Grammatical Inference as Unification

Jacques Nicolas

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE

Grammatical inference as unification

Jacques Nicolas

No 3632

mars 1999

THÈME 3

ISSN 0249-6399

Grammatical inference as unification

Jacques Nicolas *

Thème 3 — Interaction homme-machine,
images, données, connaissances
Projet Aïda

Rapport de recherche n ° 3632 — mars 1999 — 21 pages

Abstract: We propose to set the grammatical inference problem in a logical framework. The search for admissible solutions in a given class of languages is reduced to the problem of unifying a set of terms. This point of view has been already developed in the particular context of categorial grammars, a type of lexicalized grammar. We present the state of the art in this domain and propose several improvements. The case of regular grammars is studied in a second part. We show that rational unification allows to infer such grammars. We give corresponding Prolog programs in both cases. Indeed, one of the aim of this work is to show that "lean" programs are possible for grammatical inference. This approach has been successful in the field of automated theorem proving and we expect to observe the same benefits in grammatical inference : efficiency and extendibility.

Key-words: grammatical inference, unification, categorial grammars, Prolog

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

L'inférence grammaticale vue comme un problème d'unification

Résumé : Nous proposons d'inscrire le problème de l'inférence grammaticale dans le cadre de la programmation logique. La recherche de solutions admissibles dans une classe donnée de langages est réduite au problème de l'unification d'un ensemble de termes. Ce point de vue a déjà été développé dans le contexte d'un type particulier de grammaires lexicalisées, les grammaires catégorielles. Nous présentons l'état de l'art dans ce domaine et proposons plusieurs améliorations. Le cas des grammaires régulières est étudié dans une deuxième partie. Nous montrons que de telles grammaires peuvent être inférées par unification rationnelle. Nous donnons les programmes Prolog correspondants dans les deux cas. En effet, un des buts de ce travail est de montrer qu'il est possible d'écrire des programmes "maigres" pour l'inférence grammaticale, approche qui a été appliquée avec succès en démonstration automatique. Nous en attendons ici les mêmes avantages : extension aisée et efficacité du code

Mots-clé : inférence grammaticale, unification, grammaires catégorielles, Prolog

1. Introduction

We propose in this paper to establish a connection between grammatical inference and the field of logic programming. We claim such a work is useful for at least two reasons. First, it allows to better understand the fundamental structures underlying algorithms and provides a first step for the comparison and integration of Inductive Logic Programming techniques in grammatical inference. Second, it allows to quickly produce efficient algorithms, making profit of the significant background in logic programming to only focus on aspects really relevant to grammatical inference.

The idea of using unification as a controlling device for the analysis of languages has been particularly fruitful in computational linguistics and natural language processing. A number of "unification-based" grammar formalisms have been developed : Definite Clause Grammars, Lexical Functional Grammars, Head-driven Phrase Structure Grammars... The idea of considering grammatical inference as a unification problem is itself not so new, although corresponding studies still remain confidential. It has already been developed in the restricted domain of categorial grammars. Since these grammars are essentially known in the field of computational linguistics, we will first recall here some necessary definitions for the basic Adjukiewicz-Bar-Hillel categorial grammars. The rest of the section rapidly introduces definitions and notations for unification and anti-unification.
The paper itself is organized in two parts : the first one studies unification in the framework of categorial grammars and the second one proposes to use rational unification in the framework of finite automata. Prolog implementations of presented algorithms are given in the appendix.

1.1. Categorial grammars

The grammatical inference community traditionally consider rule grammars, where languages are described via sets of rules or automata breaking down a sentence in more elementary parts until the level of words. These so-called "phrase structure grammars" contrast with lexicalized grammars, where all built structures (mainly trees) are supported by at least one word. Saying it differently, the grammar lies in the lexicon : a set of possible structures is associated to each word in the lexicon and the analysis of a phrase may proceed by assembling structures corresponding to each occurrence of a word in a phrase in a suitable way. From a formal point of view phrase structure grammars and lexicalized grammars have the same expressiveness, in the sense that it is always possible to exhibit a grammar in one system that is (weakly) equivalent to a given grammar in the other system.

Basic (or classical) categorial grammars were introduced in [2] and refined by Lambek [13] in a attempt to give a mathematical basis for natural language constructions. The interested reader may consult [5, 21, 31, 20] for further references and extensions of this basic formalism. We give a minimal set of definitions, using Lambek notation for types (the term type is used instead of category, in order to avoid any confusion with objects of the category theory in theoretical computer science).

Definition 1. (Types) We assume the existence of a set P of symbols without / or \ called primitive types, made of a set of variables V and a distinguished constant principal type t. The set T_P of types is the smallest set satisfying :

- $P \subseteq T_P$
- $\forall t, t' \in T_P, t/t' \; and \; t\backslash t' \in T_P$.

Definition 2. (Basic categorial grammar) A basic categorial grammar is a triple $G = \{\Sigma, T_P, M\}$, where Σ is an alphabet (the lexicon), T_P is a set of types and M is a finite mapping from L to 2^{T_P} assigning a set of types to each symbol in the alphabet. If there is only one type for each symbol, the grammar is said to be deterministic or rigid. If there are at most k types per symbol, the grammar is said to be k-valued. Languages corresponding to k-valued grammars form a strict hierarchy for k increasing.

