Torre and C. Rouveirol. Natural ideal operators in inductive logic programming. In *9th European Conf. on Machine Learning (ECML'97), Prague, Czech Republic, LNAI 1224*, volume 1224, pages 274–289. Springer-Verlag, April 1997.

[21] R. Wilhelm and R. Maurer. *Les compilateurs - theorie, construction, generation*. Masson, 1994.