Definition 3. (Derivation, Generation) For a given set of types T_P, the $\Longrightarrow$ derivation relation is the smallest relation satisfying (note that the derivation is oriented, the direction of the applied rule $\leftarrow$ or $\rightarrow$ corresponding to the position of the argument -second parameter of the derivation rule- with respect to the functor -first parameter- in the string)

- $\forall t \in T_P, t \Longrightarrow t$
- $\forall \Gamma, \Delta \in {T_P}^+, \forall t, t' \in T_P,$
  $\leftarrow$: if $\Gamma \Longrightarrow t'$ and $\Delta \Longrightarrow t'\backslash t$, then $\Gamma, \Delta \Longrightarrow t$
  $\rightarrow$: if $\Gamma \Longrightarrow t/t'$ and $\Delta \Longrightarrow t'$ then $\Gamma, \Delta \Longrightarrow t$

For a given grammar $G = \{\Sigma, T_P, M\}$, and a string $w = w_1 \dots w_n \in \Sigma^n$, G generates w iff $\exists t_1, \dots, t_n, \forall i \in [1, n], \; M(w_i, t_i) \wedge t_1, \dots, t_n \Longrightarrow t$.
One can associate a functor-argument structure for a given parse which is a binary tree where each internal node is labeled with the name of the applied derivation rule and each leaf is an element of Σ.

EXAMPLE: The basic categorial grammar recognizing the language $a^n b^n, n \geq 1$ (note that languages must be $\epsilon - free$) is $G = \{\{a, b\}, T_{\{t,X\}}, \{< a, \{t/X, (t/X)/t\} > , < b, \{X\} >\}\}$.
The functor-argument structure corresponding to the generation of *aabb* is the binary tree $\rightarrow_t (\rightarrow_{t/X} (a_{(t/X)/S}), \rightarrow_t (a_{t/X}, b_X)), b_X)$, where the written indices give the type of each node. □

1.2. Unification and anti-unification

A unification algorithm is an algorithm solving sets of equations over a set of terms. Although it is a well known concept, we recall some definitions in order to precise notations and stay self-contained. For a good introduction, see [17].

Definition 4. (Terms, substitutions) Given a set V of variables and a set $F = \bigcup_i F_i$ of function symbols with an associated number called arity, the set T of terms is recursively defined as :

$$V \subseteq T \quad \wedge \quad \forall f_i \in F,\ \forall t_i \in T,\ f_i(t_1, \ldots, t_i) \in T.$$

A substitution is a finite mapping from V to T denoted $\sigma = \{v_1 \rightarrow t_1, \ldots, v_n \rightarrow t_n\}$. Applying the substitution σ to a term t results in a new term $t\sigma$ (an instance of t) where all variables v_i of t have been simultaneously replaced by t_i. the set of substitutions may be ordered. A substitution σ is more general than a substitution θ if there exists a substitution γ such that $\theta = \sigma \circ \gamma$.

The notions of most general unifier(mgu) and least common anti-instance (lca) are due to Robinson [25] and Plotkin [24] respectively. Under fairly general hypotheses (renaming of variables), mgu and lca of a set of terms are unique.

Definition 5. (mgu, lca) A set of terms T is unifiable iff there exists a substitution σ (the unifier) such that any couple of terms t_1 and t_2 in T is unifiable, that is, such that $t_1\sigma = t_2\sigma$. Up to a renaming of variables there exists a unique most general unifier mgu. σ is a unifier of a set of set of terms iff it unifies each set of terms.
A set of terms T is anti-unifiable iff there exists a term t (the anti-instance) such that for all t_i in T , there exists a substitution σ such that $t_i = t\sigma$. A lca is a most specific anti-instance, that is all other anti-instances are also anti-instance of the lca. Up to a renaming of variables there exists a unique lca. The anti-instance of a set of set of terms is the set of anti-instances of each set of terms.

2. Learning classes of categorial grammars

Categorial grammars have been essentially studied as a mean to formalize natural language processing. In this domain, learning must be possible from positive instances only, since a child seems to have very few negative instances available while acquiring a new language. M. Kanazawa [12] gives a thorough study of learnability in this context and exhibits some non-trivial classes of categorial grammars which are learnable from positive instances. We focus here on the two classes that seem most important, rigid and k-valued grammars, since they form the core of multiple extensions, remain practically computable under reasonable assumptions and are (at least theoretically) learnable from strings. We also present several extensions when the training context becomes richer : availability of negative instances or of a semantic representation of sentences.

2.1. Learning rigid and k-valued grammars from structured instances

Categorial grammars and ϵ-free context-free grammars represent the same class of languages. The inference of context-free grammars (CFG) from positive instances

requires the choice of a set of restrictions in order to obtain a learnable and practically computable class of grammars. Most authors assume instances are presented together with their grammatical structure. For CFGs, these structured descriptions are made of unlabeled derivation trees. It is known that the set of derivation trees of a given CFG is rational, that is, may be recognized by some tree automaton [26, 27].
In case of categorial grammars, derivation trees are oriented. Thus, an instance is a functor-argument structure (see definition 3), where each node indicates the type of the corresponding derivation. Compared to the case of CFGs, algorithms developed so far benefit from an auxiliary information, the positions of the arguments, that may be hard to obtain in practice. Although Kanazawa suggests the (theoretical) possibility to learn k-valued grammars from strings by enumerating all possible functor-argument structures, the understanding of the structure of the underlying search space remains insufficient to hope for efficient algorithms. We propose an intermediary setting, learning from totally or partially unlabeled derivation trees.

The search space of rigid categorial grammars as a lattice of terms
The class RG of rigid grammars may be partially ordered with the subsumption relation : $G_1 \leq G_2$ iff $\exists \sigma, G_1\sigma \subseteq G_2$. M. Kanazawa [12] proves that $(RG, \leq)$ forms a complete lattice if a top element is added, completing the lattice so that each pair of grammars has a least upper bound (lub).
If it exists (i.e. differs from top), the lub of two grammars may be defined simply : Let $GF(X) = \{\{T \mid < c, T > \in M_X\} \mid c \in \Sigma_X\}$. Let $\sigma = mgu(GF(G_1 \cup G_2)))$. Then $lub(G_1, G_2) = (G_1 \cup G_2)\sigma$.
The greatest lower bound (glb) of two grammars may be defined in a dual way : Let $glb(G_1, G_2) = lca(GF(G_1 \cap G_2)))$.

Furthermore, Kanazawa considers the subclass RRG of rigid grammars in reduced form and the class RFA of rigid functor-argument structures. A grammar is in reduced form if no smaller grammar produces the same set of functor-argument structures (fa-structures). He shows that the structure $(RFA, \subseteq)$ is isomorphic to the structure $(RRG, \leq)$, thus characterizing the search space for the grammatical inference problem.

Learning rigid grammars from partially labeled structures

EXAMPLE: We illustrate the behavior of the algorithms with the following example found in [18] :
$\leftarrow$(Mary,$\rightarrow$(likes,John)),
$\leftarrow$(John,$\rightarrow$(knows,Mary)),
$\leftarrow$(John,$\rightarrow$(likes,Susan)),
$\leftarrow$(Mary,$\rightarrow$(knows,$\leftarrow$(John,$\rightarrow$(likes,Susan)))) □

Buszkowski [4] has proposed an algorithm Kanazawa shows to learn the class of rigid grammars in the limit from a set P of fa-structures.
The algorithm proceeds in four steps :

1. Assign a type to each node of structures in P (the root has type t, a has type X_2 (resp. X_1/X_2) and b type $X_2\backslash X_1$ (resp. X_2) if the parent node $\leftarrow_t (a, b)$ (resp. $\rightarrow_t (a, b)$ has type X_1.

2. Collect the types of leaves, producing the general form $GF(P)$.

3. Try to compute $\sigma = mgu(GF)$.

4. If it exists, the solution is $RG(P) = GF(P)\sigma$

This algorithm runs in linear time with respect to the number of instances and is almost linear with respect to the size of instances.
We propose in the appendix a Prolog program computing RG(P). Unification with occur-check is required, since it might produce infinite types otherwise (if the prolog compiler does not implement rational unification by default, standard unification is sufficient).
fa-structures are represented as terms of the form

$$n(Rule\_Label, Right\_subtree, Left\_subtree),$$

where Rule_Label is b (backward direction) or f(forward direction).
Leaves are of the form l(Symbol).

EXAMPLE: Thus, our example is represented with the following terms :
n(b,l('Mary'),n(f,l(likes),l('John'))), n(b,l('John'),n(f,l(knows),l('Mary'))),
n(b,l('John'),n(f,l(likes),l('Susan'))),
n(b,l('Mary'),n(f,l(knows),n(b,l('John'),n(f,l(likes),l('Susan'))))).
The algorithm produces the following results at each step :
Instance 1 : RG = ['John'-X, 'Mary'-Y, likes-(Y\t)/X].
Instance 2 : RG = ['John'-X, 'Mary'-Y, knows-(X\t)/Y, likes-(Y\t)/X].
Instance 3 : RG = ['John'-X, 'Mary'-X, 'Susan'-X, knows-(X\t)/X, likes-(X\t)/X].
Instance 4 : the solution is RG = ['John'-t, 'Mary'-t, 'Susan'-t, knows-(t\t)/t, likes-(t\t)/t]. □

The lexicon is represented with an AVL tree ("balanced" tree, for which the heights of the two subtrees issued from a given node differ by at most 1), allowing lookup and update in $O(\log m)$, if m is the size of the lexicon. Note that Kanazawa also proposes in [12] a Prolog implementation of the main algorithms he describes. However, it is only for an illustration purpose. His programs are not only very inefficient but do not propose a synthetic view of the algorithm. This point is important because a compact code in logic programming (a "lean program", as it is called in [3]) capture the essence of an algorithm and leads to efficient and extendable implementations.

We slightly improve the original algorithm in two ways.
First, we propose an incremental two steps algorithm : each structure is processed in turn, recursively assigning a type to its nodes and immediately updating the lexicon when the current node is a leaf. This is done by creating a new entry in the

dictionary if it is the first occurrence of the symbol or else by computing the unified type of the symbol in the grammar. Exploiting the associativity of unification, this operation allows to detect failures as soon as possible and this advantage will reveal to be important for extensions of the basic algorithm, when it is called a number of times for various parameter values. Moreover, the grammar is computable with a space complexity not depending on the number of instances.
Then, we do not require the structures in P to be labeled. All possible assignments are checked non-deterministically.

EXAMPLE: For instance, our example may be represented with terms
n(X_1,l('Mary'),n(X_2,l(likes),l('John'))), n(Y_1,l('John'),n(Y_2,l(knows),l('Mary'))),
n(Z_1,l('John'),n(Z_2,l(likes),l('Susan'))),
n(R_1,l('Mary'),n(R_2,l(knows),n(R_3,l('John'),n(R_4,l(likes),l('Susan'))))),
where uppercase terms are variables. It produces 4 solutions such that : $X_1 = R_1 = b$, $X_2 = R_2 = f$ and $Y_1Y_2Z_1Z_2R_1R_2 \in$ {bfbfbf,ffbbbb,fffbfb,ffffff}. □

This last modification has important consequences from a computational and from a learning point of view. Let n_i denotes the size of the i^{th} string in the learning set. Since an fa-structure is a binary tree, its number of non terminal nodes is $n_i - 1$. The number of possible assignments for the label of these nodes is at most $2^{\Sigma(n_i-1)}$ (namely, each node may be a backward function or a forward function). Although this worst-case analysis introduces an exponential factor, fa-structures may be partially labeled. Furthermore it is not simple to evaluate the number of choices that are checked in practice, this number being very dependent on the size of the lexicon itself and the strength of constraints observed on frequently occurring words (for our simple example, there are only 4 solutions in a space of 1024 possibilities). Ordering instances with respect to the sum of frequencies of its words in the whole learning set seems a good strategy to reduce the number of possible assignments.
From a learning point of view, loosening the requirement for oriented derivation trees leads to interesting possible extensions of the learning criterion. The idea is to optimize some function expressing the complexity of the induced grammar. We give in the appendix two simple implementations of such criteria : the first one consist to fix the number of primitive types in the grammar, that is generally known to be quite low ; the second minimizes the total number of symbols used in the representation of the grammar (not to be confused with minimal grammars of Buszkowski and Penn [6], where the total number of types is minimized. This measure is constant and thus not relevant for rigid grammars). rg_min_size produces with unlabeled fa-structures a unique solution (the same solution as rg for labeled fa-structures). An interesting open problem is the existence of possible relations between solutions for different labels : are there any generality ordering between some solutions ? Can one detect constant type assignments in all solutions ?

Learning k-valued grammars
Kanazawa has proved that the class of k-valued grammars is learnable from structures. He also proposes an algorithm that is an extension of the rg algorithm, based

on an extended concept of unification.
Formally, a k-partial unifier of a set of types T is a substitution *sigma* such that $card(\{t\sigma \mid t \in T\}) \leq k$. A k-partial unifier of a set of set of types is a k-partial unifier of all its elements.
An other equivalent definition for k-partial unifiers is that they are unifiers of k-partitions (where a k-partition is a partition with at most k elements in each block). The difficulty is then to propose the definition of a concept similar to mgu's, with the difference that multiple incomparable solutions may exist. We need to compute the set MGU_k of maximally general k-unifiers. Kanazawa uses the second definition of k-partial unifiers to focus on the set $k-MGU$ of mgu's of k-partitions.

$$k-MGU(T) = \{mgu(P) \mid P\ is\ a\ k-partition\ of\ T\}.$$

This set contains MGU_k but may unfortunately contain other elements. It does not precludes learnability of the class of k-valued grammars but leads to an algorithm that may produce too many solutions in a practical context.

In our example, the algorithm kvg finds 224 solutions for k=2 and labeled instances, 200 of which are maximally general. He finds 126799 solutions with unlabeled instances, 126775 of which are different. This last result is computed in 40" on an ultrasparc with our program but it clearly demonstrates that the subsumption problem becomes rapidly intractable, even for small examples. As a side effect, it also shows that equivalent solutions are possible for different labelings.

The algorithm proposed by Kanazawa is an immediate extension of Buszkowski's rg algorithm, where $k-MGU$ is computed instead of mgu's. As for the case of rg, we propose to follow an improved incremental strategy to update $k-MGU$. The inference of k-valued grammars for k>1 introduces a new problem with respect to rigid grammars. For a given word, a type may be useless in the sense that it is a multiple occurrence of another type of the same word. Although Kanazawa does not mention this problem, a costly redundancy test is necessary to check these multiple occurrences. In order to reduce this cost, we propose to forbid multiple occurrences as soon as possible, that is, during the evaluation of $k-MGU$. This assumes that inequations are introduced to constraint unification. Finally, we define and use the following disjunctive unification :

$$or\_unify_k(T,t) = T_{\vee t\sigma} \cup dif(t),$$

where $T_{\vee t\sigma}$ is the set of elements built from the set of types T in which one of the type has been unified with t and $dif(t)$ is type t if k is strictly greater than the size of T and $t \neq t'$ for all type t' in T.
$k-MGU(\{t_1,\ldots,t_n\})$ may then be computed incrementally with or_unify_k :

$$T_1 = or\_unify_k(\emptyset, t_1), T_2 = or\_unify_k(T_1, t_2), \ldots,$$
$$T_n = k-MGU(\{t_1,\ldots,t_n\}) = or\_unify_k(T_{n-1}, t_n)$$

The ordered implementation we propose in the appendix has an interesting property : it produces solutions following an increasing generality ordering that is

compatible with the optimization of k. This property may be exploited in algorithms that assure the production of a grammar for any input set of fa-structures. This is always possible if k is not fixed (the general form grammar that generates exactly the input set is always a solution). The first idea for this purpose is to look for the least k such that there exists a k-valued grammar. Kanazawa proposes to learn these so called least value grammars by iterating on increasing values of k. Our algorithm is much more efficient since least value grammars are the first solutions produced : it is sufficient to stop the search when the value of k for a solution becomes larger than the value of k for the first solution.
A second idea has been proposed by Buszkowski and Penn [6]: to learn optimal grammars, where an optimal grammar has no unifiable types for a same symbol (there is in fact a second more technical condition that is easy to fulfill : all types are used in some parse tree). They define for this purpose an extended unification procedure, optimal unification. $OU(T) = \{mgu(P) \mid P is an optimal partition of T\}$, where an optimal partition is such that no coarser partition has a unifier. It corresponds to most specific solutions.

2.2. Learning categorial grammars with richer data

We end this part mentioning two works that illustrate the great variety of potential extensions of the basic framework we have just described. The first one studies the modification of the algorithm elaborated by Buszkowski and Penn when a negative information is available. The second one studies the possibility to learn in a more general context where semantic information is available.

Learning with negative instances
In [18], J. Marciniek studies the effects of taking into account negative constraints while learning optimal grammars. This allows to reject grammars that may be simple but with a weak linguistic likelihood. The idea is to restrict the set of admissible optimal unifiers, rejecting any substitution that allows to accept a negative instance. In his work, negative instances are fa-structures. Unfortunately, he does not give any explicit algorithm, nor address the case of k-valued grammars.
We propose to extend our algorithm learning k-valued grammars, taking into account a simpler negative information in the form of pairs of (symbol, sets of types that are forbidden for this symbol). In fact, such an information may be managed very naturally in our algorithm since we have already introduced the necessity to solve inequations in the computation of k-MGUs. It is thus sufficient to modify the test $dif(t)$, so that t remains also different from all forbidden types.

EXAMPLE: We consider again our example
n(b,l('Mary'),n(f,l(likes),l('John'))), n(b,l('John'),n(f,l(knows),l('Mary'))),
n(b,l('John'),n(f,l(likes),l('Susan'))),
n(b,l('Mary'),n(f,l(knows),n(b,l('John'),n(f,l(likes),l('Susan'))))).
n(X_1,l('Mary'),n(X_2,l(likes),l('John'))), n(Y_1,l('John'),n(Y_2,l(knows),l('Mary'))),
n(Z_1,l('John'),n(Z_2,l(likes),l('Susan'))),

n(R_1,l('Mary'),n(R_2,l(knows),n(R_3,l('John'),n(R_4,l(likes),l('Susan'))))).
The learned rigid grammar assign type t to Mary meaning that Mary may be a correct sentence.
We introduce now the negative type l('Mary',t). Now, no rigid grammar exist. For k=2, the algorithm find 176 solutions. The first solution is :
G = ['John'-[A],'Mary'-[A],'Susan'-[A],knows-[(A\t)/A],likes-[(A\t)/A,(A\A)/A]]
with remaining inequations $t \neq A$ and $(A\backslash t)/A) \neq (A\backslash A)/A)$ □

Learning with semantics
Two recent studies suggest yet another way to further constrain the selection of meaningful grammars, using semantical information. C. Thompson and R. Mooney [30] propose to learn a lexicon of pairs <Word, Meaning> from pairs of <Sentences, Representation>. Although they do not solve, strictly speaking, a grammatical inference problem, they use a kind of lca to produce their results by anti-unification and their problem could be easily set in the framework of lexical grammars learning. In fact, I. Tellier [29] proposes such an approach. Her approach is to learn triples <Word, Type, Logical translation> from pairs <Sentences, Logical translation>. She proposes a sketch of algorithm in three steps :
Starting from an empty current hypothesis H, do the following for each instance <Sentence S, Logical Formula F >

1. For each word of S, compute its possible types, preferring types in H ;
2. For each pair (word,type), compute its possible translations, preferring translations in H ;
3. update the current hypothesis.

Many problems have to be solved before to obtain a concrete and characterizable learning algorithm from this framework. However, it seems to be a very natural and promising setting, at least in the area of natural language processing.

3. Learning regular languages

We address in this second part the more familiar issue in grammatical inference of learning a regular language from sample strings that do or do not belong to the target language. More specifically, we are interested in the inference of deterministic finite automata. Since the work of Colmerauer on metamorphosis grammars [7], the idea of considering the parsing problem of rule grammars in a logic programming framework has been widely spread. However, to our knowledge, no attempt has been made to devise the grammatical inference problem as a unification problem.
We first recall the underlying structure of the search space of deterministic automata. We show in a second step how rational languages may be represented as rational terms and how rational unification allows to infer such languages.

3.1. The space of finite automata, a lattice of terms

Definition 6. (DFA) A finite state automaton is a quintuplet $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ an alphabet, δ a transition function from $Q \times \Sigma$ to 2^Q (extended to $Q \times \Sigma^* \rightarrow 2^Q$), q_0 is the initial state and $F \subseteq Q$ is the set of accepting or final states.
A deterministic automaton (DFA) is such that $\forall q \in Q, \forall a \in \Sigma$, $\delta(q, a)$ has at most one element. A DFA A accepts a regular language $L(A)$.

Definition 7. ($A(L)$, MCA, PTA) I_+ is *structurally complete* wrt A if there exists an acceptance of I_+ such that every transition of A is exercised and every final state of A is used as an accepting state. The canonical automaton of a language L, $A(L)$ is the DFA accepting L which has the minimal number of states.
The maximal canonical automaton with respect to a set of words I, $MCA(I)$, is the automaton A with the largest number of states such that $L(A) = I$ and I is structurally complete with respect to A.
The prefix tree acceptor of I, $PTA(I)$, is obtained from $MCA(I)$ by merging states sharing the same prefixes.

The set of finite automata may be partially ordered with a derivation relation, corresponding to the merging of states in the automata ([19, 28]).

Definition 8. (Derived automaton A/π) Given an automaton $A = (Q, \Sigma, \delta, q_0, F)$ and a partition $\pi = (B_0, B_1, \ldots, B_r)$ of Q, the derived or quotient automaton $A/\pi = (\pi, \Sigma, \Delta, B_0, R)$ is defined as follows:

- $q_0 \in B_0$;
- $R = \{B_i \in \pi, \exists q \in B_i \; st \; q \in F\}$;
- $B_j \in \Delta(B_i, a)$ iff $\exists q \in B_i, \exists q' \in B_j$ such that $q' \in \delta(q, a)$.

It is well known ([11, 9]) that $Lat(PTA(I_+))$, the class of automata derivable from $PTA(I_+)$, together with the derivation relation forms a lattice and that every automaton such that I_+ is structurally complete with respect to its canonical automaton, is learnable in this lattice.

Now, we have to choose a representation of automata that is compatible with a logic programming framework. The idea is to represent automata as infinite terms. We adopt a representation with complete deterministic automata, in order to simplify the explanations. However, our work may be easily extended to take into account incomplete automata.

One associates a distinct variable to each state of the DFA. Output transitions from a given state are represented with the list of states that are reached for each letter in the alphabet. More precisely, each state is represented with a term $state(class, ordered\_set\_of\_successors)$ where $class$ represent the class of the state (e.g. 1 for accepting states and 0 for rejecting states). The representation of a finite

automaton with infinite trees has already been proposed in [8]. In her thesis, the author illustrate the interest of infinite trees by showing how to transform a regular expression into a minimal DFA.

EXAMPLE: Consider a two letters alphabet $\sigma = [a, b]$. Then the term S defined by $S = state(1, [X, S])$ and $X = state(0, [S, X])$ denotes an infinite tree :
$S = state(1, [state(0, [state(1, \ldots), state(0, \ldots)]), state(1, [state(0, \ldots), state(1, \ldots)])])$.
It represents the language $(b^*(ab^*a)^*)^*$, accepting an even number of a . The transition function of the corresponding DFA is $\{\delta(S, a) = X$, $\delta(S, b) = S$, $\delta(X, a) = S$, $\delta(X, b) = X\}$. S is the initial state and the set of final states is $F = \{S\}$.

□

3.2. Learning finite automata

One of the most famous algorithm inferring finite automata from positive and negative instances rpni, has been proposed in [22, 23]. Although this work has been since refined or supplemented by many authors [14, 1, 10, 15, 16], the basic method of applying a state merging technique in $Lat(PTA(I_+))$ is widely used.

We propose thus to implement the behavior of rpni in a logic programming environment. Basically, rpni starts from a current automaton $A = PTA(I_+)$ and while possible, merges pairs of states in A. It consider all pairs in turn, following a fixed ordering that is derived from the standard ordering on states. Indeed, for each state in a PTA, one may associate the word prefix accepted at this state, which may be used as its index. A pair of states may be ordered with the sum of its indexes. In the following, the level of a state is simply the length of its associated word. The strategy of rpni corresponds to a kind of breadth first search through the levels of the PTA.

We propose the following algorithm emulating rpni:

1. Start from the empty level 0 with a term corresponding to the PTA of $I_+ \cup I_-$.
2. Considering increasing levels in the PTA,
   (A) try to unify all states in this level with remaining states of previous levels . If a state may be unified, it is discarded from the level.
   (B) try to unify all pairs of remaining states in this level.
3. halt when all states of the current level have been discarded

The first step of the algorithm may be achieved simply in linear time by inserting training words letter by letter in the PTA. Let $w = w_1 \ldots w_n$ be an instance of class c (positive or negative), written on a fixed ordered alphabet Σ. nth(l,L) denotes the i^{th} element of the ordered set L if l is the i^{th} element of Σ. Let S be the initial state. w is inserted with 2n+1 equations : $S = state(X_1, L_1)$, $S_1 = nth(w_1, L_1)$,$S_1 = state(X_2, L_2)$,..., $S_n = nth(w_n, L_n)$,$S_n = state(c, L_{n+1})$.

Two sets of states are managed by the algorithm : the first one, CL, contains all states of the current level, the second, OS, all states of previous levels that will

remain in the solution automaton. Note that the algorithm works with a single term representing the automaton and "shrinking" each time a merging occurs. Indeed, each step consists just to unify two terms in this automaton, without creating any new structure. This results in a very efficient memory management. Since terms may be infinite, rational unification is required.

The states of CL are computed as the set of immediate successors of the remaining states of the previous level. Together with the fact that the current automaton as never more states than the initial PTA, this ensures the algorithm always terminates. Indeed, each level is finite and states remaining in this level after mergings will never be discarded thereafter. Thus, the set OS is strictly increasing. Since its size is bounded with the size of the PTA, the set CL has to become empty after a finite number of states.

EXAMPLE: Consider a two letters alphabet $\sigma = [a, b]$, a set of positive instances $P = \{aa, b\}$ and the set of negative instances $N = \{a, ab\}$. The extended PTA is represented with the set of equations
$\{S = state(X, [A, B]), A = state(0, [AA, AB]), B = state(1, [BA, BB]),$
$AA = state(1, [L_1, R_1]), AB = state(0, [ABA, ABB])\}$
We recall that CL is the set of states of the current level and OS the set of states remaining in previous levels. The steps of the algorithm introduces the following equations :
Initially, $OS = \emptyset, CL = \{S\}$. No possible merging.
Step 1 $OS = \{S\}, CL = \{A, B\}$, A and S are not unifiable, $B = S$ gives $X = 1, A = BA, B = BB$
Step 2 $OS = \{S, A\}, CL = \{AA, AB\}$, $S = AA$, S and AB are not unifiable, $A = AB$
Step 3 $OS = \{S, A\}, CL = \emptyset$, the algorithm halts. The resulting automaton has 2 states and accepts the language described in the previous example. □

We give a prolog implementation of this algorithm in the appendix. In fact, it is an improvement with respect to rpni since we are able to enumerate all possible solutions in the lattice, simply using prolog non-determinism to try alternative solutions. This is a definite advantage when looking for a smallest automaton with a set of instances that is to small to include a characteristic set.

4. Conclusion

This paper has developed the thesis that, with appropriate reductions, unification may be the central operation involved in grammatical inference. We have applied this approach to the two main types of grammars, lexical grammars and rule grammars, through the cases of categorial grammars and finite automata. Although the logical setting is very natural for categorial grammars, it has been less studied for regular grammars, probably because a lot of efficient representations already exist in this case.
We give in the appendix a large place for prolog programs corresponding to concepts described in the paper. Our aim is not only to allow the reader to play with and

check by himself small illustrating examples. The appropriateness between the chosen representation and the programming language also allows to design very small programs (what Beckert and Possega call "lean" programs" [3]) that help a deep understanding of the computation. Advantages conveyed by such an approach are known : 1) programs are generally efficient, due to the intensive use of a single simple operation, unification, that has been optimized in time en space in the prolog compiler. 2) programs are easily extendable since the effects of changes are easily circumscribed and side effects more predictable. The cleaner and most efficient way to extend the program is to implement extended unification operations. We have tried to illustrate this point in the case of categorial grammars (k-unification, optimal unification, constrained or-unification with inequalities), but it is also true for regular inference algorithms.
Finally, we leave open a number of questions. One of the most ambitious is probably to establish a closer connection between the inference of lexical grammars and the inference of rule grammars.

Acknowledgments

We would like to thank C. Retoré for pointing us to the work of M. Kanazawa and L. Trilling for suggesting, two or three years ago, the idea to work on infinite trees for the representation of regular grammars.

Appendix

Prolog programs for grammatical inference

We give a sicstus prolog code, using simple sicstus standard libraries and two infix operators to build types in the lexicon

```
(:- use_module([library(assoc),library(terms),library(lists)]).
:- op(400,xfx,\), op(400,xfx,/).)
```

```
1.                                          % rg learns rigid grammars
2.  rg(Structures,RG):-
3.      empty_assoc(Empty), rg(Structures,Empty,ARG), assoc_to_list(ARG,RG).

4.  rg([S|Structures],InitRG, RG):-
5.      insert_structure(S,t,InitRG,NewRG), rg(Structures,NewRG, RG).
6.  rg([],RG,RG).

7.  insert_structure(n(Dir,LPart,RPart),Type,Init,RG):-
8.      structure2type(Dir,LPart,RPart,Func,Arg,Type,Type_Func,Type_Arg),
9.      insert_structure(Arg,Type_Arg,Init,New),
10.     insert_structure(Func,Type_Func,New, RG).
11. insert_structure(l(Symbol),Type,Init,RG):-
12.     (get_assoc(Symbol,Init,Value) ->
13.         (unify_with_occurs_check(Value,Type), RG=Init);
14.         put_assoc(Symbol,Init,Type,RG)).

15. structure2type(b,Arg,Func,Func,Arg,Type,Type_Arg\TType,Type_Arg).
16. structure2type(f,Func,Arg,Func,Arg,Type,Type/Type_Arg,Type_Arg).

17.       % rgN learns grammars with a given number of primitive types
18. rgN(List_structures,RG,N):-
19.     rg(List_structures,RG), term_variables(RG,LV), length(LV,M), N>=M.

20.                     % rg_min_size learns grammars of minimum size
21. rg_min_size(List_structures,RG):-
22.     findall(X,rg(List_structures,X),[P|LX]),
23.     size(P,SP), min_size(LX,[P],SP,LMin), member(RG,LMin),
24.     rg(List_structures,RG), acyclic_term(RG).

25. min_size([X|LX],PrevLMin,Min,LMin):-
26.     size(X,SX), (SX=<Min ->
27.     (NewMin=SX, (SX=Min->NewLMin=[X|PrevLMin];NewLMin=[X]));
28.         (NewMin=Min, NewLMin=PrevLMin)),
29.     min_size(LX,NewLMin,NewMin,LMin).
30. min_size([],LMin,_,LMin).
```

```
31. size(T,S):-
32.     ( compound(T) -> ( is_list(T) -> sum_sizes(T,0,S);
33.     (T=..LT, sum_sizes(LT,0,S))); S=1).

34. sum_sizes([],I,I).
35. sum_sizes([T|LT],I,S):- size(T,ST), J is I+ST, sum_sizes(LT,J,S).


36.                                     %kvg learns k-valued grammars
37. kvg(List_structures,K,KVG):-
38.     empty_assoc(Empty), kvg(List_structures,K,Empty, AKVG),
39.     assoc_to_list(AKVG,RKVG), prune_vars(RKVG,KVG).

40. kvg([S|List_structures],K,Init, RG):-
41.     insert_structure(S,t,Init,Temp,K),
42.     kvg(List_structures,K,Temp,RG).
43. kvg([],_,RG,RG).

44. insert_structure(n(Dir,LPart,RPart),Type,Init,RG,K):-
45.     structure2type(Dir,LPart,RPart,Func,Arg,Type,Type_Func,Type_Arg),
46.     insert_structure(Func,Type_Func,Init,New,K),
47.     insert_structure(Arg,Type_Arg,New, RG,K).
48. insert_structure(l(Symbol),Type,Init,RG,K):-
49.     (get_assoc(Symbol,Init,LValues) -> (or_unify(LValues,Type,[]), Init=RG);
50.     (length(LT,K), LT=[c(Type)|_], put_assoc(Symbol,Init,LT-[],RG))).

51. or_unify([Var|_],Type,P):- var(Var), !, Var=c(Type), non_member(Var,P).
52. or_unify([c(Type)|_],Type2,_):- unify_with_occurs_check(Type,Type2).
53. or_unify([A|LValues],Type,P):- or_unify(LValues,Type,[A|P]).

54. prune_vars([X-(A-_)|S],[X-B|R]):- prune_var(A,B), prune_vars(S,R).
55. prune_vars([],[]).

56. prune_var([A|S],[X|R]):- nonvar(A), !, A=c(X), prune_var(S,R).
57. prune_var(_,[]).
```

```
58.        % kvg_pn learns k-valued grammars from positive instances and
negative types
59.  kvg_pn(List_structures_pos,List_neg_types, K,KVG):-
60.  empty_assoc(Empty),
61.  insert_neg_type(List_neg_types,Empty,Init,K),
62.  kvg(List_structures_pos,K,Init,AKVG),
63.  assoc_to_list(AKVG,RKVG), prune_vars(RKVG,KVG).

64.  insert_neg_type([l(Symbol,Type)|S],Init,KVG,K):-
65.  (get_assoc(Symbol,Init,LValues-NegValues) ->
66.  put_assoc(Symbol,Init,LValues-[c(Type)|NegValues],Temp);
67.  (length(LType,K), put_assoc(Symbol,Init,LType-[c(Type)],Temp))
68.  ),
69.  insert_neg_type(S,Temp,KVG,K).
70.  insert_neg_type([],KVG,KVG,_).
```

```
1.          % rpni learns finite automata from positive and negative strings
2.  rpni(Axiom):- rpni_level_by_level([Axiom],E-E).

3.  rpni_level_by_level([],_):-!.
4.  rpni_level_by_level(States_of_the_level, Old_states-F):-
5.      product_merge_levels(Old_states-F,States_of_the_level,Rest_States_level),
6.      merge_one_level(Rest_States_level,Kernel_of_the_level,New_States),
7.      (Kernel_of_the_level=[] ->F=NF; F=[Kernel_of_the_level|NF]),
8.      rpni_level_by_level(New_States,Old_states-NF).

9.  merge_one_level([A|L],[A|R],S):-
10.     all_merge_with(L,A,Not_merged), succ_state(A,S-FS),
11.     merge_one_level(Not_merged,R,FS).
12. merge_one_level([],[],[]).


13. product_merge_levels(E-E,F,G):-var(E),!,F=G.
14. product_merge_levels([A|L]-E,To_merge,F):-
15.     product_merge_level(A,To_merge, Not_merged),
16.     product_merge_levels(L-E,Not_merged,F).

17. product_merge_level([A|L],To_merge,F):-
18.     all_merge_with(To_merge,A,Not_merged),
19.     product_merge_level(L,Not_merged,F).
20. product_merge_level([],F,F).


21. all_merge_with([A|L],B,Not_merged):-
22.     merge(A,B,Not_merged-FNot_merged),
23.     all_merge_with(L,B,FNot_merged).
24. all_merge_with([],_,[]).


25. /* one solution version merge(A,A,F-F):-!. */
26. /* all solution version */
27. merge(A,A,F-F).
28. merge(A,_,[A|F]-F).
```

References

1. C. Higuera (de la), J. Oncina, and E. Vidal. Identification of dfa : data-dependent versus data-independant algorithms. In *Grammatical inference Learning Syntax from Sentences, ICGI'96*, pages 311–325, 1996.
2. Y. Bar-Hillel. A quasi-arithmetical notation for syntactic description. *Language*, 29:47–58, 1953. Reprinted in Bar-Hillel, Language and Information Addison-Wesley, 1964, pp75-86.
3. B. Beckert and J. Posegga. Logic programming as a basis for lean automated deduction. *Journal of Logic Programming*, 28(3), 1996.
4. W. Buszkowski. *Categories, polymorphism and unification*, chapter Discovery procedures for categorial grammars. University of Amsterdam, 1987.
5. W. Buszkowski, W. Marciszewski, and J. van Benthem, editors. *Categorial grammars*. John Benjamins, Amsterdam, 1988.
6. W. Buszkowski and G. Penn. Categorial grammars determined from lingistic data by unification. *Studia Logica*, 49:431–454, 1990.
7. A. Colmerauer. *Natural Language Communication with computers*, chapter Metamorphosis grammars. Springer Verlag, Berlin, 1978. First appeared in "les grammaires de métamorphose", GIA, Université de MarseilleII, nov. 1975.
8. S. Coupet-Grimal. *Deux arguments pour les arbres infinis en Prolog*. PhD thesis, Université d'Aix-Marseille II, November 1988.
9. P. Dupont. Regular grammatical inference from positive and negative samples by genetic search : the gig method. *ICGI'94, Grammatical inference and Applications*, pages 236–245, 1994. Springer Verlag.
10. P. Dupont. Incremental regular inference. In L. Miclet and C. de la Higuera, editors, *Grammatical Inference: Learning Syntax from Sentences, ICGI'96*, volume 1147 of *Lecture Notes in Artificial Intelligence*, pages 222–237. Springer Verlag, September 1996.
11. P. Dupont, L. Miclet, and E. Vidal. What is the search space of the regular inference ? In *ICGI'94, Grammatical inference and Applications*, 1994.
12. M. Kanazawa. *Learnable classes of categorial grammars*. Studies in Logic, Language and Information. CSLI Publications and FoLLI, Leland Standford Junior University, 1998. Also a 1994 PhD dissertation at Standford University.
13. J. Lambek. The mathematics of sentence structure. *American Mathematical Monthly*, 65:154–170, 1965. Reprinted in Categorial grammars, W. Buszkowski & al., John Benjamins, 1988.
14. K. Lang. Random dfa's can be approximately learned from sparse uniform examples. *5th ACM workshop on Computation Learning Theorie*, pages 45 – 52, 1992.
15. K. Lang. Merge order count. Technical report, NEC, 1997.
16. K. Lang, B. Pearlmutter, and R. Price. Results of the abbadingo one dfa learning competition and a new evidence driven state merging algorithm. In Vasant Honavar and Giora Slutzki, editors, *Fourth International Colloquium on*

Grammatical Inference (ICGI-98), Ames, Iowa, USA, volume 1433 of *LNAI*, Berlin, July 1998. Springer Verlag.

17. J.-L. Lassez, M.J. Maher, and K. Mariott. *Foundations of deductive databases and logic programming*, chapter Unification revisited, pages 587–625. Morgan Kaufman, Los Altos, CA, 1987.
18. J. Marciniec. Learning categorial grammars by unification with negative constraints. *Journal of Applied Non-Classical Logics*, 4(2):181–200, 1994.
19. L. Miclet. *Syntactic and Structural Pattern Recognition: Theory and Applications*, chapter Grammatical Inference. World Scientific, 1990.
20. G. Morrill. *Type Logical grammars: Categorial logic of signs.* Kluwer Academic, Dordrecht, 1994.
21. R. Oehrle, E. Bach, and D. Wheeler, editors. *Categorial grammars and Natural language structures.* Reidel, Dordrecht, 1988.
22. J. Oncina and P. García. *Pattern Recognition and Image Analysis*, chapter Inferring Regular Languages in Polynomial Update Time. World Scientific, 1992.
23. J. Oncina and P. García. *Pattern Recognition and Image Analysis*, chapter Identifying Regular Languages in Polynomial Time. World Scientific, 1992.
24. G. Plotkin. *Machine Intelligence*, volume 5, chapter A note on inductive generalization, pages 153–163. American Elsevier, New-York, 1970.
25. J. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41, 1965.
26. Y. Sakakibara. Learning context-free grammars from structural data in polynomial time. *Theoret. Comput. Sci.*, 76:223–242, 1990.
27. Y. Sakakibara. Efficient learning of context-free grammars from positive structural examples. *Information and Computation*, pages 23–60, 1992.
28. J. Carr T. Pao. A solution for the syntactic induction inference problem for regular languages. *Computer Languages*, 1978.
29. I. Tellier. Meaning helps learning syntax. In *Fourth International Colloquium on Grammatical Inference (ICGI-98)*, 1998.
30. C. Thompson and R. Mooney. Semantic lexicon acquisition for learning natural language interfaces. In *6th Workshop on Very Large Corpora*, August 1998.
31. M. Wood. *Categorial grammars.* Routledge, London, 1993.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105,
78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS
Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
(France)
http://www.inria.fr
ISSN 0249-6399